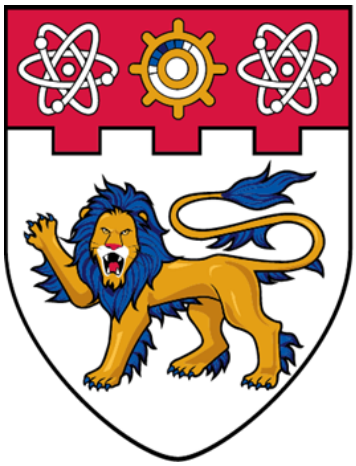


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SCHOOL OF MECHANICAL AND AEROSPACE ENGINEERING

AY 2022/23 S2

MA4830 REALTIME SOFTWARE FOR MECHATRONIC SYSTEMS

Major CA - Wave Generator

Team members:

Greg Angelo Gonzales Nonato (U2021982L)

Lim Jin Hng, Timothy (U1922411K)

Lin YiJuan (U2021676H)

Minn Set Moe Hein (U2022044J)

Mohamed Raizee Bin Mohamed Ibrahim(U2022754H)

Mohammad Uzair Bin Rosman'id (U1921508J)

Table of Contents

1. Introduction

This program is created to generate waveforms based on the user's inputs and project the waveforms on the oscilloscope. It is able to generate four types of waveforms: Sine, Square, Sawtooth, and Triangular. Users can configure the waveform settings such as type of waveform, amplitude, and frequency, by using the keyboard or DAQ board.

Figure 1: Four types of waveforms displayed on oscilloscope

1.1 Key Features

- Realtime User Interface (UI):

  - Realtime update from UI inputs into the program to change parameters of projected waveforms instantaneously (control mode, wave type, amplitude, frequency).

- READ and WRITE parameters into and from a text file:

  - The WRITE feature enables users to save the latest parameters of the waveform onto a text file which will be saved under the current directory.

  - The READ feature enables users to retrieve waveform parameters from the saved file, and serve as the parameters for the initial waveform to be projected.

- Multiple input modes:

  - Users can switch between keyboard mode and DAQ mode by toggling toggle switch 2 (2nd toggle switch from the left).

- When toggle switch 2 is in UP(0) state, it will be in DAQ mode.
- When toggle switch 2 is in DOWN(1) state, it will be in keyboard mode.

- Orderly shutdown of program:
  - This program allows users to exit the program manually for both the keyboard and DAQ mode.

2. User Manual

2.1 Program Execution

1. Compile mainV16.c

```
cc -o mainV16 mainV16.c -lm -lncurses
```

"-lm" is a linker flag that tells the linker to link the "libm" library, which provides math functions like sin, cos, sqrt, etc. This library is required when your program uses any math functions.

"-lncurses" is a linker flag that tells the linker to link the "ncurses" library, which is a library for creating text-based user interfaces in a terminal. This library provides functions for moving the cursor, changing text colors, and displaying menus and dialogs.

2. Execute program

```
./mainV16 sine 2 200
```

The first argument “sine” is to configure the wave type, the second argument “2” is to configure the amplitude of the waveform and the third argument “200” is to configure the frequency of the waveform. The user can also choose not to input any arguments at the point of program execution. In this case, the latest waveform parameters from the previous run is used.

Before executing the program, all the toggle switches should be in the down position. If any of the switches are up when the program is executed, there will be a prompt on the display to switch all the toggle switches to the down position as shown in Figure 2a.

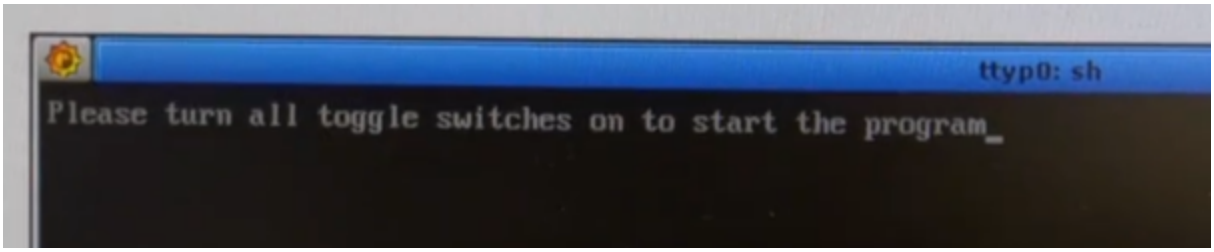


Figure 2a: Prompt when not all the toggle switches are in the down position

Once the program is executed, it will clear out the terminal and display only the necessary instructions and information. As shown in Figure 2b, the instructions for keyboard and DAQ input, and 4 parameters (control mode, wave type, frequency, amplitude) are displayed in real time.

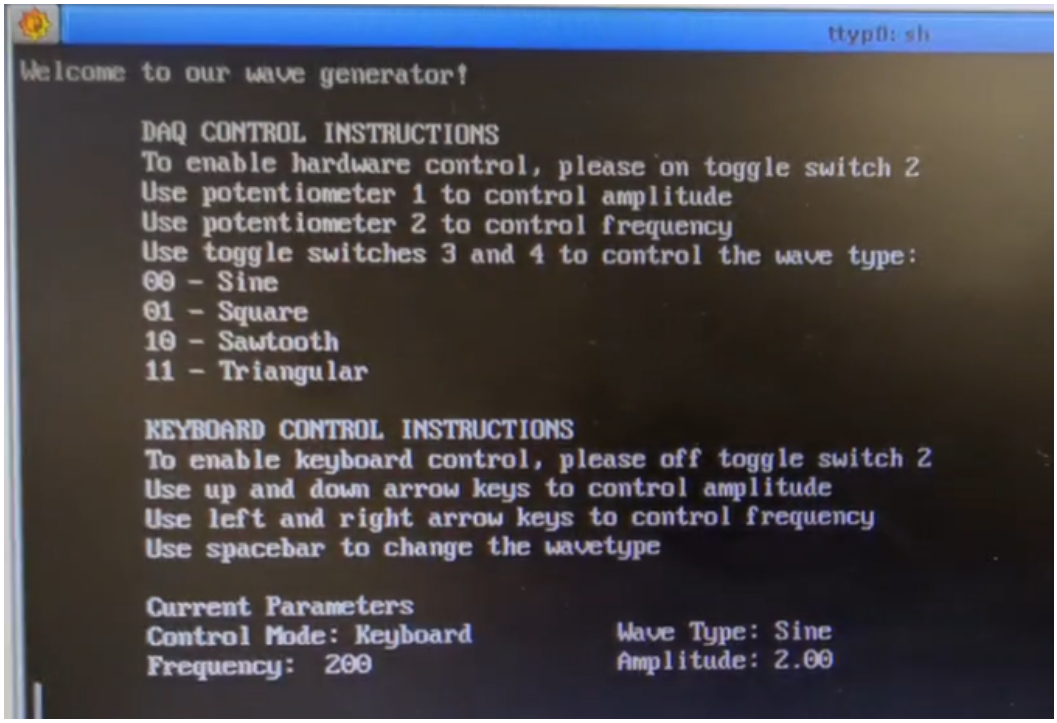


Figure 2b: Display when the program is run

2.2 Generate Initial Waveform and Change Control Mode

The initial waveform and changing of control mode can be triggered by toggle switches. To generate the initial waveform, set all 4 toggle switches in DOWN(1) state, as shown in Figure 4. The toggle switch 2 is used to toggle between DAQ control and keyboard control mode:

- To enable DAQ control mode, set toggle switch 2 in UP(0) state, as shown in toggle switch 2 (TG2) of Figure 3. The waveform parameters can then be adjusted by the toggle switches and potentiometers.
- To enable keyboard control mode, set toggle switch 2 in DOWN(1) state as shown in toggle switch 2 (TG2) of Figure 4. The waveform parameters can then be adjusted by the arrow keys and spacebar on the keyboard.

