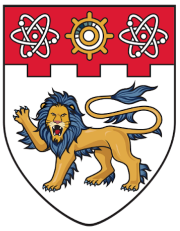

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

MA4830 Realtime Software For Mechatronic Systems

AY 24/25 Sem 2

Major CA Waveform Generator

SCHOOL OF MECHANICAL AND AEROSPACE ENGINEERING

Group Members' Names and Photo

From left to right:

Sin Jun Yuan (U2222037J)

Josef Alexis Estoque Mercado (U2122702K)

Darren Suen Wei Jie (U2121258B)

Baick Geunmyeong (U2023753J)

Chan Shawn Keng Kiat (U2121744C)

Rhee Sung-Jae (U2023183K)

TABLE OF CONTENTS

1. Introduction

In MA4830 Real-time Software for Mechatronics, we covered concepts relevant to real-time systems including embedded systems, processes, and POSIX threads (Pthreads) in the C Programming Language. For our Major CA Assignment, we were assigned to write a Wave Generator program to interface with an analog & digital I/O board and oscilloscope applying the above relevant concepts.

Our program has the following features:

1. **Wave Generator** of Different Waveforms (sine, square, triangular, sawtooth, pulse and cardiac),
2. **Visual and Auditory** cues for Generated Wave
3. **Parsing of Command Line Arguments**
4. **A/D Input Control** (amplitude control, frequency control),
5. **File I/O Mechanism** (reading & writing of default settings, .txt files, and log files)
6. **Multi-Thread Operation using mutex** (I/O and waveform generation).

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <signal.h>
#include <hw/pci.h>
#include <hw/inout.h>
#include <sys/neutrino.h>
#include <sys/mman.h>
#include <math.h>
#include <pthread.h>
#include <time.h>
#include <string.h>

#define INTERRUPT       iobase[1] + 0           // Badr1 + 0 : also ADC register
#define MUXCHAN         iobase[1] + 2           // Badr1 + 2
#define TRIGGER         iobase[1] + 4           // Badr1 + 4
#define AUTOCAL         iobase[1] + 6           // Badr1 + 6
#define DA_CTLREG       iobase[1] + 8           // Badr1 + 8

#define AD_DATA         iobase[2] + 0           // Badr2 + 0
#define AD_FIFOCLR      iobase[2] + 2           // Badr2 + 2

#define TIMER0          iobase[3] + 0           // Badr3 + 0
#define TIMER1          iobase[3] + 1           // Badr3 + 1
#define TIMER2          iobase[3] + 2           // Badr3 + 2
#define COUNTCTL        iobase[3] + 3           // Badr3 + 3
#define DIO_PORTA       iobase[3] + 4           // Badr3 + 4
#define DIO_PORTB       iobase[3] + 5           // Badr3 + 5
#define DIO_PORTC       iobase[3] + 6           // Badr3 + 6
#define DIO_CTLREG      iobase[3] + 7           // Badr3 + 7
#define PACER1          iobase[3] + 8           // Badr3 + 8
#define PACER2          iobase[3] + 9           // Badr3 + 9
#define PACER3          iobase[3] +             // Badr3 + a
#define PACERCTL        iobase[3] +a            // Badr3 + b
                                b
#define DA_Data         iobase[4] + 0           // Badr4 + 0
#define DA_FIFOCLR      iobase[4] + 2           // Badr4 + 2

