Master's Degree in Embedded Computing Systems

SUPER: A SUrface PlayER

Project developed for the course of

Real-Time Systems

February 14, 2019

Gabriele Ara
Student ID: 501239
Email: g.ara@studenti.unipi.it

Contents

1. Introduction

This document is a report for the project required for the course of Real-Time Systems for Master's Degree in Embedded Computing Systems, developed by Gabriele Ara.

The original assignment is the following:

> "Attach a microphone to a table and process the signal in order to generate sound as a function of the produced signal. The sound can be generated via MIDI notes or by manipulating a wave file."

The developed project also will follow common requirements for all projects developed for the course; this means that it shall be developed in C language under the Linux operating system, adopting `pthread` library for real-time task management and the Allegro library to implement a simple graphical user interface.

Starting from these mandatory requirements, the project aims to develop a standalone application that will be able to listen to a microphone input and recognize some user-defined sounds pre-recorded during a configuration phase and upon recognition reproduce the corresponding wave files with the smaller delay possible. To do so, the developed application provides a command-line interface that can be used to configure all the parameters for audio acquisition

Albeit simple, this project aims to provide musicians a reliable instrument that could be used to way to enrich their compositions with the execution of pre-recorded samples in real-time along the musician performance with a regular instrument or an improvised one, like for example a simple table.

This document will first describe more precisely actual software requirements and functionalities, followed by an in-depth description of the implementation of the program; implementation details illustrated will comprehend an analysis of the global data structures of the program, the list of real-time tasks used, their attributes and a description of the possible compilation attributes that may change the behavior of the program. Finally, results of a few user-case scenarios will be illustrated, along with potential improvements that may be done to the provided functionalities.

2. Requirements

This chapter illustrates more in depth the requirements of the project as a whole, followed by the requirements of each of the three modules in which the program has been partitioned.

2.1 Global Requirements

As stated in the introduction, the developed program shall be able to listen to a microphone, preferably a contact microphone (also called piezo), so that it may be attached to any surface that needs to be played. If an external microphone is not provided, any internal microphone could suffice, in order to listen for audio input.

Given the audio acquired by the microphone, the program shall be able to recognize a set of specified sounds that have been previously recorded and associated each with an with audio sample.

Whenever one of the pre-recorded sounds is detected, the corresponding audio sample is played. Optionally, the user shall be able to configure the inter-arrival time between two consecutive executions of the same audio files, so that sustained sounds like for example musical notes won't trigger multiple executions of the same audio file.

The delay between the acquisition of the audio from the microphone interface and the triggering of one of the audio samples should be minimal, because even a delay of a few tens of a second may negatively affect the harmoniousness of the musical performance.

Sounds that may be recognized by the program must be defined by the user during an initial configuration phase, in which the microphone is not used unless the input audio sample itself is being recorded. After this phase, the program will continuously listen to microphone input to check whether a specified sample has been detected. Later in this document, these two phases will be referred as the configuration phase and the active phase respectively.

While in configuration phase, the program will implement a small set of command-line instructions that can be used to change program parameters and trigger specific actions. When the program moves from the configuration to the active phase, a new window is opened, containing a graphical user interface that allows users to both monitor the microphone input and tweak some parameters in real-time (such as volume of the files opened for playback, their base frequency, left-right panning, etc.).

2.2 Main Module Requirements

This module has two completely different purposes between configuration and active phases: during configuration phase, this module is responsible for handling user input via a command-line interface; when the user requests to switch to the active phase this module starts the concurrent tasks in the other modules and enters sleep, waiting for one of the concurrent tasks to request the termination of the active phase.

When that event arises, this module will wake up and signal all the concurrent tasks that the termination of the active phase has been requested, waiting then for each of the tasks to terminate its loop gracefully. Then the program will revert to configuration phase.

2.3 Audio Module Requirements

This module interacts with the sound card of the host computer, both for recording and playing sounds. It holds all the information of audio files opened via command line during configuration phase, as well as all the information about recorded audio samples that can be necessary to later detect them during the active phase.

Whenever the program needs to listen to the microphone input or playback a certain audio sample, the action will be requested to this module.

2.3.1 Opening Audio Files

During configuration phase, the user can open audio files that will be used as sound bank during the active phase. Each of these files can then be associated with one audio sample that may be used as trigger to start their execution during active phase. The association is 1 to 1 and if an audio file should be triggered by multiple kinds of signals it can be opened multiple times and associated each time with a different sample.

Files can be opened/closed during configuration phase only, while other parameters such as volume, panning etc. may be modified during active phase only.

2.3.2 Listening to the Microphone

While in configuration phase, requesting a set of data of a specified length from the microphone will block the requesting thread execution until all data have been successfully recorded. In contrast, during active phase requesting data from a microphone is a non blocking operation and if no new data is available then the most recent sample will be provided, along with a timestamp.

In either case, the sound card should sample external input at least with a frequency from 40 kHz to 60 kHz; this because the human ear can detect sounds with a frequency from 20 Hz to 20.000 kHz range. Using Nyquist-Shannon sampling theorem, the sampling rate should be at least double the maximum frequency, thus a commonly adopted sampling rate for high quality audio acquisition is 44.100 kHz.

