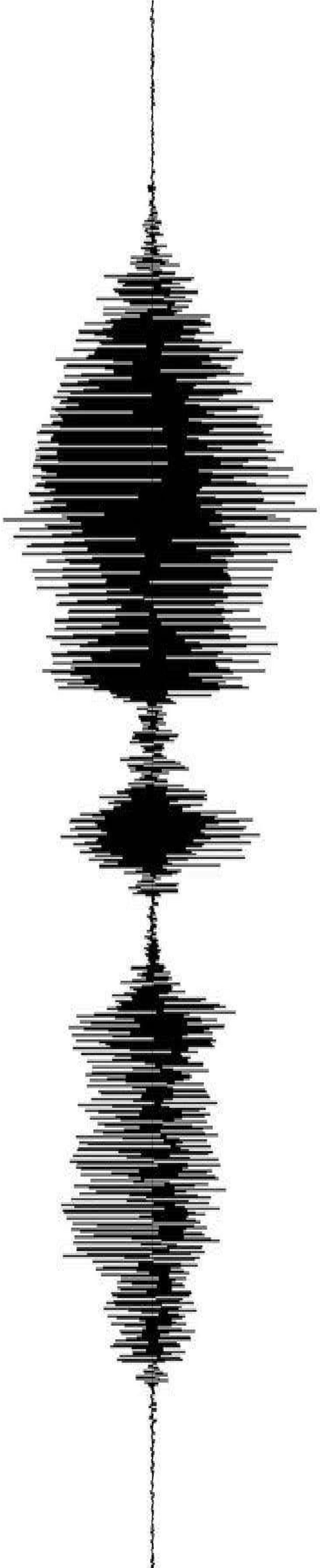


Sant'Anna
Scuola Universitaria Superiore Pisa

MiniMoog Additive Synthesis

Adbelrahman S. A

M.Sc. Embedded Computing Systems
Real-Time System course
email: m.rahm7n@gmail.com
Matricola: 590759

Supervisor:
prof. Giorgio Buttazzo.

Scuola Superiore Sant'Anna
Istituto TeCIP - RETIS Lab,
Web: http://retis.sssup.it/ giorgio/

Hint
This system is developed under Linux Ubuntu 16.04 using Low Latency Kernel 4.8.0-36, Allegro5 for GUI and pthread library for real time periodic task managements

Contents

1 System Architecture Overview

In this Document, we present an emulation of an additive synthesizer (the classical Mini Moog). In this system we relay on the **pthread** library for multi-threading programming. The system is responsible for the generation of the following wave-forms (sine, triangle and square) each with a specific frequency notes decided by users and then they are added/integrated together after they are passing through a dedicated Band Pass Filter (BPF).

Below in Figure 1, is the task models description. In the model multiple threads are running concurrently

1. TASK Graphics: allocated for rendering the graphics.
2. TASK Keyboard: Actively poll the keyboard to trigger and control the frequency notes of each wave
3. TASK Wave-forms: each waveform has an independent periodic tasks.
4. TASK Audio handles the Integration of the wave-forms and passing the generated buffer through the audio card driver.

The user can decide which frequency notes trigger for each waveform but Its limited to [Notes A4, B4, C4, D4, E4, F4, G4]. Also it's possible to trigger each wave to either ON or OFF and the same for filter.