#define PI              3.14159265358979323846

int badr[5];                                                    // PCI 2.2 assigns 6 IO base addresse
          s
```

Figure 1. Headers and Initial Settings for the PCI Board

2. Determining Optimum Number of Steps

As per the project brief, we initially wrote our code to generate a sine wave with 100 steps from 0 to 2π (period of sine wave). However, this number of steps was **too large** for generating waveforms of a wide range of frequencies. The identified problems of using 100 steps are as follows:

1. **Short Sample Intervals**: 100 steps per cycle means that if the frequency of a waveform is 10 kHz, the PCI plots data points every 1 μs. This is a significantly short time interval for one loop iteration (for calculating and plotting the data point) to run.
2. **CPU Overhead**: Code lines including `loop` and `out16()` operations take computation time, which adds to the overall I/O latency for the program. These latencies interfere with the sampling process from time to time, inhibiting the PCI Board from receiving the sampled data at the right time.
3. **DAC Settling Time**: Similar to the issue with CPU Overhead, the PCI Board needs time to update and execute each loop that was transmitted from the computer. Having 100 steps might be an issue as the time to run each step might be less than this required timeframe, thereby resulting in a waveform with a smaller frequency to be displayed on the PCI Display.

For these reasons stated above, we have decided to decrease the number of steps for our waveforms from **100** to **20**. Although having fewer steps might cost our program overall accuracy in sampling, it allows us to maintain an acceptable level of stability and continuity in producing waveforms.

3. Pthreads

As stated in **Section 1**, we decided to use multi-thread processing to allow low-latency control updates and synchronised processing between a mutex and threads. Our program operates with three Pthreads: **Input Thread**, **Waveform Thread, and Analog Input Thread**.

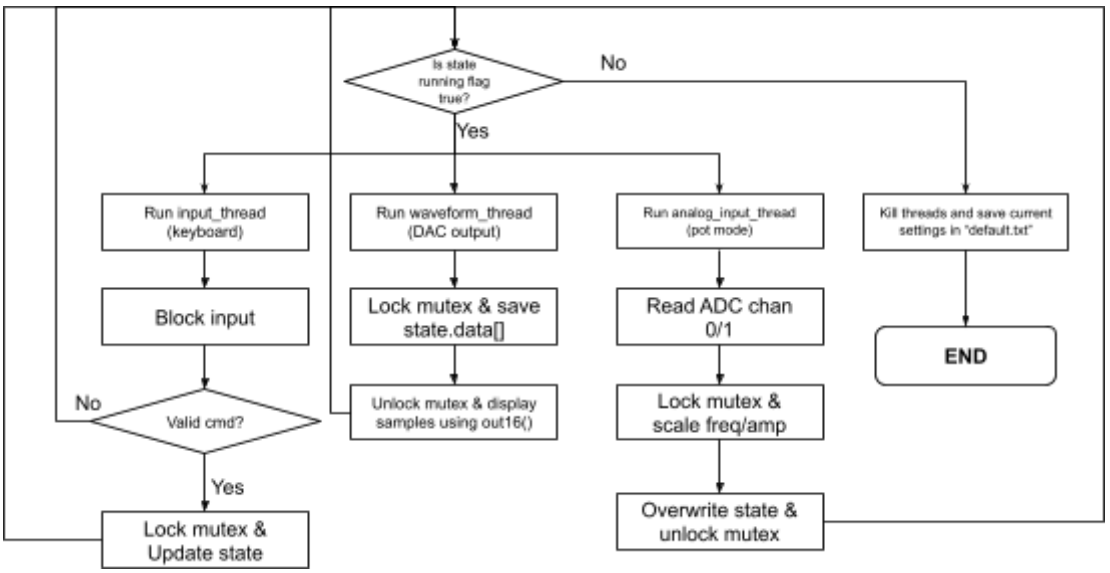

*Figure 2: Pthreads Flowchart (Full Flowchart in **Section 6**)*

3.1 Input Thread - Keyboard Control

In order to keep listening for new user inputs, a thread is created to process inputs and update the saved characteristics of the waveform to be generated. The **`input_thread()`** function uses **`scanf()`** to take user inputs via terminal and parses it to classify which actions to take, namely setting **waveform type,** setting **frequency,** setting **amplitude,** or giving a **help** message to the user.

When taking in new inputs, **characteristics that are not called by the input remain unchanged** so that the waveform maintains similar behaviour from the previous input. Prior to changing the state object's attributes, this thread locks the state object's mutex so that no other thread modifies its information simultaneously. The mutex is then unlocked after updating the relevant variables. This updated information is then processed by the waveform thread to update the generated waveform.

3.2 Waveform Thread - Real-Time DAC Output

The waveform thread constantly checks the latest attributes of the state object and generates the waveform by writing out to the oscilloscope according to the saved characteristics. The generation of the updated waveform is made as a separate thread so as to not be blocked by functions like **`fgets()`** in **`input_thread()`** function and continue generating the waveform while waiting for the user's input. Similar to **`input_thread()`** function, **`waveform_thread()`** function locks the mutex prior to accessing the state object's data to avoid issues of the data being overwritten while it is being read. The thread then unlocks the mutex after outputting the waveform.

3.3 Analog Input Thread - ADC Polling

Similar to **`input_thread()`**, **`analog_input_thread()`** continuously checks for any changes in the analog inputs of the potentiometers of the PCI board. Since this thread only briefly locks around each register access and logging call, interference with the nanosecond-precise waveform output loop.

When the program is signaled to stop, it cleanly exists and enables the main cleanup routine to close files and hardware handles.

4. Key Functions

In this section, we describe the positive attributes of the various functions of our program, as well as their uniqueness.

4.1 Waveform Generator

To enable the switching between the various waveforms, we utilise **`case`** and **`switch`** statements for reduced verbosity and easier scalability of the function should future waveforms be added. This is as opposed to writing several **`else if`** statements. To maintain the atomicity of this function, the function first obtains the mutex to modify the current state of the program using **`pthread_mutex_lock(&mutex)`**. This ensures that any race conditions are avoided should the state be modified by any other active threads. Once the new state has been set by changing **`state.data[i]`**, the mutex is then released to ensure that the function gives up control for other threads.