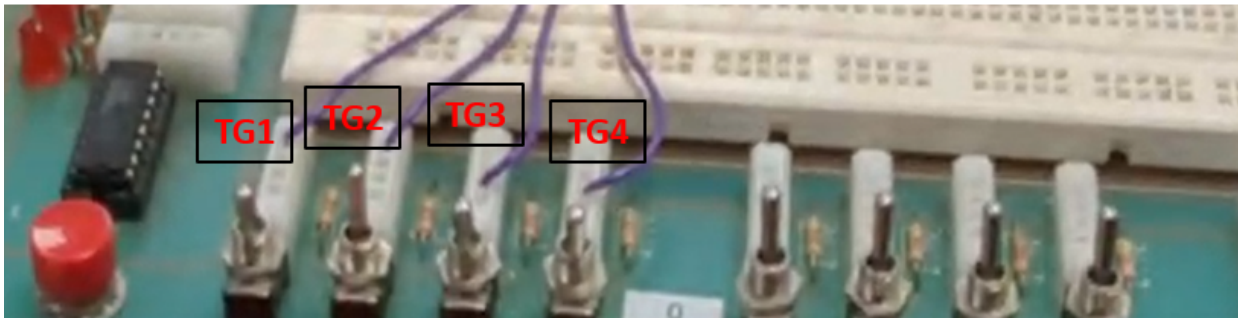


Figure 3: Toggle switches in DAQ(hardware) control mode

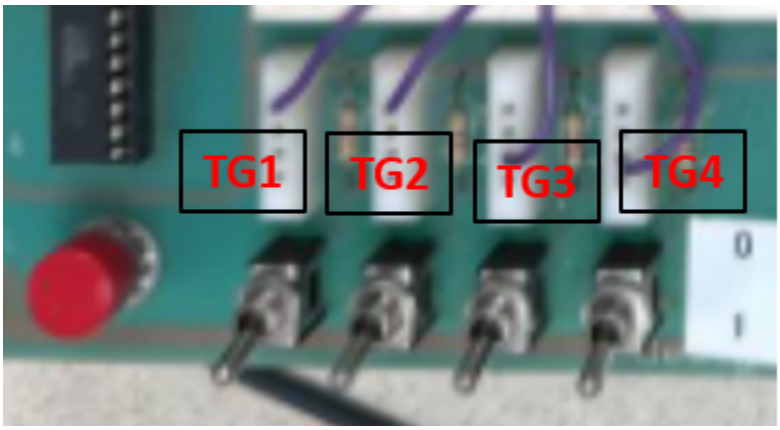


Figure 4: Toggle switches in keyboard control mode

2.3 Changing the Wave Parameters in Real-Time via Data Acquisition Board (DAQ)

The necessary instructions to operate the program are displayed on the terminal. For DAQ mode, the waveform parameters and exiting of program can be configured by using toggle switches and potentiometers as follows:

- Toggle switch 1 (TG1) - acts as an exit button for the program. Set TG1 in UP(0) state to exit program, as shown in Figure 5. The program will then show a message to indicate that the user has exited the program successfully, and that the latest wave configuration has been saved in a text file, as shown in Figure 6.
- Toggle switch 2 (TG2) - acts as a control mode toggle. Set TG2 UP(0) or DOWN(1) to toggle between DAQ or keyboard mode.
- Toggle switch 3 & 4 (TG3 & TG4) - changes the waveform type. Set TG3 and TG4 to 00 for Sine waveform, 01 for Square waveform, 10 for Sawtooth waveform, and 11 for Triangular waveform respectively. An example of waveform type change is shown in Figure 7.
- Potentiometer 1 & 2 (labeled in Figure 5) - changes the amplitude and frequency of the waveforms respectively. Turn the knobs clockwise/anti-clockwise to adjust the amplitude and frequency. Examples of amplitude change and frequency change of the waveform are shown in Figures 8 and 9 respectively.

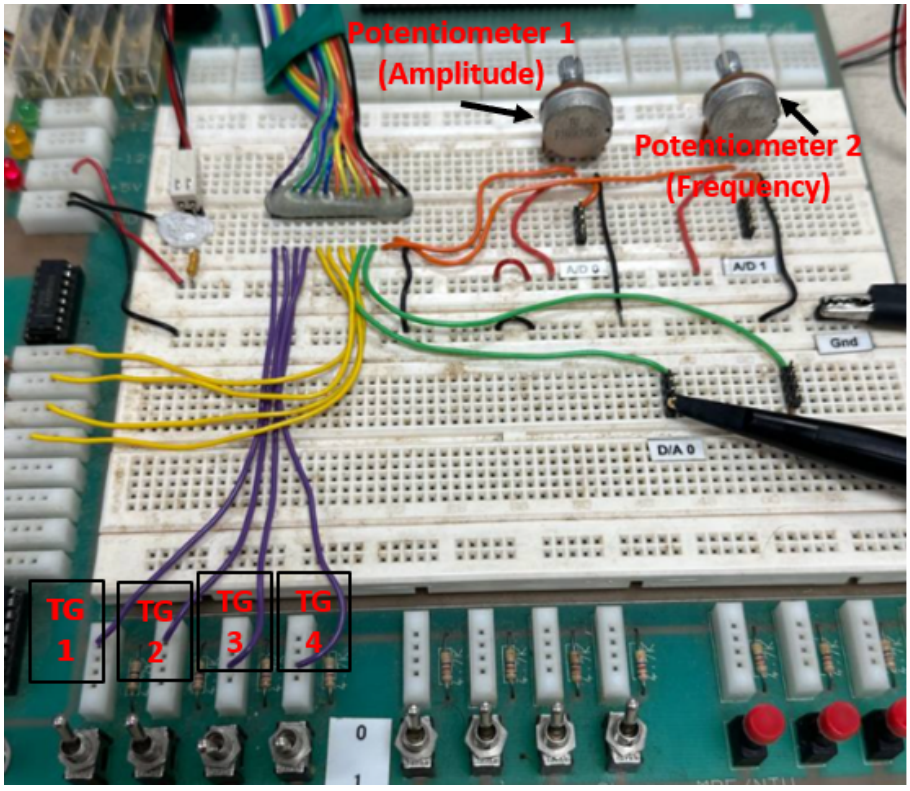


Figure 5: Toggle switches in terminate program configuration

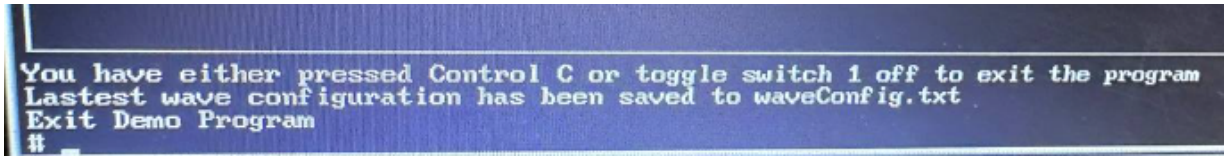


Figure 6: Display for exit program and wave configuration saved

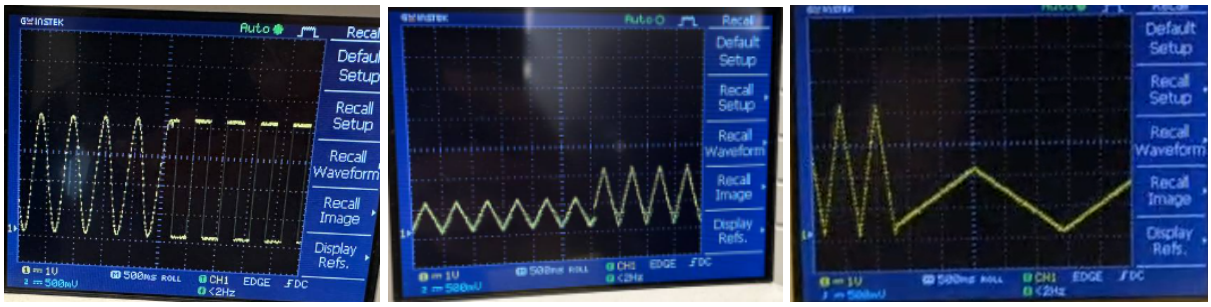


Figure 7, 8 and 9: Waveform generated from different parameters (change in waveform, change in amplitude and change in frequency)

2.4 Changing the Wave Parameters in Real-Time via Keyboard Input

Apart from using the DAQ board to adjust the waveform parameters, this program provides users with the flexibility to switch to keyboard mode via the toggle switch, to change the waveform parameters or exit the program.

The waveform parameters that can be changed by the keyboards are:

- Waveform Type - can be changed by pressing down the spacebar on the keyboard. Pressing down the spacebar will enable users to explore through the list of waveform types available and select from the list (Sine→Square→Sawtooth→Triangular).
- Amplitude - can be changed by pressing the down arrow key to decrease amplitude, and up arrow key to increase amplitude. Amplitude varies by 0.5 each time an up or down arrow key is hit.
- Frequency - can be changed by pressing the left arrow key to decrease frequency, and right arrow key to increase frequency. Frequency varies by 100 Hz each time a left or right arrow key is hit.