Audio will be sampled in mono mode, because for our purposes we have no need to treat differently left and right channel. The received samples from the sound card will be expressed in Pulse Code Modulation (PCM), one value for each sample. Given the default acquisition rate of 44.100 kHz, the sound card throughput is about 44100 samples per second.

A real-time task shall be executed to acquire values produced by the sound card and store them within buffers that may be accessed by other tasks or modules. To do so, the task shall execute at a rate that ensures both reactivity to the system and that no samples can be lost because of buffer overflows.

2.3.3 Reproducing Sounds on Triggers

Once data acquired from the microphone is published at a given rate, other tasks shall analyze it in order to check whether one of the trigger samples have been recorded, which means that a certain audio sample shall be executed. Since multiple pre-recorded samples may be configured, each of them will spawn a different task during active phase for audio recognition.

This operation will require a comparison between some stats that characterize a certain pre-recorded audio sample and the most recent sample acquired from the microphone. For a more in depth description of how this will be performed, refer to section 3.

The user shall be able to specify whether every time a certain sound is matched against a specific recording the corresponding audio file should be played or a minimum interval should elapse before the very same audio file can be reproduced again. This way, inputs

Figure 2.1: Screenshot of the graphical user interface during active phase.

that may last for an extended time may trigger only a single audio execution, avoiding unintended overlapped executions.

2.4 Video Module Requirements

This module is responsible for providing the user a graphical interface that will show real-time information about the system and to handle user input, either with the keyboard or the mouse.

To do so, two tasks will be defined, one to display the GUI on the screen and another to handle inputs.

2.4.1 Elements of the Graphical User Interface

The GUI will provide both real-time information about the audio signal received from the microphone and provide access to attributes associated with opened audio files. One task will run periodically during active phase to refresh the content of the screen. During configuration phase this module does nothing.

The graphic interface will be divided in three sections:

1. The spectrum section will display the output of the Fast Fourier Transform of the most recent recorded signal (see section 3), which for simplicity will be converted from complex values to magnitudes only.

2. The time section will display the recent history (up to a few seconds) of the strength of the recorded audio over time.

3. The right panel will show the attributes of each opened audio file, such as volume, left-right panning and base frequency and it will allow the user to tweak their values.

A screenshot of the GUI provided by the program is shown in Figure 2.1.

2.4.2 Interaction with the User

During the active phase, the video module is responsible for reacting to user inputs either via the keyboard or the mouse.

Keyboard interaction is used to start execution of audio samples as if the keyboard was a sort of piano, each number associated with the respective file by index, staring from 1. The 'Q' key is also used to quit the active phase and close the window, returning to configuration phase in text-only mode.

The mouse can be used to interact with the buttons placed on the right panel (the orange ones); such buttons can:

- reproduce a specific sample;
- increase/decrease volume of a specific sample, from 0 to 255;
- increase/decrease left-right panning of a specific sample, from 0 to 255, with 0 being left only, 255 right only and 128 mid-panning.
- increase/decrease base frequency adjustment of a specific sample, from 0 to 999.

The latter option allows the user to speed-up or slow down sample execution rate; the normal speed value is 100 and changing it affects the pitch of the sample when played linearly (which means that a value of 200 doubles the pitch, while 50 playbacks it with half speed).

3. Sound Analysis and Spectrum Recognition

The core requirement of the project is the ability to recognize sounds recorded in the configuration phase in real-time, once the active phase has began.

To do so, the program will calculate the *normalized cross correlation* of two audio signals of same length, which is an index between -1 and 1 that indicates the similarity between them. In particular, such index can be used to tell whether there exists a transformation that can bring one of the two signals to be the most similar possible to the other in the time domain by applying only translation and scaling operations. The higher its absolute value, the most similar the two signals can become by applying said transformations.

3.1 Non Normalized Cross Correlation

The normalized cross correlation between two signals is easily obtained from the non normalized cross correlation (or simply the cross correlation) between them, so we will start by describing how to compute the latter one. To calculate it in the time domain for discrete signals[1], the following steps are necessary:

1. Time-reverse the second of the two signals that must be compared.

[1] Our audio samples are expressed in Pulse-code Modulation, which is a discrete time representation.

2. Compute the convolution between the time-reversed version of the first signal and the second one; a third signal is obtained.

3. Take the maximum value over time of the resulting signal.

As a formula, this can be expressed as[2]:

$$(f \star g)[n] = \sum_{m=-\infty}^{\infty} f[m]\, g[-(n-m)] \tag{3.1}$$

$$R_{f,g} = \max_{n} \quad (f \star g)[n] \tag{3.2}$$

There is a more simple way to compute it in the frequency domain, by taking advantage of the Convolution Theorem; to do this we need first to calculate the Discrete Fourier Transform of the two signals and then multiply together the first and the complex conjugate[3] of the second one; given F and G the two discrete transforms of the original signals, this operation is reduced to:

$$H[k] = F[k] \cdot \overline{G[k]} \tag{3.3}$$

Then the Inverse Fourier Transform of H must be computed to obtain $(f \star g)[n]$; finally we can simply apply equation 3.2 on the result.