```
void generate_waveform(void) {
    float delta = (2.0 * PI) / POINTS_PER_CYCLE;
    float value;

    pthread_mutex_lock(&mutex);

    for(i = 0; i < POINTS_PER_CYCLE; i++) {
        switch(state.type) {
            case SINE:
                value = sin(i * delta);
                break;
            case SQUARE:
                value = (i < POINTS_PER_CYCLE/2) ? 1.0 : -1.0;
                break;
            case TRIANGLE:
                value = (2.0 * fabs(i * (2.0/POINTS_PER_CYCLE) - 1.0) - 1.0);
                break;
            case SAWTOOTH:
                value = (double) i / (double) (POINTS_PER_CYCLE -1);
                break;
            case PULSE:
                value = (i < POINTS_PER_CYCLE * PULSE_WIDTH_RATIO) ? 1.0 : 0.0;
                break;
            case CARDIAC:
                t =  (double) i / POINTS_PER_CYCLE;
                value = exp(-200.0 * pow(t-0.2, 2)) - 0.1 * exp(-50.0 * pow(t - 0.35, 2)) + 0.05 * exp
                (-300.0 * pow(t-0.75, 2));
                break;
            case NOTHING:
                value = 1;
                break;
        }

        // Scale value to DAC range and apply amplitude
        state.data[i] = (unsigned int)((value + 1.0) * 0x7fff * state.amplitude / 100);
    }