Exiting the program via keyboard:

- The program can be exited in the keyboard mode by hitting “Ctrl+C” on the keyboard. The exit message will then be printed to users, same like the DAQ(hardware mode) when users set the TG1 to UP(0) state.

2.5 Read and Write File

The program has a feature where it writes the current wave configuration into a text file called ‘waveConfig.txt’ when the program is exited using the termination process. The program writes the wave type (1-Sine, 2-Square, 3-Sawtooth, 4-Triangular), amplitude and frequency as floating point numbers and integers into the text file, as shown in Figure 10. The wave configuration saved can be used to generate the initial waveform the next time the user runs the program without any arguments.

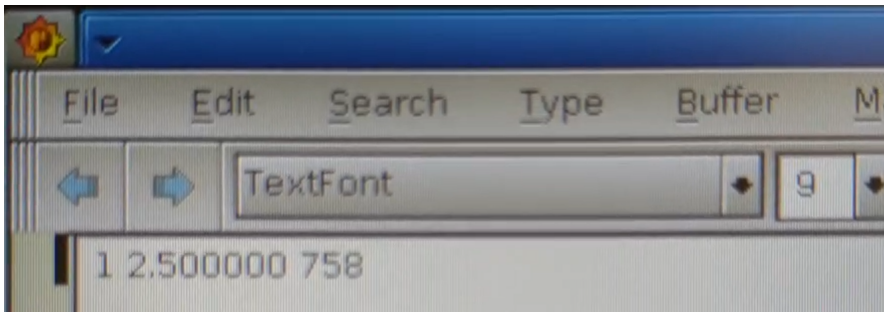


Figure 10: waveConfig.txt file

2.6 Range of Amplitude and Frequency

The minimum and maximum values of the amplitude and frequency variables are shown on Figure 11.

	Min	Max
Amplitude/ V	0.5	2.5
Frequency/ Hz	10	1000

Figure 11: Range of wave parameters

3. Backend Process

3.1 Initialisation

When the program is executed the following initialisation process is implemented in sequence:

1. Definitions and global variables are declared.
2. ncurses screen is initialised with *initscr()* and reading of function keys is enabled with *keypad()*.
3. Assigned signal (SIGINT) with a signal handler (signalHandler).
4. The mapping of the DAQ and PCI board addresses and configuration of the analog and digital input/ output ports.
5. *cmdArg()* called to extract the initial wave parameters from the input arguments.
6. Initial waveform is generated using *waveGeneration()*.
    a. *waveGeneration()* first declares some variables, including "delta," "dummy," "i," and "scale." The "scale" variable is calculated by dividing the amplitude value pointed to by "ptr_amplitude" by 2.5. This results in a scale value that ranges from 0 to 1.
    b. The function then enters a switch statement based on the value pointed to by "ptr_wavetype" where it generates 50 values for each wave, calculates the delta value, multiply by the scale and 0xFFFF (maximum value for a 16-bit unsigned integer and the resulting value is stored in an array called “data”. There are four cases defined in this switch statement:
7. Graphical User Interface (GUI) is generated and displayed onto the terminal by calling *startmsg()*.
8. The following two *pthreads* are created alongside the main thread, using the POSIX thread library function *pthread_create( )*: *thdaq* and *thKey.*
    a. Thread *thdaq*: Executes start routine *adcIn( )* which performs reading and writing procedures to PCI-DAS1602/16 Board.
    b. Thread *thKey*: Executes start routine *event_loop( )* which detects key presses to change *wavetype, frequency,* and *amplitude* from the standard keyboard.

3.2 Execution

3.2.1 *main* Thread

In the main thread, the ADC port is configured to enable the system to perform analog-to-digital conversions. It changes the analog output based on the data array from the *waveGeneration()* function, and the output rate is controlled by the frequency variable. The frequency variable is controlled using *clock_gettime()* function, to measure the time at the start of the loop and then calculates the elapsed time during each iteration of the loop. If the elapsed time is less than the desired period, the loop waits until desired period is reached, and outputs the next data point in the waveform. This process repeats until the program is terminated. Two pthreads are created for the DAQ and keyboard control mode (*thdaq* and *thKey*), to read users input from the DAQ hardware and keyboard. Lastly, the various functions will be activated to perform the necessary tasks of the program. The flow of *main* thread for the program is represented by the flowchart in Figure 12 below.

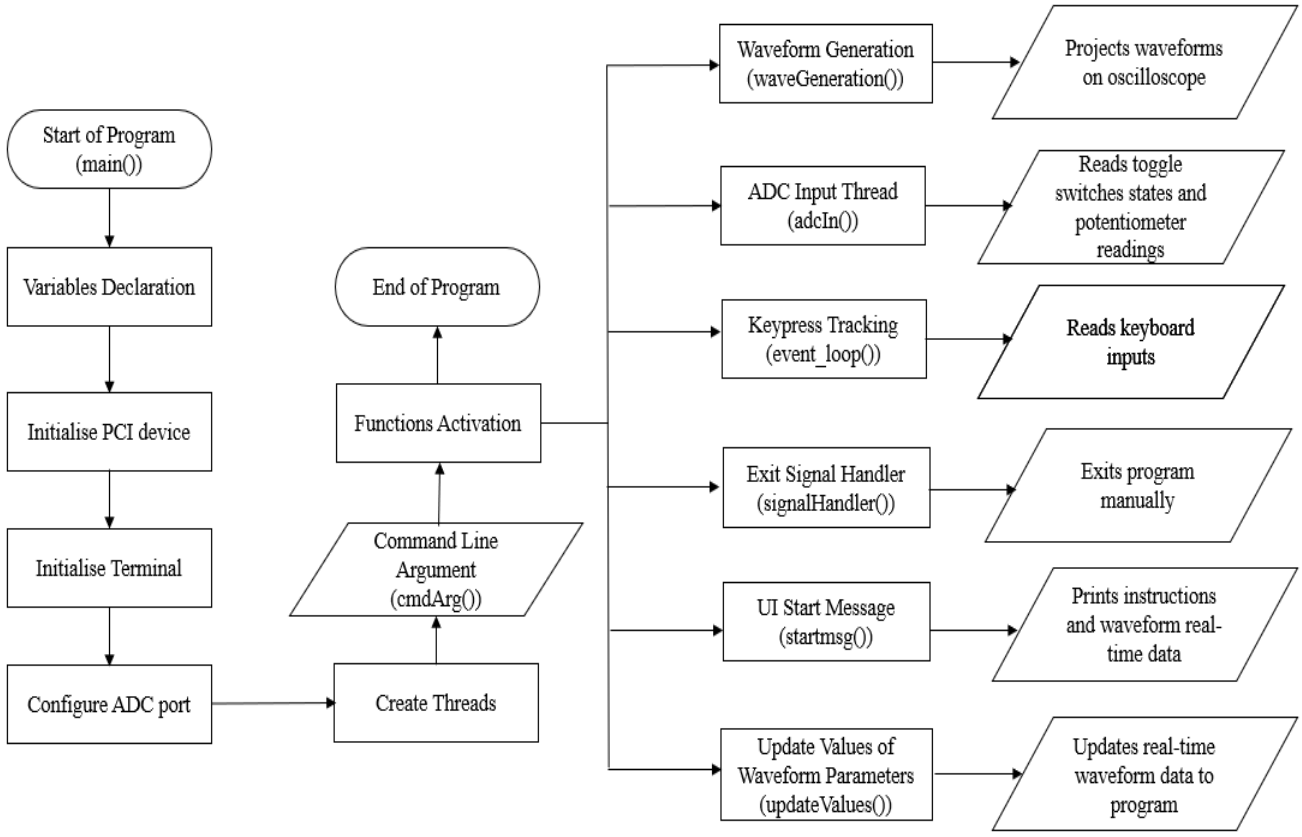


Figure 12: Flowchart for the flow of main thread

3.2.2 *thdaq* Thread

When the program is in DAQ control mode (TG2 in the up (0) state), the *thdaq* thread actively reads the states of the toggle switches and potentiometer readings. The function runs in an infinite loop until it is interrupted by the SIGINT signal (when TG1 is in the down (1) state). Initially, the function sets the channel, initializes the ADC and stores the converted value in the 'adc_in' variable. After reading from the ADC, it reads from the digital input/output (DIO) port A to check the value of the toggle switches. Based on the value of the toggle switches, the control mode and different wave types will be set. Afterwards, it reads from the potentiometers. If the potentiometer 1 knob is turned, adjustments are made to the wave amplitude based on the new knob position:

- adc_in < 0x327C: amplitude = 0.5
- 0x327C < adc_in < 0x6537: amplitude = 1.0
- 0x6538 < adc_in < 0x9783: amplitude = 1.5
- 0x9784 < adc_in < 0xCB6F: amplitude = 2.0
- 0xCB70 < adc_in < 0xFFFF: amplitude = 2.5

If potentiometer 2 is turned, the frequency will be calculated and converted to a range of 10-1000Hz. Prior to changing the control mode, wave type, amplitude, frequency values, *pthread_mutex_lock()* is called, to prevent concurrent editing with the *thKey* thread, and subsequently unlocked using *pthread_mutex_unlock()* after changing those variable values. The function *updateValues()* is called after every variable change to provide a real time update of the wave parameters in the user interface on the terminal. A flowchart of *updateValues()* can be found in Appendix A. The flowchart of the *thdaq* function is represented in Figure 13 below.