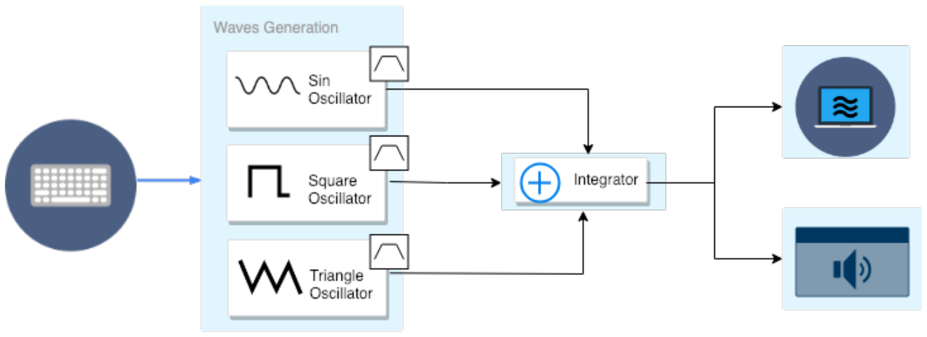
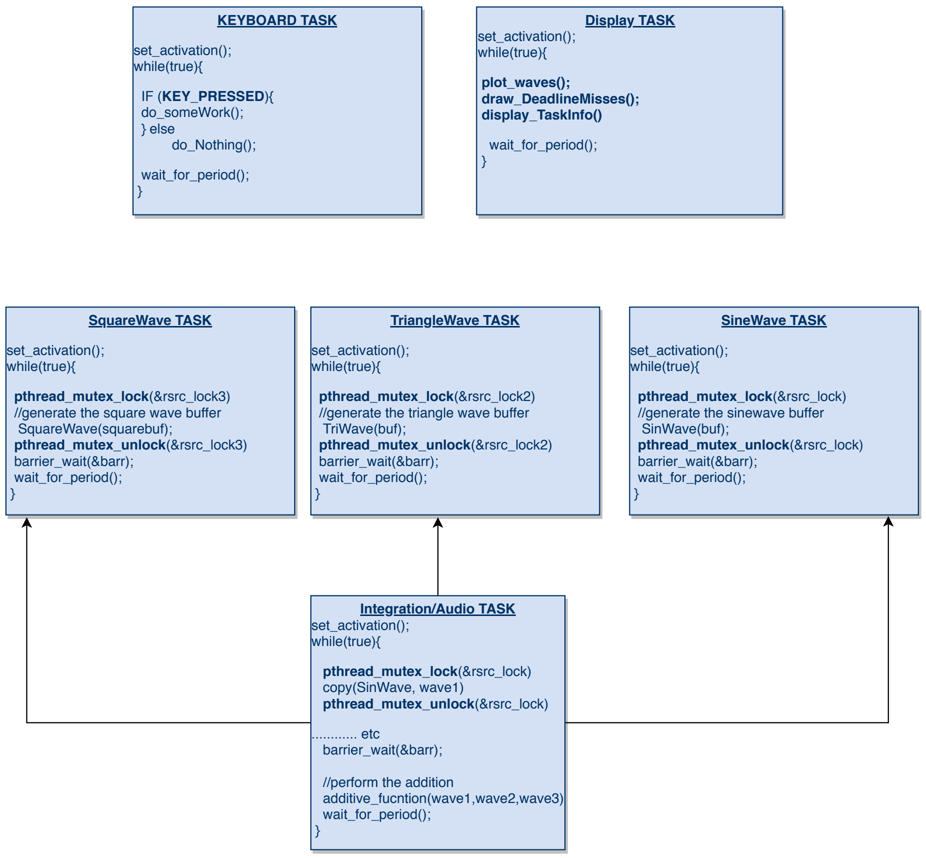


Figure 1: Additive Synths Task Model

2 Periodic Tasks Description

As illustrated below, 5 periodic tasks are running concurrently using the scheduling policy Round Robin SCHED_RR. The main functions which are designed to manage periodic tasks are developed over pthread library **ptask** and **timemng** as reported in [3]. Tasks are configurable through the file **task_configs**



```
KEYBOARD TASK
set_activation();
while(true){
 IF (KEY_PRESSED){
 do_someWork();
 } else
        do_Nothing();

 wait_for_period();
}
```

```
Display TASK
set_activation();
while(true){
 plot_waves();
 draw_DeadlineMisses();
 display_TaskInfo()

  wait_for_period();
}
```

```
SquareWave TASK
set_activation();
while(true){
 pthread_mutex_lock(&rsrc_lock3)
 //generate the square wave buffer
  SquareWave(squarebuf);
 pthread_mutex_unlock(&rsrc_lock3)
 barrier_wait(&barr);
 wait_for_period();
}
```

```
TriangleWave TASK
set_activation();
while(true){
 pthread_mutex_lock(&rsrc_lock2)
 //generate the triangle wave buffer
  TriWave(buf);
 pthread_mutex_unlock(&rsrc_lock2)
 barrier_wait(&barr);
 wait_for_period();
}
```

```
SineWave TASK
set_activation();
while(true){
 pthread_mutex_lock(&rsrc_lock)
 //generate the sinewave buffer
  SinWave(buf);
 pthread_mutex_unlock(&rsrc_lock)
 barrier_wait(&barr);
 wait_for_period();
}
```

```
Integration/Audio TASK
set_activation();
while(true){
 pthread_mutex_lock(&rsrc_lock)
 copy(SinWave, wave1)
 pthread_mutex_unlock(&rsrc_lock)

........... etc
 barrier_wait(&barr);

 //perform the addition
 additive_fucntion(wave1,wave2,wave3)
 wait_for_period();
}
```