Since the Fourier Transform of the acquired audio is necessary in order to display it in the GUI and the transforms of the pre-recorded signals can be computed offline during the configuration phase, it is simpler and more efficient to compute the non normalized correlation value by using Fourier Transforms.

3.2 Normalized Cross Correlation

Once the normalized correlation value between two signals can be easily obtained from the non normalized one and the two non normalized auto-correlations of the two signals; the latter are defined as the non normalized correlations of each signal with itself. The following formula is used to compute the normalized correlation value:

$$\widehat{R}_{f,g} = \frac{R_{f,g} \cdot R_{f,g}}{R_{f,f} \cdot R_{g,g}} \tag{3.4}$$

Experimental results have shown how usually similar signals have an absolute value of $\widehat{R}_{f,g}$ greater or equal than ~ 0.3, however this threshold will be left as a configurable parameter for the program during compilation phase.

4. Implementation

The implementation of the project can be found on GitHub using the following url: `https://github.com/gabrieleara/super-surface-player`.

This section will illustrate briefly the content of the project.

[2] We assume both signals are real.

[3] Calculating the complex conjugate of a Fourier Transform is equivalent to obtaining the Fourier Transform of the time-reversed signal.

4.1 Project Structure

Project structure is shown in Figure 4.1, where some files have been omitted for a better clarity. Some of the folders are automatically generated and filled during project build by `make` command and thus their content has not been expanded in the Figure.

Each folder has a different scope:

`dist` contains build targets and Doxygen documentation

`dep` contains automatically generated prerequisites for `make` rules, as `*.d` files

`docs` contains Doxygen configuration file

`inc` contains header files, divided between API-related files and application-specific ones

`obj` contains automatically generated object files

`res` contains bitmaps used to draw the graphical user interface

`src` contains source files, divided between API-related files and application-specific ones

The three modules described by their requirements in section 2 have each a different source/header files pair assigned, while globally shared constants are placed in `constants.h` file.

4.1.1 Compilation Parameters

While some of those constants should not be modified, the first section of said file has been purposely created to configure some build parameters for the program (default values are between parentheses when relevant):

`AUDIO_DESIRED_RATE` is the desired acquisition rate in Hertz (44100).

`AUDIO_DESIRED_FRAMES` the desired number of frames contained within the audio acquisition window (4096); it should be a power of two to make the Fast Fourier Transform computation faster. Notice that increasing this value may lead to more precise sound recognition, at the cost of a greater latency.

`AUDIO_ENABLE_PADDING` is a Boolean value that enables zero padding when calculating the Fast Fourier Transform (1); padding with zeros is a simple way to increase the resolution of the FFT without increasing the dimension of the acquisition window.

`AUDIO_PADDING_RATIO` is the ratio between the number of samples with zero padding (if enabled) and the number of samples in the acquisition window (4).

`AUDIO_THRESHOLD` is the threshold that is used to check whether the recorded audio corresponds to a pre-recorded audio sample (0.3).

`AUDIO_ANALYSIS_DELAY_MS` is the minimum interarrival time between two samples that needs to elapse for them to be recognized as the same pre-recorded sample, in milliseconds (500); this value should be tuned when sustaining input produces overlapping or undesired playbacks.

FFT_PLOT_SCALING controls the scaling of the spectrum plot.

TIME_MAX_AMPLITUDE controls the scaling of the time plot.

AUDIO_LATENCY_REDUCER reduces the period of the tasks within the audio module by the given divider (8); this can be used to increase the reactivity of the system at the cost of additional task wake ups, see section 4.6.1.

AUDIO_APERIODIC if defined compiles the aperiodic version of the program, see section 4.6.1; by default this constant is not defined and it can even be defined direcly in the building command: to do so, perform first a `make clean` command, followed by a `make CFLAGS=-DAUDIO_APERIODIC` .

4.2 Documentation

The whole project is documented using Doxygen. Such documentation is generated automatically whenever the project is built in *Release* mode and it is also available at `https://codedocs.xyz/gabrieleara/super-surface-player/files.html`.

4.3 Libraries

The application is built using a few Open Source libraries:

- Allegro 4.4.2 [`https://liballeg.org/old.html`] for the graphic environment and a few of the routines that open and playback wave audio files;
- Advanced Linux Sound Architecture (ALSA) [`https://www.alsa-project.org`] as audio driver for recording and playback of recorded audio samples;
- Fastest Fourier Transform in the West (FFTW) [`http://www.fftw.org`] as efficient and configurable implementation of the Fast Fourier Transform algorithm.

The program also requires the support from the system for the Linux Real-Time Schedulers (`librt`), using a custom library for task definition and handling that basically wraps the POSIX Thread (`libpthread`) library for the necessary operations. The custom library is based on the data structures and routines shown by Professor Giorgio Buttazzo during his Real-Time System course, see `http://retis.sssup.it/~giorgio/rts-MECS.html` for further details.