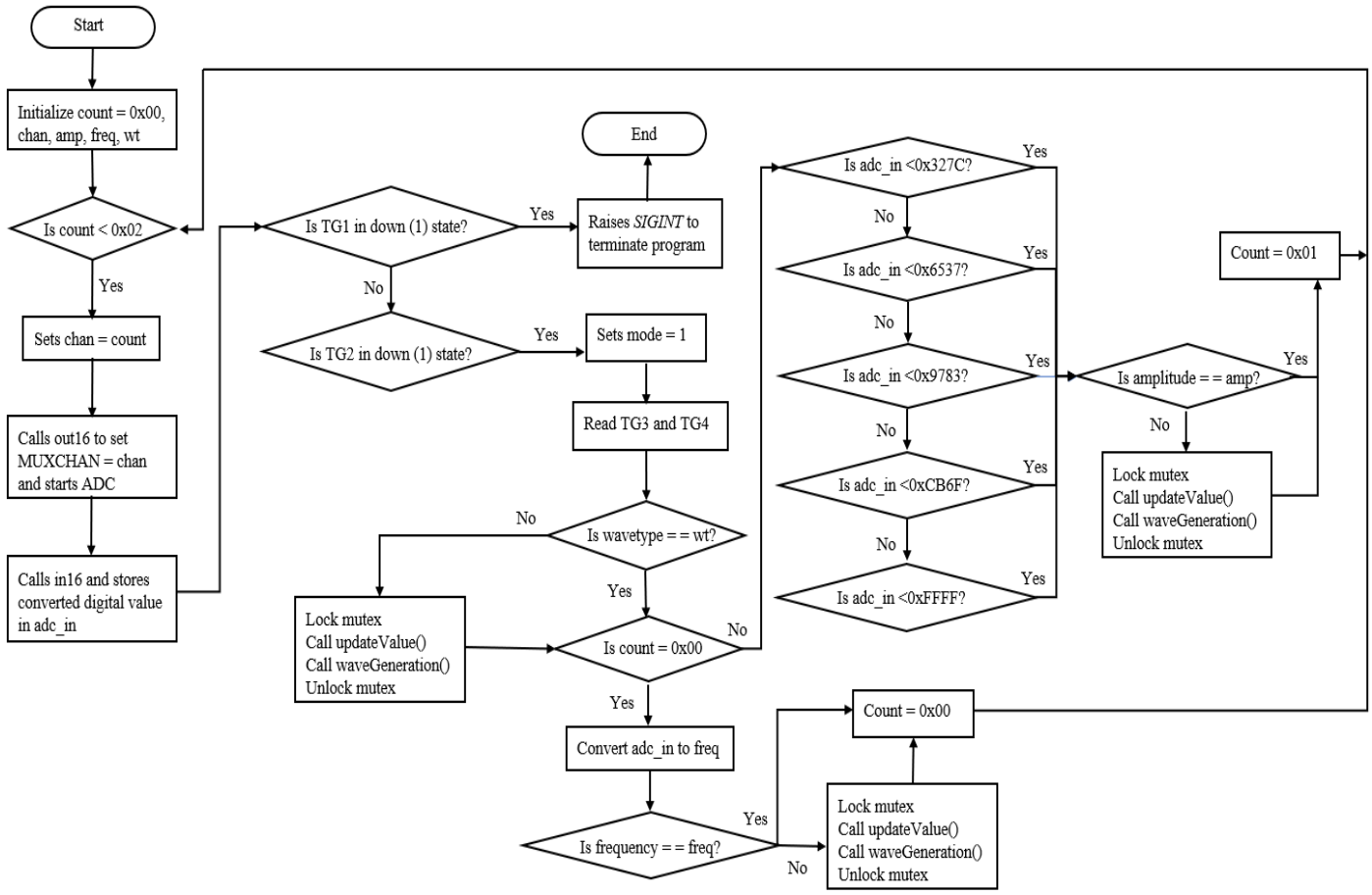


Figure 13: Flowchart for the flow of the *thdaq* thread

3.2.3 *thKey* Thread

When the program is in 'Keyboard' control mode (TG2 in the down (1) state), the *thKey* thread actively listens for key presses by the user on the keyboard. Using the *ncurses* library, the *getch()* function is called and returns the specific KEY_value based on the key pressed by the user. If the key pressed was an arrow key or the spacebar, the following adjustments were made to the wave parameters:

- Up arrow key: Increase amplitude by 0.5
- Down arrow key: Decrease amplitude by 0.5
- Left arrow key: Decrease frequency by 100
- Right arrow key: Increase frequency by 100
- Spacebar: Change wave type between sine, square, sawtooth and triangular

Prior to changing the *amplitude*, *frequency* and *wavetype* values, pthread_mutex_lock() is called, to prevent concurrent editing with the *thdaq* thread, and subsequently unlocked using *pthread_mutex_unlock()* after changing those variable values. The function *updateValues()* is called similarly to the *thdaq* thread. The flowchart is represented in Figure 14 below.

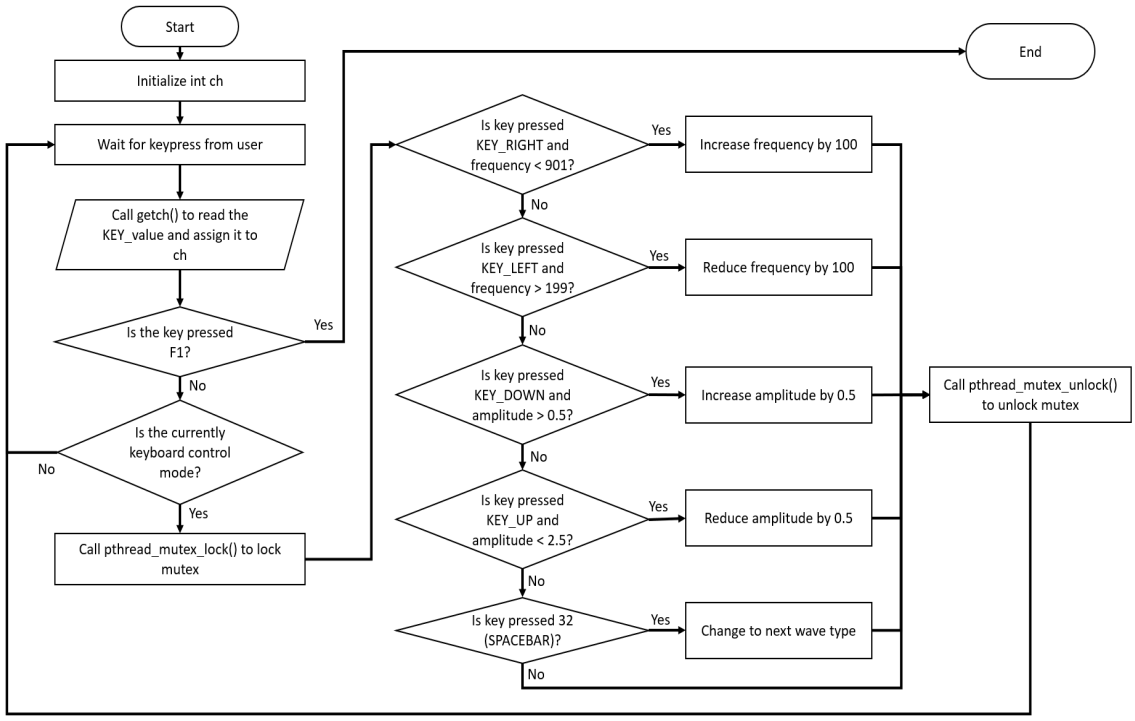


Figure 14: Flowchart for the flow of the *thKey* thread

3.3 Termination

3.3.1 Methods of termination

Termination of the program can be done through 2 methods, by either pressing Ctrl+C or toggling TG1 to 0 through DAQ. 1st Method – **Ctrl + C** command will raise the signal SIGINT using C Library function *signal()* and execute exit signal handling function, *void signalHandler(int signum)* specified by the program. 2nd Method – **Toggle TG1** will raise the signal SIGINT through *raise()* which will invoke *signal()* to execute exit signal handling function. For detailed sequence of execution of the termination procedure, refer to the flowchart in Figure 15 below.