Figure 2: Periodic Tasks Model Description

The Inter-Task Communications between the Integration/Audio TASK which act as a reader and the Waveforms TASKS which are acting as writers are protected from simultaneous accesses to shared resources. This achieved through the mutex mechanism, which uses a semaphore variable of type pthread_mutex_t mutex semaphore to execute the critical sections of code in mutual exclusion. If our interest to have a very clear sound its necessary to provide a syncronization mechanism beside the mutex semaphore, otherwise, we can get a clear wave but sound will be a little bit distorted after each period, this is achieved by using barrier synchronization by specifying the number of threads that are synchronizing on the barrier (4 in our case, 3 writers, one reader), and set up threads to perform tasks and wait at the barrier until all the threads reach the barrier. When the last thread arrives at the barrier, all the threads resume execution. However, a trade-off of using the barrier mechanism is the observed delay which in turns leads to deadline misses. Therefore, a BARRIER_ENABLE flag could be set 1 or 0 through task_configs file to allow or deny synchronization.

Reported below the tested configuration for all the threads in the system. And In the next sections each Task will be discussed in details

Table 1: Tasks Configurations

TASKS	ID	PERIOD	DEADLINE	PRIORITY
TASK sin_src	0	20	20	10
TASK square_src	1	20	20	10
TASK triangle_src	2	20	20	10
TASK audio_thread	3	20	20	20
TASK allegro_draw	4	50	50	30
TASK keyboard_thread	5	10	10	40

3 Digital Signal Processing Module

The waveform generation of the system was generated based on the standard *sin* function which is declared in math.h library. The produced samples of the *sin* function are 32-bit floating point so no the need to scale the produced output as the produced sampled goes in range between -1 and +1. Therefore all generated wave-forms streams that outputs sine, square, and triangle waves will be derived through to the sound card over the audio driver which could be configurable to be either directsound, alsa, openal, oss or pulseaudio.
All audio and waves declaration could be found in the signal_src.c and signal_src.h module there are 3 functions to generate the desired wave but all of them actually depends on the standard sin function.

1. **sin_osc**: generate a simple sine wave generated in the sinWave_buf which is self explanatory in the code.
2. **square_osc**: generating any wave could be done by adding different harmonic of sin wave. but in this case for simplicity of design, a simple threes-hold was defined as a reference for the sin wave, if the threes-hold more than 0 we generate a positive step (an amplitude of 1), and if its ¡ 0 we generate a -ve step. In this case there is no transition delay in falling or rising edges which is not practical but fits in our design which leads to an ideal square wave, samples were stored in squareWave_buf.
3. **triangle_osc**: triangle wave is generated using a similar approach from the sin wave using the fmod function, so we incrementally build the triangle wave using a linear step, samples were stored in triangleWave_buf.

Below **as shown in Figure 3**, is the schematic showing the process of the waveform generation, then integrating them into a single buffer and after that they derived using alsa driver to be played through the audio card.

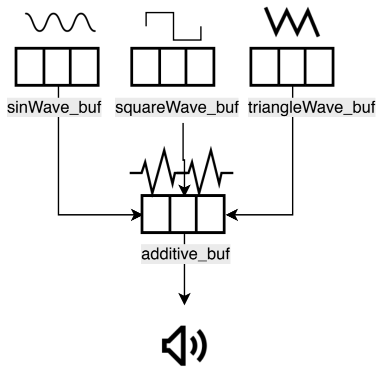


Figure 3: Audio driven flow and process

In order to make a precise sound output we have to set the output of sampling time of each wave to be equivalent to the audio sampling rate. In this case, where the audio sampling frequency could be configured to be 32 KHz, 44.1 KHz or 48 KHz therefore, the dt: **sampling time** should be"

$$dt = \frac{1}{FS\_Audio}$$

Regarding the audio driver, we could set it using the allegro configuration file allegro5.cfg or it's always better to set it throught the api in the application, which is defined in the minimoog.c init function
al_set_config_value(al_get_system_config(),"audio","driver","alsa");

The most important function is:
al_create_audio_stream(fragment_count, samples, freq, Audio_depth, chan_conf)

where we encounter a big delay in the system which produce and incremental increasing with respect to deadline misses but produces an overall very good sound quality following the default configuration.

in this table below is a two trade-off with respect to quality of sound vs deadline misses. Hence, the less the sampling frequency, the lower audio quality.