4.4 Data Structures and Communication

Communication between components is obtained by accessing globally shared data structures; each module defines a few global structures both for internal communication and to exchange data with the other modules. Access to data structures is protected by mutex semaphores, occasionally using condition variables when a task needs to wait for a certain event in a critical section.

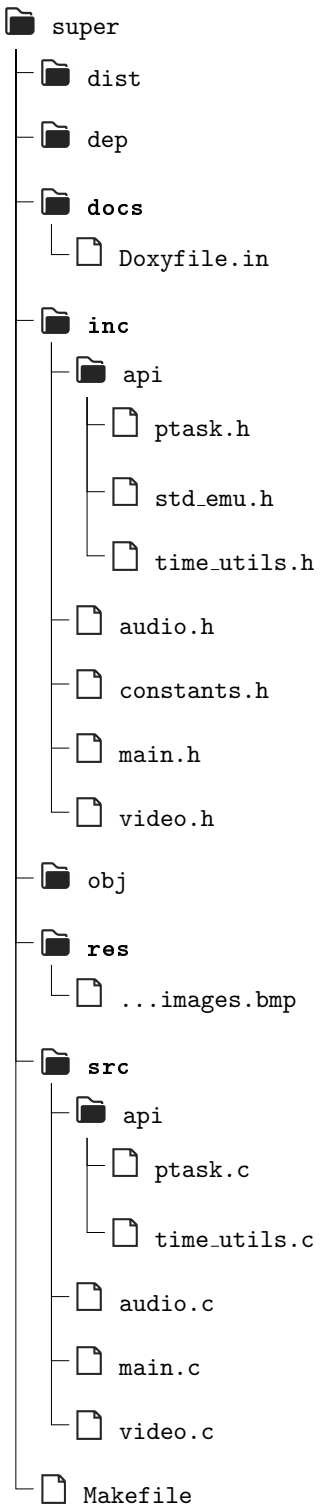

Figure 4.1: Main project folders and files. Folders that are not highlighted in bold are automatically generated by `make` command during project build.

To avoid global variables names pollution, each module defines his own global variables as attributes of a data structure and then allocates a single actual global variable that contains all the globally shared attributes of such module; the variable is even declared as `static`, thus communication between modules can happen only via non-`static` functions exposed by each of the modules.

Critical sections are kept as small as possible, usually they consist in the copy from/to a few local variables for later usage. However, the application needs also to exchange big amounts of data when audio samples collected from the microphone need to be propagated to other components.

For this purpose, a small library implementing Cyclic Asynchronous Buffers (CAB) have been developed; CABs avoid access conflicts to big data structures by replicating internal buffers and maintaining pointers to the most recently updated buffer in the data structure. This way, whenever a new buffer has to be published the writer can simply commit operations performed on a previously reserved buffer (which involve only changing a few attributes) and readers can continue using the buffers they already acquired without any interference. This reduces blocking times when accessing big amounts of shared data to a minimum, decoupling tasks from each other.

The library does not allocate arrays that will be used for buffers; instead they should be allocated in an appropriate section of the program, such that the actual buffers life cycle is longer than the CAB structure used to access them; this is accomplished in the application by allocating both buffers and the CAB structure in global shared data structures.

The assumption on which the library relies is that the allocated CAB will have a sufficient number of buffers with respect to the number of tasks that will potentially use it. If a CAB is used by N tasks, to avoid blocking for a buffer allocation there should be at least $N + 1$ buffers; the $(N + 1)$-th buffer is used to keep the most recent message in the case all the other buffers are used, each by a different task.

4.4.1 Main Module Data Structures

The global data structure containing all global shared attribute of the main module is the following one:

```
/// Structure containing the global state of the program
typedef struct
{
    bool            tasks_terminate;///< Tells if concurrent tasks should stop
                                    ///< their execution
    bool            quit;           ///< Tells if the program is shutting down
    bool            log_level;      ///< The system log level
    char            directory[MAX_DIRECTORY_LENGTH];
                                    ///< The specified directory where to search
                                    ///< for audio files (also referred as the
                                    ///< current working directory)
    ptask_t         tasks[TASK_NUM];///< All the tasks data
    ptask_mutex_t   mutex;          ///< Protects access to this data structure
    ptask_cond_t    cond;           ///< used to wake up the main thread when in
```

```
                                    ///< graphical mode
} main_state_t;
```

Most relevant attributes are `task_terminate` one, which is used to signal concurrent tasks that they should terminate gracefully their execution, the pointers to the concurrent tasks that have been created and the pair of mutex and condition variables used to suspend main thread execution when the program is in active phase, waiting for the user to request the termination of said phase.

4.4.2 Audio Module Data Structures

There are a few data structures that are used by the audio module to keep a consistent state thorough the execution of the program.