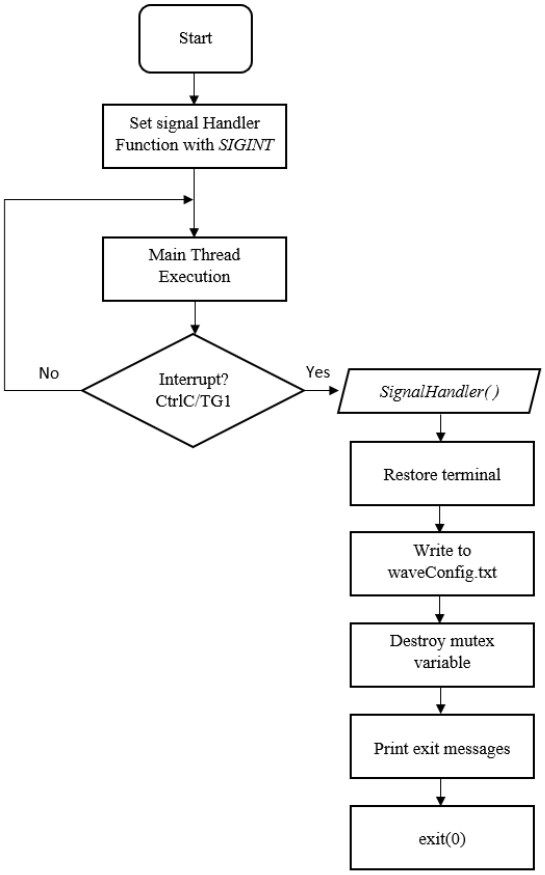


Figure 15: Flowchart of the flow of the termination program

3.3.2 Signal handler for exit procedure

Exit signal handling function *signalHandler(int signum)* is defined by the program. The function will carry out the following exit procedures:

1 – Restore the terminal after *ncurses* activities.

2 – Save the current wave configurations into *waveConfig.txt.*

3 – Destroy *mutex* variables.

4 – Print exit messages.

5 – Exit the program with exit(0) terminating all the threads.

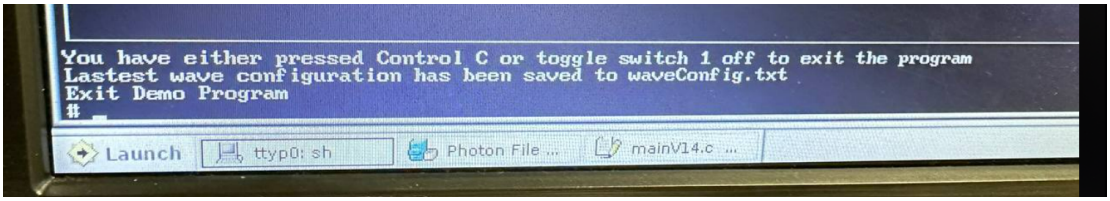


Figure 16: Display for exit program

4. Program Novelty and Limitations

4.1 Novelty

- Able to save current waveform parameters into a file and use them to project waveforms onto the oscilloscope the next time.
- Ease of switching of control mode between keyboard and DAQ system by just toggling the second toggle switch.
- Types of waveforms, amplitude, and frequency can be adjusted by keyboard(arrow keys and spacebar) as well as DAQ system(toggle switches and potentiometers).
- Realtime update from UI inputs into the program to change parameters of projected waveforms instantaneously.
- Allows users to exit the program manually in both keyboard and DAQ mode.
- Mutex to lock global variables so that only one thread can modify them and prevents other threads from modifying them until it gets unlocked.

4.2 Limitations

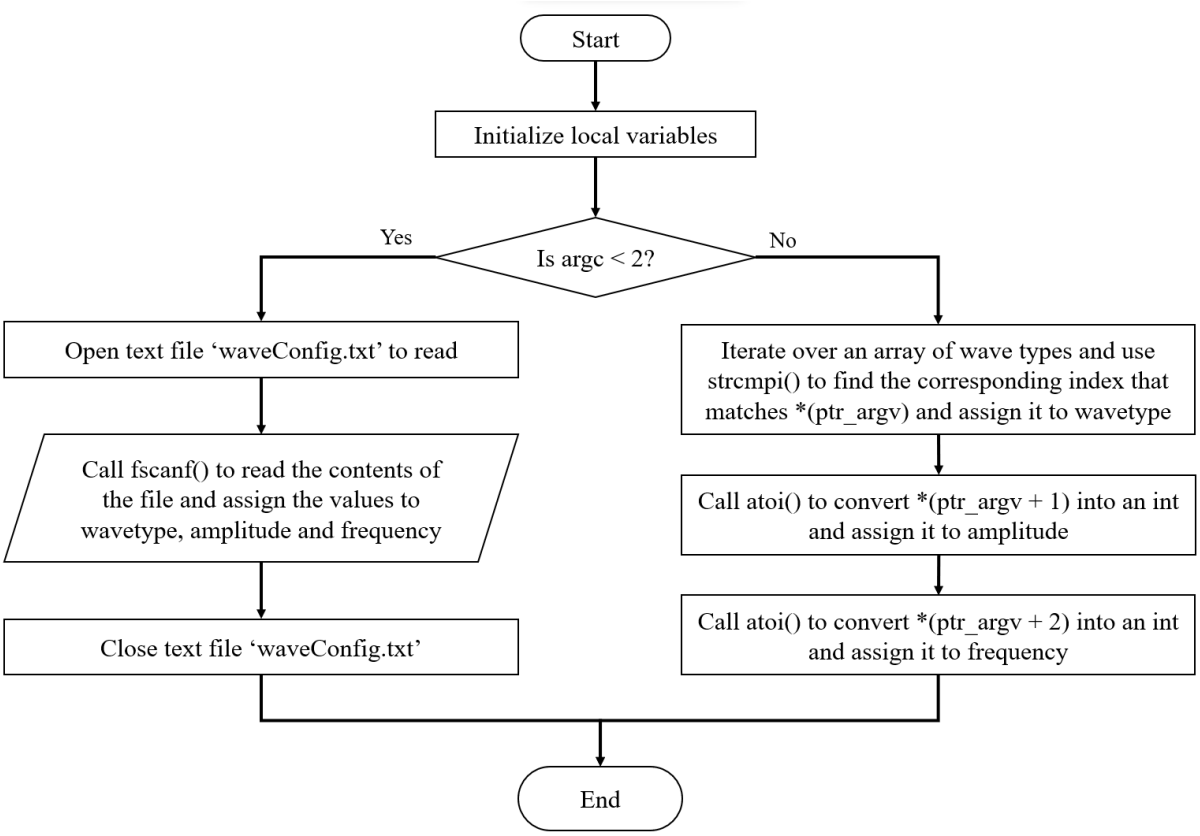
- Program does not prompt users to re-enter arguments on the command line if the arguments are invalid. Due to time constraints we did not manage to test out the codes for invalid inputs, future works can add in this part.