    pthread_mutex_unlock(&mutex);
}
```

Figure 3. Code Snippet of our Waveform Generating Function, ***void generate_waveform()***

The program currently accepts frequency values between **`float 1`** to **`float 1000`** and amplitude values between **`int 1`** to **`int 100.`** Values inputted beyond the range will not be saved, and a message will prompt users of the acceptable range for the respective inputs.

Once the waveform, amplitude, and frequency values have been inputted, the program produces a visual and audio cue in sync with the wave generated. Several samples of the visual cues are displayed below in *Figure 4*. The audio cue is a beep that sounds after every cycle of the wave. These cues follow any changes to the frequency.

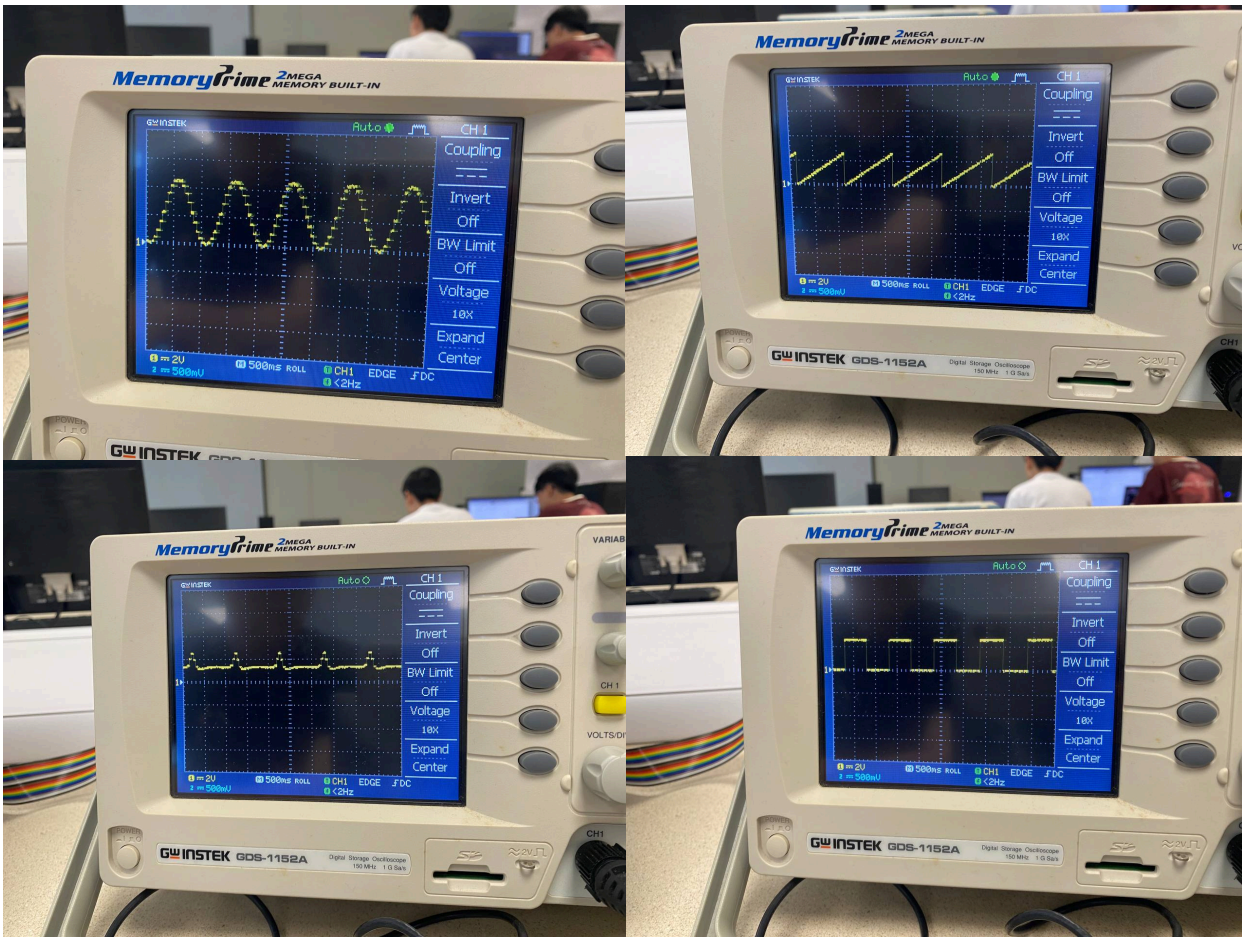

Figure 4. Samples of Visualisation of Sine, Sawtooth, Cardiac, Square
(Top Left, Top Right, Bottom Left, and Bottom Right respectively)

4.2 File Writing

This function enables a user to open a log file of a waveform and recreate it in the programme. **`open()`** is called to open the **`waveform_log.txt`** file, and three different flags are passed to the function to tell the program how to treat the file and set its permissions. The first flag, **`O_WRONLY`**, ensures that the file is opened only for write-only access. **`O_CREAT`** will create the file if it does not already exist, preventing any errors if **`waveform_log.txt`** has yet to be created. Lastly, **`O_TRUNC`** will check if the file exists, and if it does, it will clear its contents so that any new content written is not added to the existing file. If **`open()`** fails, it returns -1 and the program exits.