Audio File Descriptor

Each wave audio file that is opened during program execution has an associated data structure of the following type:

```
/// Opened audio file descriptor
typedef struct
{
    audio_pointer_t datap;      ///< Pointer to the opened file
    audio_type_t    type;       ///< File type
    int             volume;     ///< Volume used when playing this file
    int             panning;    ///< Panning used when playing this file
    int             frequency;  ///< Frequency used when playing this file,
                                ///< 1000 is the base frequency
    bool            has_rec;    ///< Tells if the file has an associated
                                ///< recorded audio that can be recognized to
                                ///< start it playing

    char            filename[MAX_AUDIO_NAME_LENGTH];
                                ///< Name of the file displayed on the screen,
                                ///< contains only the basename, ellipsed if
                                ///< too long

    double          autocorr;   ///< The cross correlation of the signal with
                                ///< itself

    short           recorded_sample[AUDIO_DESIRED_BUFFER_SIZE];
                                ///< Contains the recorded sample by the user
                                ///< to play the audio file whenever a sample
                                ///< similar to the recorded one is detected

    double          recorded_fft[AUDIO_DESIRED_PADBUFFER_SIZE];
                                ///< Contains the FFT of the recorded sample,
```

```
                                ///< precomputed for efficiency reasons
} audio_file_desc_t;
```

The last three attributes of the data structure are meaningful only if the `has_rec` one is `true` and they represent characteristics of the associated recorded sample that are computed offline for later usage. This means that an opened file can only be associated with a single recorded sample, if multiple triggers should be associated to the same file, it can be opened multiple times and be associated to different triggers.

Fast Fourier Transform Descriptor

As already stated, whenever the application has to exchange big amounts of data a CAB is used; to exchange Fast Fourier Transforms of the recorded samples each buffer is actually an instance of the following data structure:

```
/**
 * Structure used to publish a new FFT obtained from continuous microphone
 * recording.
 */
typedef struct
{
    double  autocorr;           ///< The auto-correlation of the given sample
    double  fft[AUDIO_DESIRED_PADBUFFER_SIZE];
                                ///< The FFT of the given sample, its format is
                                ///< Halfcomplex-formatted FFT
} fft_output_t;
```

This way a transform and its associated auto-correlation can be published together. The structure that contains both the buffers and the associated CAB is the following one:

```
/// Status of the resources used to perform fft
typedef struct
{
    unsigned int        rrate;  ///< Recording acquisition ratio, as accepted
                                ///< by the device
    snd_pcm_uframes_t   rframes;///< Period requested to recorder task,
                                ///< expressed in terms of number of frames
    fftw_plan           plan;   ///< The plan used to perform the FFT
    fftw_plan           plan_inverse;
                                ///< The plan used to perform the inverse FFT
    fft_output_t        buffers[AUDIO_FFT_NUM_BUFFERS];
                                ///< Buffers used within the cab
    ptask_cab_t         cab;    ///< CAB used to handle allocated buffers
} audio_fft_t;
```

The two attributes `plan` and `plan_inverse` are used to configure parameters for the FFTW library to compute forward and inverse Fourier Transforms.

The Audio Global Data Structure

All the previously illustrated data structures, along with some minor other ones, have been grouped within another data structure:

```
/// Global state of the module
typedef struct
{
    audio_file_desc_t   audio_files[AUDIO_MAX_FILES];
                            ///< Array of opened audio files descriptors for
                            ///< reproduction
    int                 audio_files_opened;
                            ///< Number of currently opened audio files
    audio_record_t      record; ///< Contains all the data needed to record
    audio_fft_t         fft;    ///< Contains all the data needed to perform
                            ///< FFT and IFFT
    audio_analysis_t    analysis;///< Contains all the data needed to perform
                            ///< analysis of FFTs
    ptask_mutex_t       mutex;  ///< Protects access to opened files attributes
                            ///< in multi-threaded environment.
} audio_state_t;
```

4.4.3 Video Module Data Structures

The video module stores the few bitmaps used to draw the graphic interface into a global structure, to use them multiple times:

```
/**
 * It contains all the static elements of the gui (backgrounds), reused when
 * multiple elements of the same time are drawn.
 */
typedef struct
{
    BITMAP* background;     ///< Static background of the interface
    BITMAP* element_sample; ///< Static background of an element of the
                            ///< interface representing an opened sample
    BITMAP* element_midi;   ///< Static background of an element of the
                            ///< interface representing an opened midi
} gui_static_t;
```

The global data structure containing all global shared attribute of the video module is then the following one:

```
/// Global state of the module
typedef struct
{
    BITMAP*     virtual_screen; ///< Virtual screen, used to handle all gui
```

```
                                ///< operations, it is copied in the actual
                                ///< screen when refreshed
    gui_static_t static_screen; ///< Contains all the gui bitmaps
    bool        initialized;    ///< Tells if this interface has been
                                ///< initialized, i.e.\ all bitmaps are loaded
                                ///< and so on
    bool        mouse_initialized;
                                ///< Tells if the mouse handler has been
                                ///< initialized
    bool        mouse_shown;    ///< Tells if the show_mouse function has been
                                ///< called on the current screen
    ptask_mutex_t mutex;        ///< This mutex is used to protect the access
                                ///< to mouse flags
} gui_state_t;
```

Most relevant attribute is the `virtual_screen` one, which is used to draw at each screen refresh the whole screen, which is then copied on the actual screen only at the end.