5. Conclusion

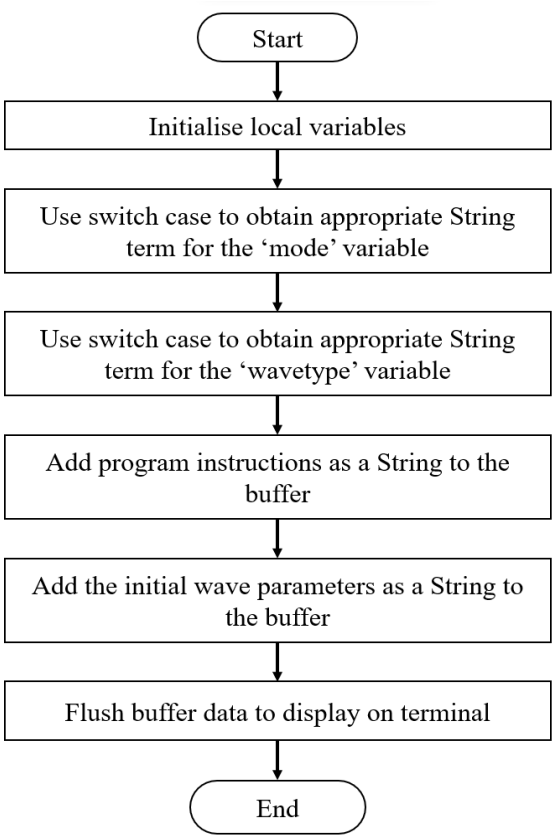
This project has helped us apply whatever we have learnt about C language programming and using real-time systems to create a functioning waveform generator. We have managed to showcase the use of multi-threading, formatted buffer I/O and PCI-DAS hardware input and output. We also have applied extra features, such as keyboard inputs that change input parameters and toggle switch inputs to change between hardware and keyboard control modes as well as to exit the program. Given more time, more features can be added into this program to make it a more efficient and user friendly program.

Appendix A - Software Flowcharts

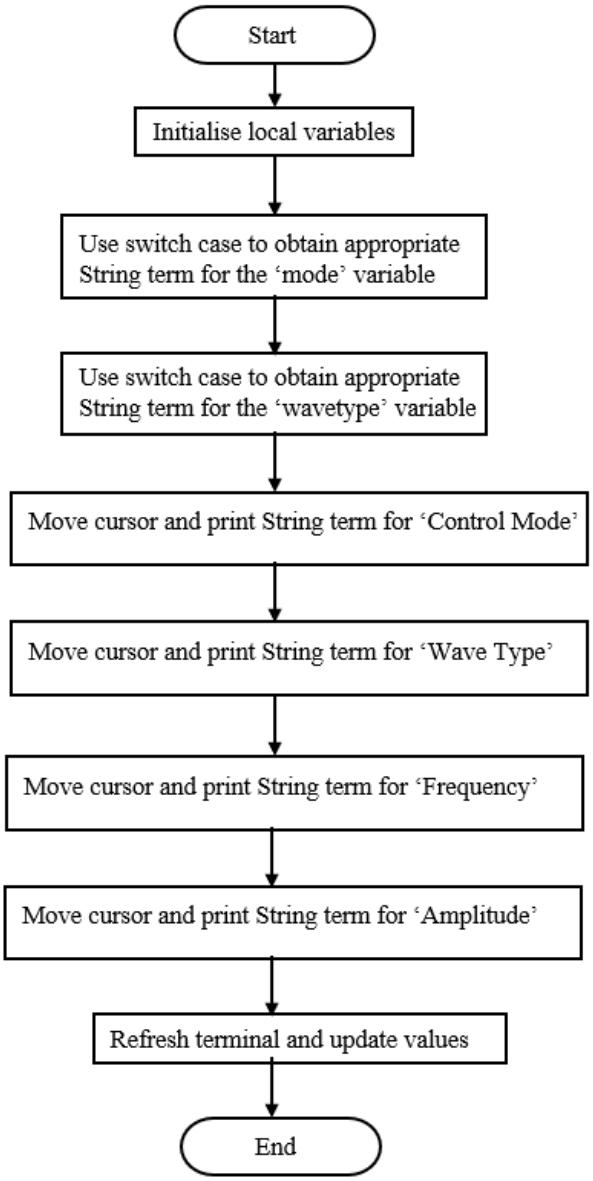
1. *cmdArg()*


Start
Initialize local variables
Is argc < 2?
Yes
No
Open text file 'waveConfig.txt' to read
Call fscanf() to read the contents of the file and assign the values to wavetype, amplitude and frequency
Close text file 'waveConfig.txt'
Iterate over an array of wave types and use strcmpi() to find the corresponding index that matches *(ptr_argv) and assign it to wavetype
Call atoi() to convert *(ptr_argv + 1) into an int and assign it to amplitude
Call atoi() to convert *(ptr_argv + 2) into an int and assign it to frequency
End

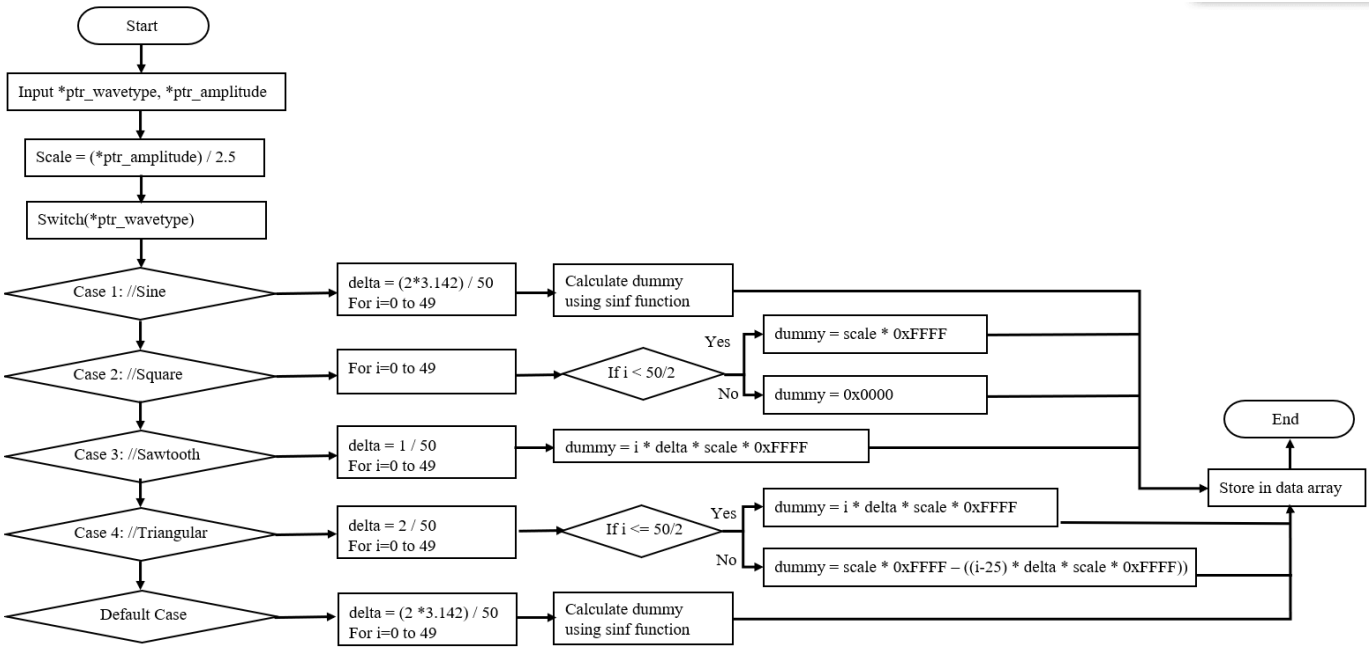
2. *startmsg()*


Start
Initialise local variables
Use switch case to obtain appropriate String term for the ‘mode’ variable
Use switch case to obtain appropriate String term for the ‘wavetype’ variable
Add program instructions as a String to the buffer
Add the initial wave parameters as a String to the buffer
Flush buffer data to display on terminal
End