```c
void write_file(void) {
    char buffer[256];
    int len;

    // Open file with write, create, and truncate flags
    fd = open("waveform_log.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("Cannot open log file");
        exit(1);
    }

    // Write initial settings
    len = sprintf(buffer, "Initial settings:\n");
    write(fd, buffer, len);
    len = sprintf(buffer, "Waveform type: %d\n", state.type);
    write(fd, buffer, len);
    len = sprintf(buffer, "Frequency: %.2f Hz\n", state.frequency);
    write(fd, buffer, len);
    len = sprintf(buffer, "Amplitude: %d%%\n\n", state.amplitude);
    write(fd, buffer, len);
}
```

Figure 5. Code Snippet of our File Writing Function, ***`write_file()`***

To write the header lines, **`sprintf()`** is called to write formatted strings into a buffer. Next, **`write()`** is called to push the bytes from the buffer into the **`waveform_log.txt`** file. This process is repeated for each line written to the file, recording the current state's waveform type, frequency and amplitude.

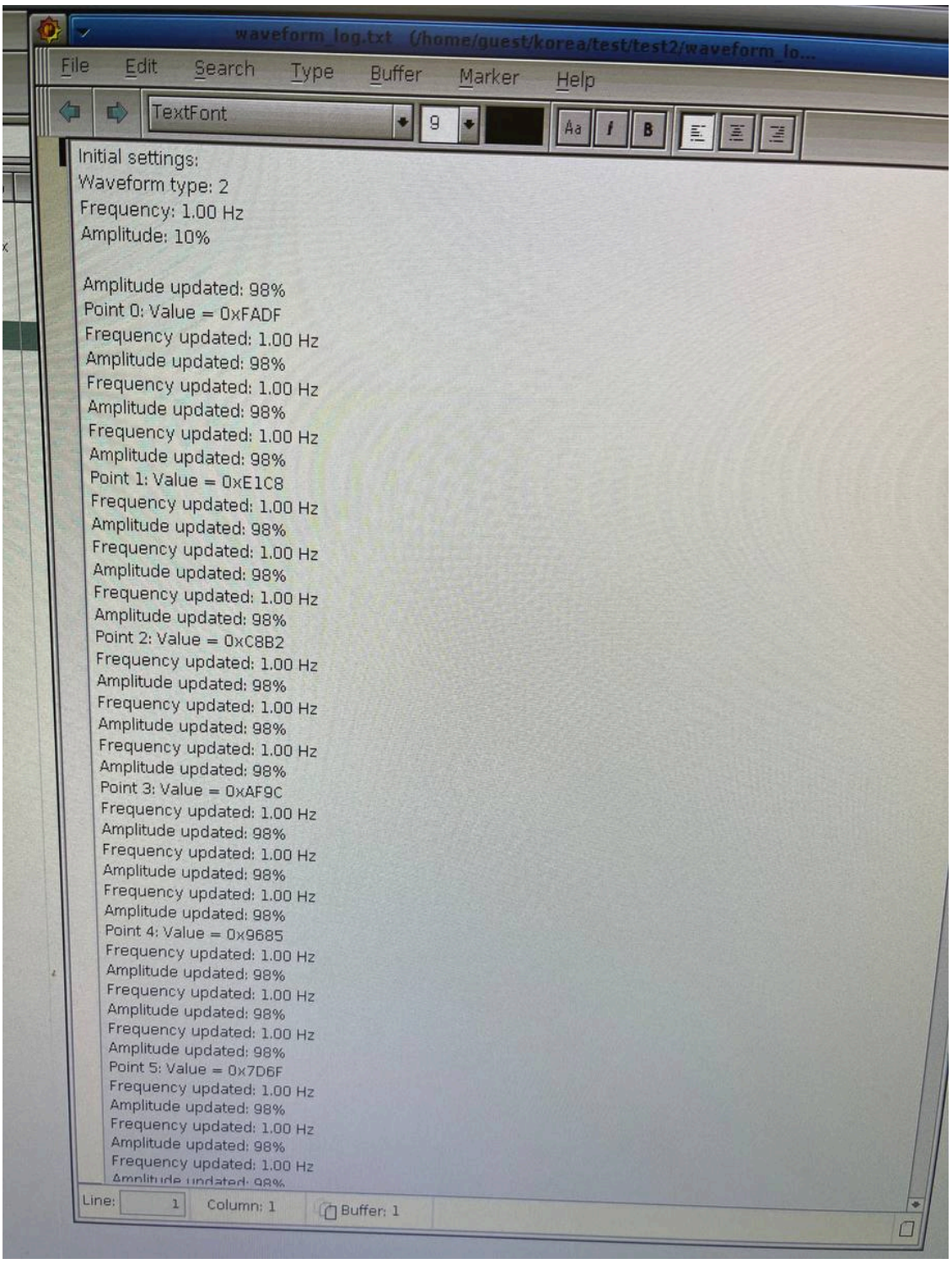