Table 2: Tasks Configurations

KNOBS		**METERS**	
Samples	Frequency	DLM	SoundQuality
1024	44.1KHz	1	percise
512	44.1KHz	0	poor
256	44.1KHz	0	poor
1024	32.0KHz	1	less percise
512	32.0KHz	0	less percise
256	32.0KHz	0	less percise

The choice of fragment_count, [1] samples and frequency (FS_Audio) directly influences the audio delay. The delay in seconds can be expressed as:

$$delay = \frac{fragment\_count * samples}{freq}$$

sample_size = al_get_channel_count(chan_conf) * al_get_audio_depth_size(depth)
samples = bytes_per_fragment / sample_size

Finally, a Finite Impulse Response (FIR), Band-pass filter is designed to attenuate the signals if its desired [4], hence, Filter function is called one time only for each wave generator to generate the filter coefficient before the activation of each periodic task. **As shown in Figure 4,**

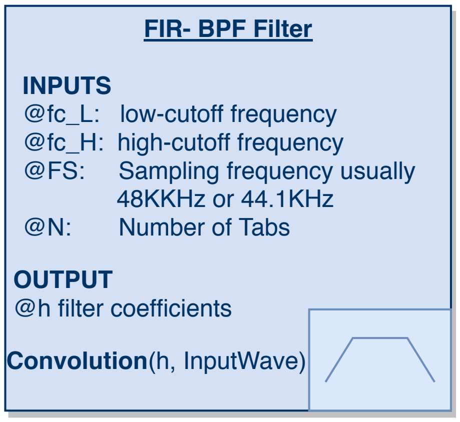

FIR- BPF Filter
INPUTS
@fc_L: low-cutoff frequency
@fc_H: high-cutoff frequency
@FS: Sampling frequency usually 48KKHz or 44.1KHz
@N: Number of Tabs
OUTPUT
@h filter coefficients
Convolution(h, InputWave)

Figure 4: FIR, BPF

$$h = \frac{(sin(fc\_H * icount * PI) - sin(fc\_L * icount * PI))/(icount * PI);\_count * samples}{freq}$$

$$\forall icount = 1 : nCoeef$$

After computing the coefficient the signal is convoluted with the waveform buffer and then each waveform after passing throught the BPF are added by a periodic task **Task Audio** which integrate the waveforms buffers afer convolution and play the audio stream. Hint the default setting is to play the pure sound waves without the filter, where filter activation is triggered by pressing SPACE key.

4 Keyboard Interpreter Module

The keyboard module is a periodic tasks does nothing except checking for pressed keys events. The user can control each waveform frequency note from A4 to G4 by pressing a corresponding key. Most of configuration are defined in the table and figure below. Hence, each keyboard row control a certain wave frequency in order A4, B4,... G4 in order from left to right.

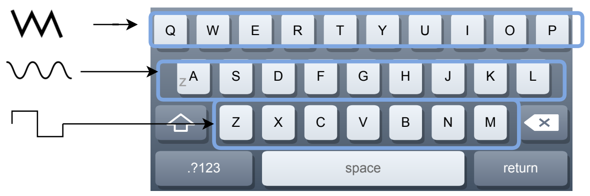
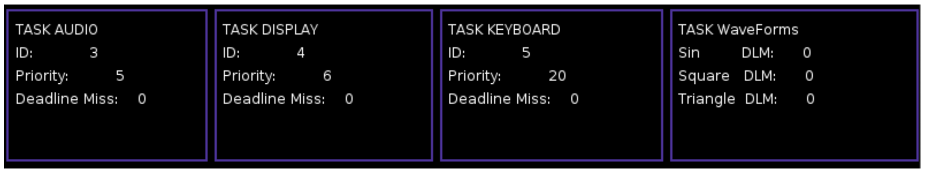