4.5 Concurrency Model

As the program is divided in two phases, also the concurrency model of the program changes from a phasde to another:

1. In the configuration phase, the program has only one thread, namely the main thread, which handles user input, records audio, and so on.

2. In the active phase, the program has multiple tasks that execute concurrently while the main thread is suspended waiting for the termination of the active phase.

Functions developed for the program actually differentiate between thread-safe ones (that can be used during active phase from multiple threads) and the ones that can be used only when there is a single thread. Since the only phase with actual concurrent tasks is the active one, further reasoning about concurrency is related to that phase alone.

For the implementation of the requirements as per section 2, the following tasks have been defined:

Graphical User Interface Task (GUI Task) is the one that is responsible to refresh the content of the graphical window of the program.

User Interaction Task (UI Task) is the one that is responsible to handle keyboard and mouse input from the user when the graphical window is focused.

Microphone Task is the one that is responsible to copy data from the audio driver buffer into globally shared audio buffers, performing the preliminary FFT and auto-correlation evaluation on the acquired audio samples and publish all the computed data.

Analysis Tasks are the ones that should recognize sounds collected by the Microphone Task; their number varies depending on the number of opened files with an associated pre-recorded sample: one task is spawned for each pre-recorded sample.

In addition, another task called the **Checker Task** is used if the non-periodic version of the application is compiled, see section 4.6.2 for further details about that task and how other tasks parameters change in that version.

In the following sections we will illustrate the parameters of each of those tasks; all the tasks deadlines are implicitly declared equal to their periods and their priorities are assigned proportionally to the inverse of their periods. For this reason, a fixed priority scheduler is sufficient to schedule the tasks in the program; thus the scheduler adopted by the program is `SCHED_FIFO`.

4.5.1 Graphical User Interface Task

The GUI Task is supposed to refresh the screen at 60 Hz, hence its period is statically set to 16 ms and thus its priority is the lowest one in the system.

4.5.2 User Interaction Task

The UI Task is supposed to handle user input; typically, tasks like this one run with a frequency of 60 Hz, hence its period is statically set to 10 ms.

4.5.3 Microphone Task

This task should run once every time there is at least one acquisition window full of audio samples in the audio driver buffer; this varies depending on the compilation parameters that can be specified in `constants.h` file, as described in section 4.1.1.

From those parameters, if we express as N_{WIN} the value of `AUDIO_DESIRED_FRAMES`, L is the value of `AUDIO_LATENCY_REDUCER` (see section 4.6.1) and R_{MIC} is the acquisition rate as defined in `AUDIO_DESIRED_RATE`, we can obtain the following value for the period of the Microphone Task in (milliseconds):

$$1000 \cdot \frac{N_{WIN}}{L} \cdot \frac{1}{R_{MIC}} \tag{4.1}$$

This means that this task has the highest priority of the system[1], along with each of the Analysis Tasks.

4.5.4 Analysis Tasks

For the same reasons described in the previous section, the period and the priority of these tasks are the same as the ones of Microphone Task.

[1] This is no more true for the Non-Periodic version of the program, see section 4.6.2.

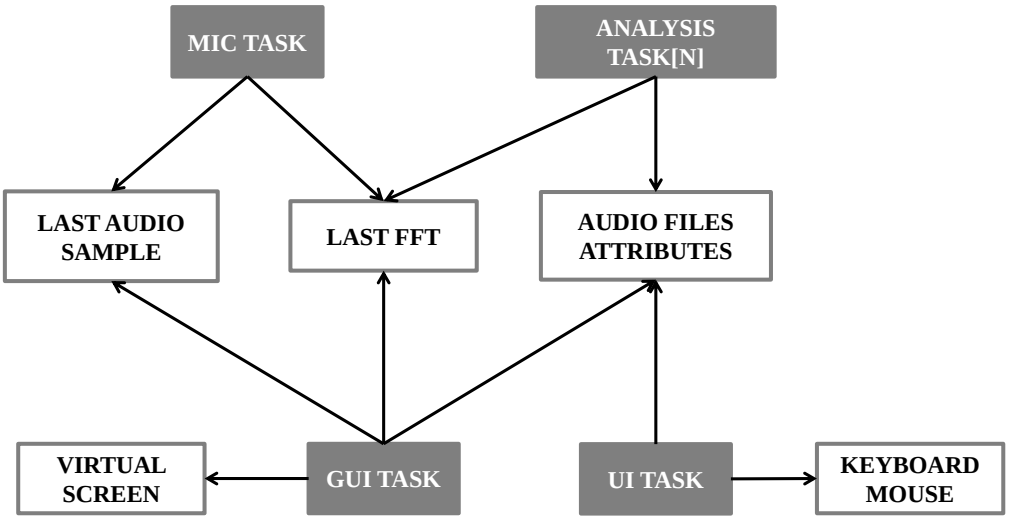

Figure 4.2: Relationship between real-time tasks and shared resources.