Figure 6. ***waveform_log.txt***

4.3 Potentiometer Control

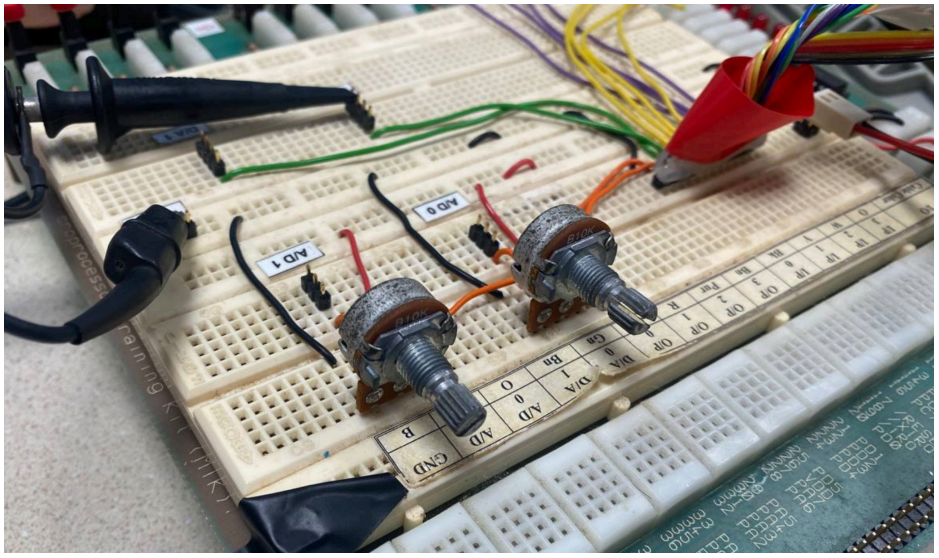

Figure 7. Two Potentiometers Controlling Waveform Amplitude (Left) and Frequency (Right)

On our PCI Board, we have two potentiometers controlling our waveform attributes. As shown in **Figure 2**, the potentiometer connected to **A/D 0** modifies the value of **`pot_freq`**, which is then multiplied with the current value of **`state.frequency`** and saved in a variable called **`scaled_freq`** as a new frequency value. Similarly, the potentiometer connected to **A/D 1** changes the value of **`pot_amp`**, which is then multiplied with the current value of **`state.amplitude`** and saved in a variable called **`scaled_amp`** as a new amplitude value. The value of current voltage of potentiometer, **`adc_in`**, scale for voltage to frequency conversion, **`pot_freq`**, and the value of converted frequency value, **`scaled_freq`**, are then written in our file **`fd`** ("**`waveform_log.txt`**") whenever a change is made. Here is an example of the saved line:

```
write_len = sprintf(log_buffer, "ADC0 (Freq): Raw=%d, Scale=%.3f, Result=%.2fHz\n",
                    adc_in, pot_freq, scaled_freq);
write(fd, log_buffer, write_len);
```

Figure 8. Code snippet of File Write Function to update the value of waveform frequency

The file writing logic remains identical for amplitude control. The program identifies which attribute is being modified by looking at the value of **`count`**, which decides the channel of potentiometer operation (**`count = 0`** → frequency, **`count = 1`** → amplitude).

5. Instructions for Use

5.1 Starting the Program

Upon starting the program, the waveform described by the **command line arguments** (if any) will be generated on the oscilloscope. The example format for the command line argument is as follows:

```
./wg -w sine -f 100 -a 50
```

The explanation of this command line is as follows (the detailed logic with code snippets will be explained later in this section):

1. **`wg:`** this is the name of the compiled program as created by the user and is saved as **`argv[0]`**.
2. **`-w sine:`** when the user inputs **`-w sine`** on the keyboard, **`-w`** is saved as **`argv[i]`**, where **i** = **1** from the initialisation of the for loop inside our **`int main()`**. Using **`strcmp()`** function, our program confirms that the user has inputted **`-w`** and proceeds to check if the user only inputted **`-w`** without the waveform type. If this is the case, the if statement returns 1 and restarts **`int main()`**. However, since **`sine`** is typed by the user and is separated with a space character from **`-w`** in the example, **`i`** increases from 1 to 2, allowing the program to run the codes for the wave argument. Our program uses **`strcmp()`** function again to check if it is equal to **`sine`**. As they are equal, we assign **`sine`** to our **`state.type`** attribute.
3. **`-f 100`**: the exception handling logic is similar to that of **`-w sine`**; however, since **`state.frequency`** is supposed to be saved as a **`float`**, we utilise **`atoi()`** function to convert **`str 100`** to **`float 100`**.
4. **`-a 50:`** the logic is identical to that of **`-f 100`**.

Figure 9 below shows code snippets from declaration of **`struct state`** and its initialisation in **`int main()`**. By creating a struct to manage all our variables as its attributes, the readability of our code is enhanced.

```c
// Global variables
struct {
    enum WaveformType type;
    float frequency;
    int amplitude;
    int running;
    unsigned int* data;
    unsigned short chan;
;
} state;
```