Figure 5: keyboards Notes Map

Table 3: Keyboards Key Configurations

KEYS	FUNCTIONS
1	activate sinWave
2	activate squareWave
3	activate triangleWave
4	deactivate sinWave
5	deactivate triangleWave
6	deactivate triangle wave
-	Volume Up
+	Volume Down
A	sin Note A4, Freq 440.00 Hz
S	sin Note B4, Freq 493.88 Hz
D	sin Note C4, Freq 261.63 Hz
F	sin Note D4, Freq 293.66 Hz
G	sin Note E4, Freq 329.63 Hz
H	sin Note F4, Freq 349.23 Hz
J	sin Note G4, Freq 392.00 Hz
SPACE	FILTER ON/OFF

5 User Interface Module



```
TASK AUDIO            TASK DISPLAY          TASK KEYBOARD         TASK WaveForms
ID:          3        ID:          4        ID:          5        Sin       DLM:    0
Priority:         5   Priority:         6   Priority:        20   Square    DLM:    0
Deadline Miss:    0   Deadline Miss:    0   Deadline Miss:    0   Triangle  DLM:    0
```

Figure 6: Concurrent Task Information Field

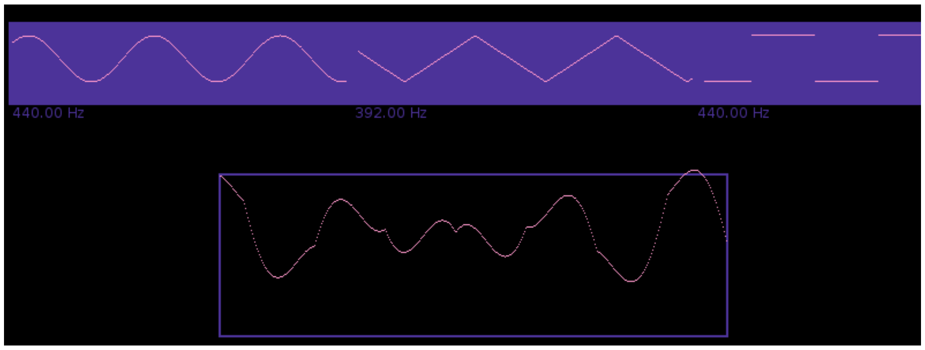
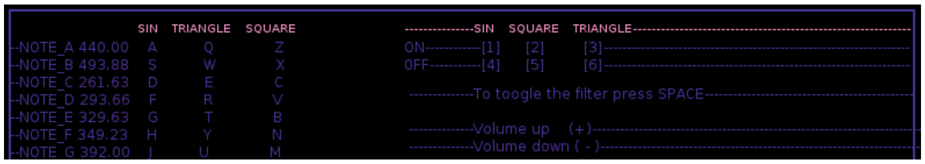


Figure 7: Oscilloscope Monitor



```
                     SIN   TRIANGLE   SQUARE         --------------SIN    SQUARE   TRIANGLE--------------------------
--NOTE_A 440.00      A        Q          Z           ON-----------[1]     [2]        [3]-----------------------------
--NOTE_B 493.88      S        W          X           OFF----------[4]     [5]        [6]-----------------------------
--NOTE_C 261.63      D        E          C
--NOTE_D 293.66      F        R          V            ------------To toogle the filter press SPACE-------------------
--NOTE_E 329.63      G        T          B
--NOTE_F 349.23      H        Y          N            ------------Volume up    (+)-----------------------------------
--NOTE_G 392.00      J        U          M            ------------Volume down ( - )---------------------------------
```

Figure 8: Instruction Field

The Display is divided into 3 fields, Task releated information, to monitor deadline misses. Oscilloscope monitor filed which is responsible for plotting/rendering signals samples in real-time, hence each sample is represented by a pixel [2].

$$\forall i = 1 : Samples\_Per\_Buffer$$

$$al\_draw\_pixel(x, y, i)$$

where x and y are the screen cooridinates

References

[1] Allegro5 manual https://www.allegro.cc/manual/5/al_create_audio_stream. *Allegro 5.0 reference manual - Allegro 5 Manual.*

[2] Allegro5 manual https://www.allegro.cc/manual/5/al_draw_pixel. *Allegro 5.0 reference manual - Allegro 5 Manual.*

[3] G. Buttazzo and G. Lipari. Ptask: An educational c library for programming real-time systems on linux. *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, 2013.

[4] D. Morgan. *Practical DSP modeling, techniques, and programming in C.* Wiley, 1995.