4.5.5 Tasks and Shared Resources

The relationship between real-time tasks and main shared global resources is depicted in Figure 4.2.

More in details:

- The GUI Task needs to access data for the last audio sample, the last Fourier Transform and attributes of all of the files to refresh the content of the graphical window, via the virtual screen bitmap image.
- The UI Task needs to modify attributes of audio files and request audio files playback whenever an input from the keyboard/mouse requests it.
- The Microphone Task is the one responsible to update the last acquired audio sample and compute its Fourier Transform.
- The Analysis Task needs to access with read-only privileges certain attributes of the opened audio files that cannot be modified by any other task during the active phase, thus its only source of interference is the access to the last Fourier Transform available.

Notice however that access to the last audio sample and the last Fourier Transform is regulated via the usage of CABs, which reduces to the minimum the interference between the tasks. The other critical sections are so small that there is virtually no way to reduce interference any further.

4.6 Design Choices

The implementation of the program has been strongly driven by the requirement of reducing as much as possible the delay between user input via the microphone and the response of the system. Following is the analysis of the worst case scenario.

4.6.1 Reactivity to Audio Inputs

The reactivity of the system to external audio inputs (that is the delay between the user performs the triggering sound and the corresponding audio file is triggered) is a major concern for this kind of applications, because delays affect negatively the outcome of the musical performance.

The steps that the system must go through from the initial user input to the beginning of the corresponding audio playback are the following ones:

1. The sound card driver needs to store at least a certain amount of data inside a system buffer; the minimum number of samples that the sound card driver has to collect before any of them may become visible from user-space will be referred as *acquisition window*.

2. The recorder task needs to copy said data from the system buffer to a user space buffer.

3. A preliminary analysis of the acquired audio window is performed, to extract the attributes that characterize it (see section 3); since this step is common between all possible recorded sounds, this action can be performed by the recorder task itself between one audio acquisition and the next, as long as this operation cannot delay the next activation of the recorder task.

4. Results of previous operations are published on a global data structure, updating most recent data.

5. The task responsible for the recognition of that particular sound acquires those stats and performs some operations that indeed prove that the recorded audio correspond to its associated sample.

6. Finally, the playback of the given audio sample can be requested within the same task.

The worst-case scenario for the system reactivity is depicted in Figure 4.3, where it is clear how the delay between user input and the system response might be influenced by tasks activation times. In the scheme, T_{window} refers to the duration of the audio acquisition window, T_{MIC} is the period of the recording task, T_{ALS} is the period of the analysis (recognition) task. The actions performed by each task (other than checking for new data) are identified by letters: (A) copy data from kernel to user space; (B) preliminary analysis; (C) per-sample analysis.

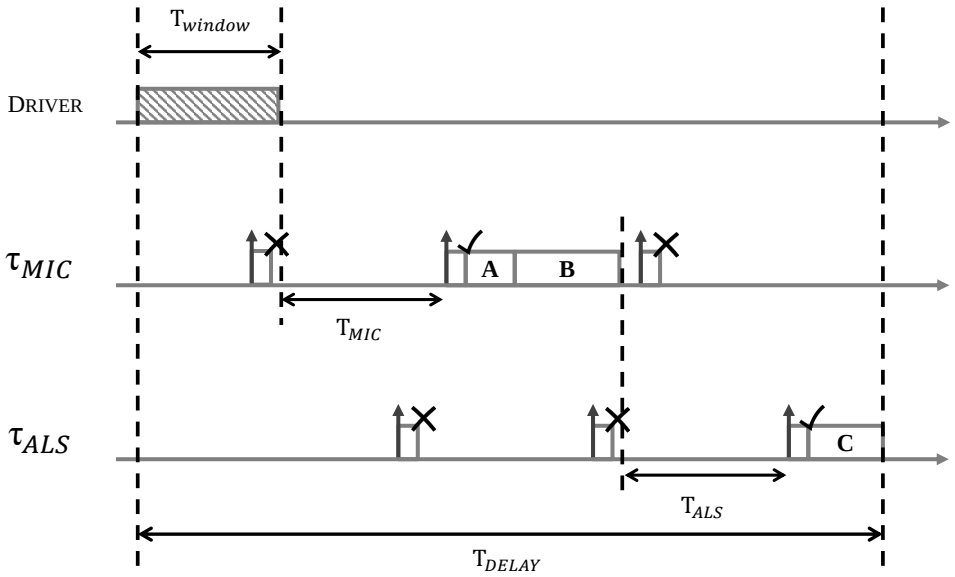

Figure 4.3: Worst-case scenario for system reactivity in the case of a periodic recorder task.

4.6.2 Non-Periodic Version of the Application

Of course, reducing the period of the displayed tasks might result in much smaller response times, but on the other hand this might lead to many unnecessary task activations, like the ones shown in the Figure 4.3 with a crossed mark. In that cases, all the tasks do is notice that there is no actual new data to process and get back asleep until next period.

An alternative that might reduce this delay while maintaining most of the predictable behavior of the system consists in switching from a periodically-activated task to an event-based one for the recording task; in this situation, the audio acquisition driver or another service might wake up the recording task every time there is enough data in the buffer to be copied.