```c
int main(int argc, char* argv[]) {
    // Initialize state with defaults
    state.type = SINE;
    state.frequency = 1;
    state.amplitude = 100;
    state.running = 1;
    state.data = malloc(POINTS_PER_CYCLE * sizeof(unsigned int)); // memory allocation of steps
    pthread_mutex_init(&state.mutex, NULL); // initialisation of mutex for multi-threading
```

Figure 9. Implementation of ***`struct state`*** *with Different Attributes (Top) and Initialisation of State with Default Values (Bottom)*

After starting the program, the program will ask the user for the waveform generation mode that they want. The user can input 1 or 2 for “Keyboard Input” and “Analog Input” modes respectively.

5.2 Terminal Inputs

After selecting “Keyboard Input” mode, the terminal window will prompt the user to input changes to modify the generated waveform on the oscilloscope. Some inputs include the following:

Function	**Input**	**Response**
Help	**`help`**	Explains how to key in inputs to change specific characteristics.
Change Waveform **Type**	**`sine, triangular, square, sawtooth, pulse, cardiac`**	Changes the waveform type to the input shape while maintaining its frequency and amplitude.
Change Wave **Frequency**	**`freq 1 to 1000`** (e.g. **`freq 50`**)	Updates the frequency (Hz) of the waveform to the new input while maintaining its type and amplitude.
Change Wave **Amplitude**	**`amp int 0 to 100`** (e.g. **`amp 50`**)	Updates the amplitude of the waveform to the new amplitude while maintaining its type and frequency.
Save waveform to .txt file	**`save <filename>`** (e.g. **`save test.txt`**)	Saves current waveform settings to the specified file.
Load .txt file	**`load <filename>`** (e.g. **`load test.txt`**)	Loads waveform settings from the specified file and generates waveform.
Quit	**`quit`**	Exits the program.

Table 1. Key Functions of Command Line Arguments

5.3 Analog Input

After selecting “Analog Input” mode, the waveform type generated will be loaded from the **`default.txt`** configuration file. The user can control the frequency and amplitude of the waveform through two potentiometers. The A/D0 potentiometer adjusts the frequency from 1 to 1000 Hz, while the A/D1 potentiometer adjusts to amplitude from 0 to 100%.

5.4 Waveform Log

As previously mentioned in Section 4.2, the program keeps a record of the various wave settings and parameters used throughout the runtime of the wave generator.

5.5 Auto-Save on Exit

When the program is terminated, the program saves the memory of the previous settings in an automatically saved file with the **`save_default_settings()` function.** This enhances user experience by protecting the data of the previously generated waveform.

5.6 Controlled Termination of Program

To kill the program, a “**`quit`**” input can be given to exit the program if it were running in “Keyboard Input” mode. Else, holding down ‘Ctrl’ and ‘C’ at the same time works to exit the program as well. This sends an interrupt signal to the program, causing it to terminate. The oscilloscope will stop generating the waveform.

6. Appendix

Logic Flowchart

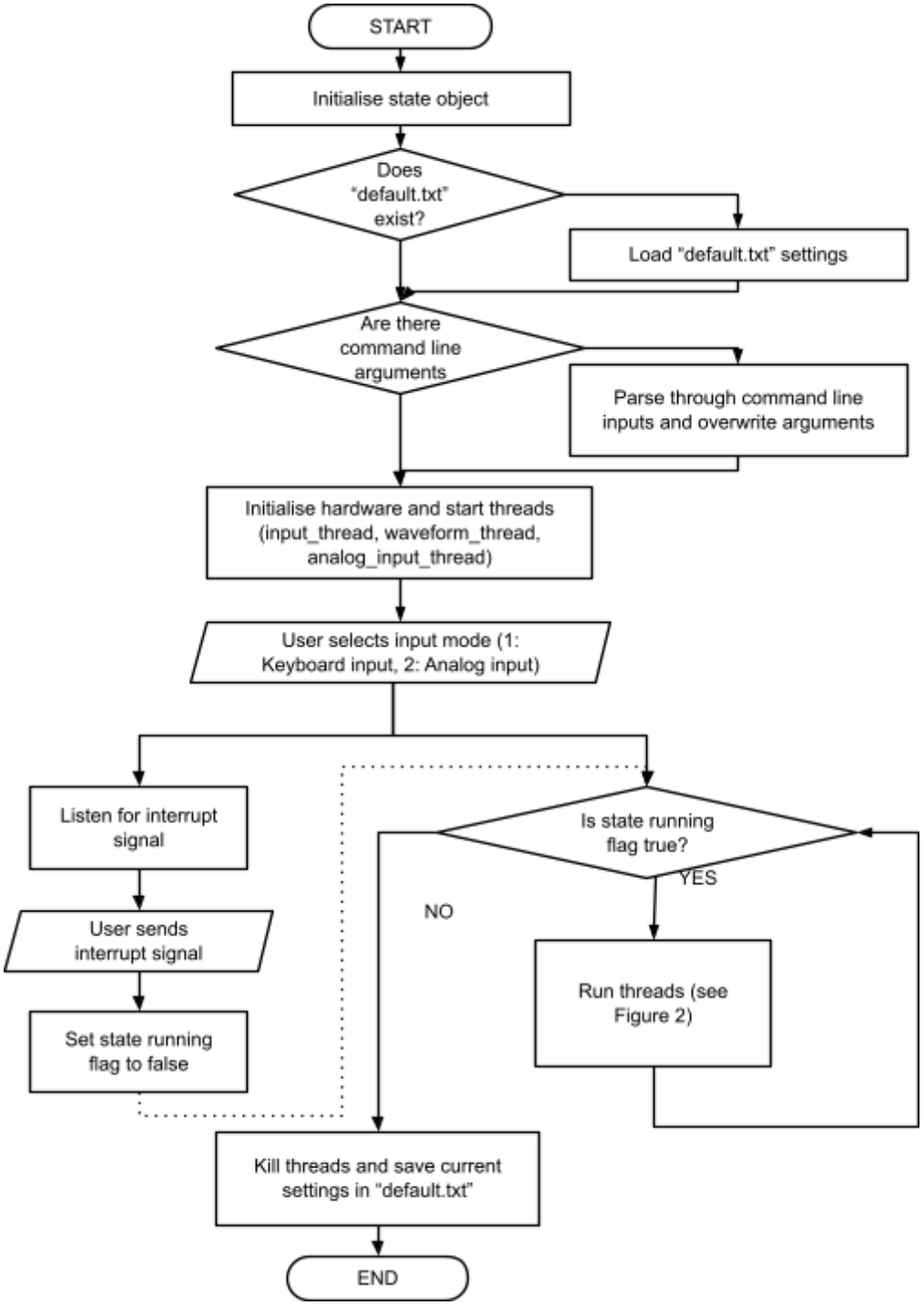

Figure 10. Flowchart Illustrating Logic of Program