3. *updateValues()*



4. *waveGeneration()*



Appendix B - Program Listing

```
// standard headers
#include <stdio.h>
#include <stdlib.h>
// #include <stdbool.h>
#include <unistd.h>
#include <time.h>
#include <ncurses.h>
#include <signal.h>


// headers for the pci and DAQ
#include <hw/pci.h>
#include <hw/inout.h>
#include <sys/neutrino.h>
#include <sys/mman.h>
#include <math.h>

// headers for pthreads
#include <pthread.h>
#include <process.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] + a           // Badr3 + a
#define PACERCTL        iobase[3] + b           // Badr3 + b

#define DA_Data         iobase[4] + 0           // Badr4 + 0
#define DA_FIFOCLR      iobase[4] + 2           // Badr4 + 2

int badr[5];                                                    // PCI 2.2
assigns 6 IO base addresses


#define BILLION     1000000000L
#define MILLION     1000000L
#define THOUSAND    1000L


struct timespec start, temp;
double accum =0.0;
double period = 0.01;
int ii = 0;
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

// Declare functions

void cmdArg(int argc, char **argv, int *ptr_wavetype, float *ptr_amplitude, int
*ptr_frequency);
void waveGeneration(int *ptr_wavetype, float *ptr_amplitude);
void signalHandler(int signum);
void *adcIn(void *arg);
void *event_loop(void *arg);
void startmsg();

void updateValues();

// Global Variables

int wavetype, frequency;
float amplitude;
```

```
unsigned int data[100];
uint16_t adc_in;
uintptr_t iobase[6];
uintptr_t dio_in;
int mode = 0;
int old_mode=0;
pthread_t thKey, thdaq;

FILE *fp;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int main(int argc, char **argv) {
    struct pci_dev_info info;
    void *hdl;

    unsigned int i;

    initscr();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);


    signal(SIGINT, signalHandler);

    memset(&info,0,sizeof(info));
    if(pci_attach(0)<0) {
    perror("pci_attach");
    exit(EXIT_FAILURE);
    }
                                                                    /*
Vendor and Device ID */
    info.VendorId=0x1307;
    info.DeviceId=0x01;

    if ((hdl=pci_attach_device(0, PCI_SHARE|PCI_INIT_ALL, 0, &info))==0) {
    perror("pci_attach_device");
    exit(EXIT_FAILURE);
```

```
    }
                                                                                //
Determine assigned BADRn IO addresses for PCI-DAS1602

    //printw("\nDAS 1602 Base addresses:\n\n");
    for(i=0;i<5;i++) {
    badr[i]=PCI_IO_ADDR(info.CpuBaseAddress[i]);
    //printw("Badr[%d] : %x\n", i, badr[i]);
    }

     //printw("\nReconfirm Iobase:\n");                            // map I/O base
address to user space
     for(i=0;i<5;i++) {                                                 // expect
CpuBaseAddress to be the same as iobase for PC
    iobase[i]=mmap_device_io(0x0f,badr[i]);
   // printw("Index %d : Address : %x ", i,badr[i]);
   // printw("IOBASE  : %x \n",iobase[i]);
    }
                                                                                //
Modify thread control privity
    if(ThreadCtl(_NTO_TCTL_IO,0)==-1) {
    perror("Thread Control");
    exit(1);
    }


//************************************************************************
    // ADC Port Functions

//************************************************************************
    // Initialise Board
    out16(INTERRUPT,0x60c0);                // sets interrupts   - Clears
     out16(TRIGGER,0x2081);                      // sets trigger control: 10MHz,
clear, Burst off,SW trig. default:20a0
     out16(AUTOCAL,0x007f);                      // sets automatic calibration :
default

    out16(AD_FIFOCLR,0);                    // clear ADC buffer
```

```
     out16(MUXCHAN,0x0D00);                      // Write to MUX register - SW
trigger, UP, SE, 5v, ch 0-0
                                               // x x 0 0 | 1  0  0 1  | 0x 7   0 |
Diff - 8 channels
                                                 // SW trig |Diff-Uni 5v| scan 0-7|
Single - 16 channels

     out8(DIO_CTLREG,0x90);        // Port A : Input,  Port B : Output,  Port C
(upper | lower) : Output | Output
    if(in8(DIO_PORTA)!=255){
        printw("Please turn all toggle switches on to start the program");
        refresh();
        while(in8(DIO_PORTA)!=255){delay(1);}
    }

    // Reading and assigning cmd line argument
    cmdArg(argc, argv, &wavetype, &amplitude, &frequency);

    // Wave Generation based on cmd line arguments
    waveGeneration(&wavetype, &amplitude);

    startmsg();

    pthread_create(&thdaq,NULL, &adcIn, NULL);

    pthread_create(&thKey, NULL, &event_loop, NULL);


    while(1) {

ii = 0;
        accum = 0;

        if(clock_gettime(CLOCK_REALTIME,&start)==-1)
        {
            perror("clock gettime");
            exit(EXIT_FAILURE);
        }
```

```
        while(ii<50){
            if(accum > period){
                  out16(DA_CTLREG,0x0a23);            // DA Enable, #0, #1, SW 5V
unipolar        2/6
                out16(DA_FIFOCLR, 0);                  // Clear DA FIFO  buffer
                out16(DA_Data,(short) data[ii]);
                  out16(DA_CTLREG,0x0a43);            // DA Enable, #1, #1, SW 5V
unipolar        2/6
                out16(DA_FIFOCLR, 0);                  // Clear DA FIFO  buffer
                out16(DA_Data,(short) data[ii]);
                start = temp;
                ii++;
            }

            if(clock_gettime(CLOCK_REALTIME,&temp)==-1)
{
                perror("clock gettime");
                exit(EXIT_FAILURE);
            }
               accum = (double)(temp.tv_sec-start.tv_sec)+(double)(temp.tv_nsec-
start.tv_nsec)/BILLION;
        }

    }
                                                        // Unreachable code
                                                        // Reset DAC to 5v
    out16(DA_CTLREG,(short)0x0a23);
    out16(DA_FIFOCLR,(short) 0);
    out16(DA_Data, 0x8fff);                    // Mid range - Unipolar

    out16(DA_CTLREG,(short)0x0a43);
    out16(DA_FIFOCLR,(short) 0);
    out16(DA_Data, 0x8fff);

    printw("\n\nExit Demo Program\n");
    pci_detach_device(hdl);
    endwin();

    return(0);
```

```
}

// Function 1: Command line arguments //
void cmdArg(int argc, char **argv, int *ptr_wavetype, float *ptr_amplitude, int
*ptr_frequency)           // *ptr_wavetype -> &wavetype
{
     char arr[4][100] = {"Sine", "Square", "Sawtooth", "Triangular"}; // an array
of different waveforms
      char  **ptr_argv  =  &argv[1];       // start first argument after name of
file.exe
    bool valid_wave = FALSE;
    unsigned int i;
    // Check if the arguments are passed

    if(argc < 2)  // if no argument was passed, read from waveConfig.txt
    {
        FILE *fp = fopen("waveConfig.txt", "r");
        if (fp == NULL) {
        printf("Error opening file\n");
        }

        // Read the contents of the file
        fscanf(fp, "%d %f %d", &wavetype, &amplitude, &frequency);
        printw("Wave configuration was read from the waveConfig.txt");
        // Close the file
        fclose(fp);
    }
    else
    {
        // For Waveform //
        for (i = 0; i < 4; i++)
        {
             if (strcmpi(arr[i], *(ptr_argv)) == 0)     // return 0 if all chars
are = (ignore cases)
            {
            // printw("Plotting %s Wave...\n", arr[i]);
            valid_wave = TRUE;
             *ptr_wavetype = i + 1;       // 1st argument = Type of waveform - 1:
Sine, 2: Square, 3: Sawtooth, 4: Triangular (call by ref)
```

```
            }
        }
        // Amplitude //
        *ptr_amplitude = atoi(*(ptr_argv + 1));      // 2nd argument = Amplitude
        // Frequency //
        *ptr_frequency = atoi(*(ptr_argv + 2));      // 3rd argument = Frequency
    }
}

void waveGeneration(int *ptr_wavetype, float *ptr_amplitude)
{
    double delta, dummy;
    unsigned int i;
    double scale;
    scale = ((*ptr_amplitude)/2.5);  // scale will range from 0 to 1


    switch (*ptr_wavetype)
    {
    case 1: // Sine
        delta=(2.0*3.142)/50.0;                 // increment
        for(i=0;i<50;i++) {
             dummy= ((sinf((float)(i*delta))) +1 )* 0.5 * scale * 0xFFFF  ;    //
FFFF max value - 5 V, 7FFF mid point value - 2.5 V, 0000 lowest 0V
            data[i]= (unsigned) dummy;
        }
        break;

    case 2: // Square
        for (i=0; i<50; i++){
            if (i < (50)/2)
            {
                dummy = scale * 0xFFFF;
                data[i] =  (unsigned) dummy;
            }
            else if (i >= 50/2)
            {
                dummy = 0x0000;
                data[i] =  (unsigned) dummy;
```

```
            }
        }
        break;

    case 3: // Sawtooth
        delta = 1.0/(50);
        for (i=0; i<(50); i++){
            if (i <= (50)/2)
            {
                dummy = ( i * delta) * scale  * 0xFFFF;
            }