This solution can eliminate completely the delay introduced by T_{MIC} (or reduce it to a minimum), without however completely jeopardizing the predictability of the system. In fact, the constant throughput of the data produced means that the behavior of the microphone task will resemble the one of a periodic task, even if it is not activated by a timer. Its period will be about the same of the audio acquisition window, while any difference in phase between the two would be virtually eliminated.

The resulting worst-case scenario is depicted in Figure 4.4. Since there may be tasks assigned to audio recognition (one for each pre-recorded sample), we decided to leave them as periodic tasks, reducing their nominal period.

The final system will contain both solutions in its implementation and a define constant will be used to compile one version or the other (usually referred as the periodic or the non-periodic one, even if that characteristic is referring to the microphone task only).

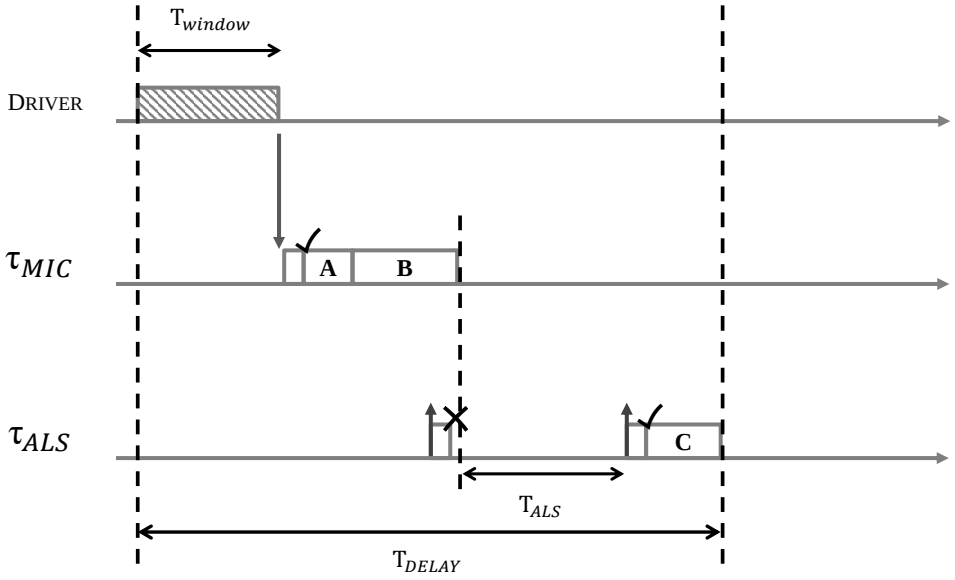

Figure 4.4: Worst-case scenario for system reactivity in the case of a non-periodic recorder task.

4.6.3 Task Parameters in Non-Periodic Version

Since the Microphone Task cannot be implemented as a periodic task anymore, an additional periodic task has been defined, called the Checker Task, whose only purpose is to check the amount of data present in the audio driver buffer and notify the Microphone Task when it reaches the desired threshold.

Since this task is very light and it should check the actual value as many times as possible per second, its period has been set to the minimum possible with the task handling library, that is 1 ms, which means that the priority of the task is the highest one.

The period of the Microphone Task has no meaning anymore[2], however the Analysis Tasks remain periodic and the original analysis used to determine their period is still valid and it will have a priority second only to the one of the Checker Task.

5. Conclusion

A few experimental tests have been performed to check the behavior of the application. While we can't show any particular statistic about the system performance in sound recognition, the system performed pretty well when specific sounds need to be recognized.

This means that for example a specific note played on a guitar string can be easily recognized, while it is virtually impossible to make the program recognize generic sounds (like "clapping hands") because of the nature of the sound recognition algorithm, the program has been able to recognize recorded sounds even in noisy environment or when

[2]Its priority will remain unchanged for scheduling purposes.

the sound is just a part of a composition (for example a specific note within a chord), as long as the sound to noise ratio is not too high.

The program responsiveness is also reasonable (well under a tenth of a second) for music production, which was a key aspect of the project requirements; however this depends on the kind of sound that is used as trigger, as explained in the next section.

5.1 Future Works

While the delay of the system response is minimal when the actual pre-recorded sound is executed in front of the microphone, the way the pre-recorded samples are acquired may lead to additional delays. In fact, for the sake of simplicity the program performs a small countdown at the end of which it records automatically for the amount of time equivalent to an acquisition window, which is usually so small that it's difficult to react accordingly, for example when trying to record a picked string on a guitar.

This means that most of the times the pre-recorded sounds will consist of notes that are sustained for long times, that are different from the ones produced at the exact moment the plectrum hits the guitar string; this means that during the active phase there will be a noticeable delay between the plectrum hitting the string and the response of the program, because the very first sample will have a different spectrum with respect to the pre-recorded one, while the following ones will be practically identical to the stored sample.

An improved way to record and select parts of a recorded sample as pre-recorded base for audio triggers will be crucial for further improvements of the system. However this is left for future versions of the application.