            if (i > (50)/2 && i <(50))
            {
                dummy = ( i * delta)  * scale * 0xFFFF;
            }
            data[i] = (unsigned)(dummy);
        }
        break;

    case 4: // Triangular
        delta = 2.0/(50);
        for (i=0; i<50; i++)
        {
            if(i <= 50/2)
                {
                dummy = i * delta * scale * 0xFFFF;
            }
            if (i > 50/2)
            {
                dummy = (scale * 0xFFFF) - ( (i-25)  * delta * scale *  0xFFFF);
            }
            data[i] = (unsigned)(dummy);
        }
    break;

    default:
        delta=(2.0*3.142)/50.0;                 // increment
        for(i=0;i<50;i++) {
```

```
            dummy= ((sinf((float)(i*delta))) +1 )* 0.5 *scale * 0xFFFF  ;    //
FFFF max value - 5 V, 7FFF mid point value - 2.5 V, 0000 lowest 0V
            data[i]= (unsigned) dummy;
        }
        break;
    }
}



// ADC input Thread
void *adcIn(void *arg)
{

    unsigned int count;
    unsigned short chan;
    float amp;
    int freq, wt;

    while(1) {

        count=0x00;

        while (count < 0x02) {
            chan= ((count & 0x0f)<<4) | (0x0f & count);
            out16(MUXCHAN,0x0D00|chan);     // Set channel   - burst mode off.
              delay(1);                                          // allow mux to
settle
            out16(AD_DATA,0);                           // start ADC
            while(!(in16(MUXCHAN) & 0x4000));
            adc_in = in16(AD_DATA);
            // printw("ADC Chan: %02x Data [%3d]: %4x \n", chan, (int)count,
(unsigned int)adc_in);     // print ADC


            // DIO input for mode switching //


            dio_in=in8(DIO_PORTA);                  // Read Port A
```

```
            //printw("Port A : %02x\n", dio_in);

            if(dio_in - 240 <= 7)
            {
                raise(SIGINT);
            }
            else if(dio_in -248 <= 3)
            {
                mode = 1;                       // Hardware Mode
                if(old_mode!=mode){
                    updateValues();
                    old_mode = mode;
                }
                wt = dio_in - 248 +1;
                if (wavetype != wt){
                    pthread_mutex_lock( &mutex );
                    wavetype = wt;
                    updateValues();
                    waveGeneration(&wavetype, &amplitude);
                    pthread_mutex_unlock( &mutex );
                }
            }
            else
            {
                mode = 0;                       // Keyboard Mode
                if(old_mode!=mode){
                    updateValues();
                    old_mode = mode;
                }
            }


            delay(10);

            if(mode){
                                        // Write to MUX register - SW trigger, UP,
DE, 5v, ch 0-7

            if (count == 0x00) {
```

```
            if (adc_in < 0x327c){
                amp = 0.5;
            }
            else if(adc_in >= 0x327c && adc_in < 0x6537){
                amp = 1.0;
            }
            else if(adc_in >= 0x6538 && adc_in < 0x9783){
                amp = 1.5;
            }
            else if(adc_in >= 0x9784 && adc_in < 0xcb6f){
                amp = 2.0;
            }
            else if(adc_in >= 0xcb70 && adc_in < 0xffff){
                amp = 2.5;
            }

            if(amplitude!=amp){
                pthread_mutex_lock( &mutex );
                amplitude = amp;
                updateValues();
                waveGeneration(&wavetype, &amplitude);
                pthread_mutex_unlock( &mutex );
            }
            count = 0x01;
        }

        else {
            freq = (int) (adc_in/66.19 + 10);

            if(frequency!=freq){
                pthread_mutex_lock( &mutex );
                frequency = freq;
                updateValues();
                period = (float) 1.0/frequency;
                pthread_mutex_unlock( &mutex );
            }
            count = 0x00;
        }
        }
```

```
        }
    }

}


// Keypress tracking //
void *event_loop(void *arg){
    int ch;
    while ((ch = getch()) != KEY_F(1)){
        if(!mode){
            pthread_mutex_lock( &mutex );
            if (ch == KEY_RIGHT && frequency < 901){
                frequency+= 100;
                period = (float) 1.0/frequency;
                updateValues();
            }
            else if (ch == KEY_LEFT && frequency > 199){
                frequency -= 100;
                period = (float) 1.0/frequency;
                updateValues();
            }
            else if (ch == KEY_DOWN && amplitude > 0.5){
                amplitude -= 0.5;
                waveGeneration(&wavetype, &amplitude);
                updateValues();
            }
            else if (ch == KEY_UP && amplitude < 2.5){
                amplitude += 0.5;
                waveGeneration(&wavetype, &amplitude);
                updateValues();
            }
            else if (ch == 32){
                if(wavetype!=4){
                    wavetype++;
                }
                else{
                    wavetype = 1;
                }
```

```
                waveGeneration(&wavetype, &amplitude);
                updateValues();
            }
         pthread_mutex_unlock( &mutex );
        }
    }
    //pthread_EXIT(NULL);
}

// Exit Signal Handler //
void signalHandler(int signum)
{
    endwin();
     printf("\nYou have either pressed Control C or toggle switch 1 off to exit
the program\n");

            // Procedure for writing to file upon exit //
    fp = fopen("waveConfig.txt", "w");

    // Check if the file was opened successfully
    if (fp == NULL) {
        printf("Error opening file\n");
    }
    fprintf(fp, "%d %f %d\n", wavetype, amplitude, frequency);
    fclose(fp); // Close the file
    pthread_mutex_destroy( &mutex );
    printf("Lastest wave configuration has been saved to waveConfig.txt\n");
    printf("Exit Demo Program\n");
    exit(1);
}

// UI
void startmsg(){

    char *wavetype_str, *mode_str;

    switch(mode) {
    case 1:
        mode_str = "DAQ     ";
```

```
        break;
    case 0:
        mode_str = "Keyboard";
        break;
    }

    switch(wavetype) {
    case 1:
        wavetype_str = "Sine      ";
        break;
    case 2:
        wavetype_str = "Square    ";
        break;
    case 3:
        wavetype_str = "Sawtooth  ";
        break;
    case 4:
        wavetype_str = "Triangular";
        break;
    }


  box(stdscr, 0, 0);
  move(0,0);
  addstr("Welcome to our wave generator!\n\n");
  refresh();

  // Instructions for Hardware
  attron(A_UNDERLINE);
  addstr("\tDAQ CONTROL INSTRUCTIONS\n");
  attroff(A_UNDERLINE);
  addstr("\tTo enable hardware control, please on toggle switch 2\n");
  addstr("\tUse potentiometer 1 to control amplitude\n");
  addstr("\tUse potentiometer 2 to control frequency\n");
    addstr("\tUse toggle switches 3 and 4 to control the wave type:\n\t00 -
Sine\n\t01 - Square\n\t10 - Sawtooth\n\t11 - Triangular\n\n");

  //Instructions for keyboard controls
  attron(A_UNDERLINE);
```

```
  addstr("\tKEYBOARD CONTROL INSTRUCTIONS\n");
  attroff(A_UNDERLINE);
  addstr("\tTo enable keyboard control, please off toggle switch 2\n");
  addstr("\tUse up and down arrow keys to control amplitude\n");
  addstr("\tUse left and right arrow keys to control frequency\n");
  addstr("\tUse spacebar to change the wavetype\n\n");

  attron(A_UNDERLINE);
  addstr("\tCurrent Parameters\n");
  attroff(A_UNDERLINE);
  refresh();
    printw("\tControl  Mode:  %s\t\tWave  Type:  %s\n",  mode_str,  wavetype_str);
//hardware or keyboard,    use move() to dynamically edit the values when they
change respectively
  printw("\tFrequency: %4d\t\t\tAmplitude: %.2f\n", frequency, amplitude);
  curs_set(0);
  refresh();
}

void updateValues(){
    char *wavetype_str, *mode_str;

    switch(mode) {
    case 1:
        mode_str = "DAQ     ";
        break;
    case 0:
        mode_str = "Keyboard";
        break;
    }

    switch(wavetype) {
    case 1:
        wavetype_str = "Sine      ";
        break;
    case 2:
        wavetype_str = "Square    ";
        break;
    case 3:
```

```
        wavetype_str = "Sawtooth  ";
        break;
    case 4:
        wavetype_str = "Triangular";
        break;
    }

    move(19,22);  //move cursor to control mode value
    printw("%s", mode_str);
    move(19, 51); //move cursor to wave type value
    printw("%s", wavetype_str);
    move(20,19);  //move cursor to frequency value
    printw("%4d", frequency);
    move(20,51);  //move cursor to amplitude value
    printw("%.2f", amplitude);

    //change the values
    refresh();
}
```

Appendix C - Group Photo

From left to right:

(Lim Jin Hng, Timothy), (Greg Angelo Gonzales Nonato), (Minn Set Moe Hein), (Mohamed Raizee Bin Mohamed Ibrahim), (Lin YiJuan), (Mohammad Uzair Bin Rosman'id)