PulseAudio and the knife-alien sound processor – Taking MATLAB to the next level

Patrik Dahlström (patda293), *Electronic design programme, Linköping university*

***Abstract*—The knife-alien sound processing engine is an easy and intuitive tool to manipulate audio streams, but currently lacks satisfactory audio output. This report describes an attempt to use PulseAudio for audio playback. The implementation presented however does not produce smooth and low-latency playback.**

The appendix describe how the PulseAudio implementation can be used together with SIMULINK.

INTRODUCTION

Sound processing is a subject that is as current today as it was 50 years ago. The knife-alien sound processing engine presents an easy and intuitive way to manipulate an audio stream. Filters are easily added and removed, without the need of stopping and starting recording.

However, MATLABs sound output is far too slow to effectively output audio at the same rate as audio is recorded, and the output become choppy. Therefore an external library is used to decrease latency and choppiness: PulseAudio.

This report will describe the various parts of knife-alien that the author was the major contributor to as well as an evaluation of PulseAudio and how well it substitutes MATLABs built-in audio functions.

The appendix contains a chapter on how PulseAudio can be used together with SIMULINK

I. KNIFE-ALIEN

The knife-alien sound processing engine implement a combination of MATLABs events and listeners concept and a linked list of filters.

A. Functional overview

The filtering process can be described in 5 steps:

1) An audio source notifies the event NewAudioData
   a) The topmost graph of the GUI present the frequency spectrum of this data.
   b) The new audio data is passed to a dummy filter which does not alter the signal.
2) The data progress through all active filters in order, each filter notifying the event FilteringComplete.
3) The FilteringComplete event of the last filter (which also is a dummy filter) triggers the listener callback functions that will save and play the filtered audio, as well as present it in the bottommost graph of the GUI.

See **??** for a graphical overview of this process.

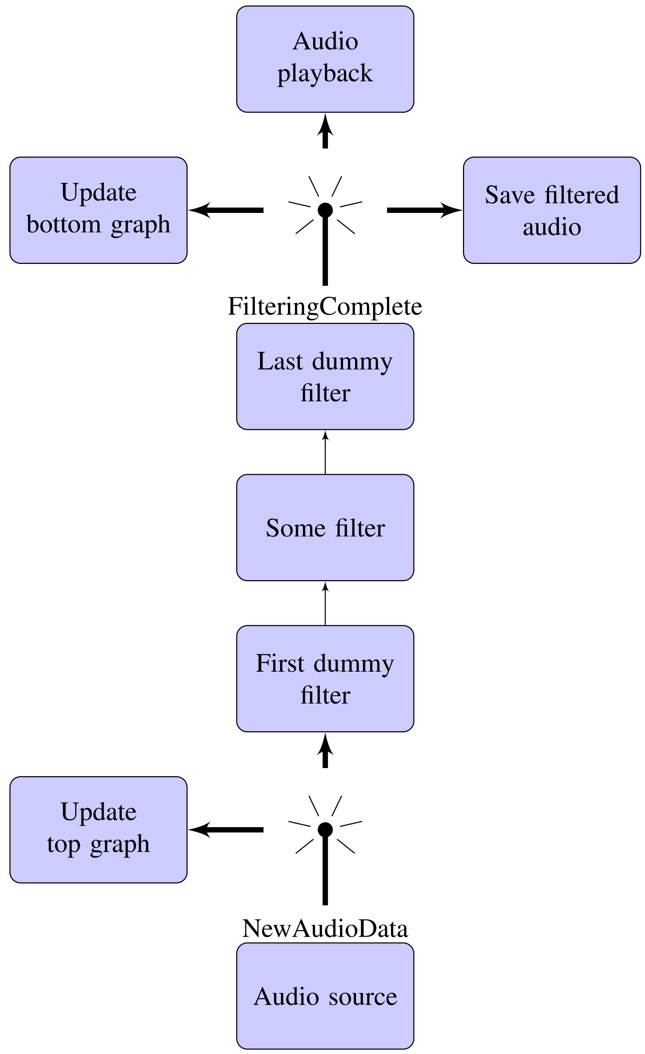

Fig. 1. Overview of knife alien filtering procedure

B. Audio source

Currently, there are two types of audio sources available in knife-alien: from a microphone or from a wave file[1]. The classes responsible for each audio source are *CustomAudioRecorder* and *CustomAudioPlayer* respectively.

All audio sources perform a Fast Fourier Transform on its data before notifying NewAudioData.

For recording audio from a microphone the MATLAB class *audiorecorder* was used as base class. *CustomAudioRecorder* adds some properties as well as an event – NewAudioData.

[1]Currently under development

MATLABs *audiorecorder* class provide the functionality of executing a callback function with regular intervals. This functionality is used by *CustomAudioRecorder* to call its member function customTimerFcn(). customTimerFcn() calculates how many new samples have been recorded since last time it was executed and performs a fast Fourier transform on only the new audio data before notifying the NewAudioData event.

The source code of *CustomAudioRecorder* is provided in **??**.

Listing 1. CustomAudioRecorder.m: Source code of CustomAudioRecorder

```
classdef CustomAudioRecorder < audiorecorder & handle
    properties
        listener
    end
    properties (SetAccess = protected)
        Data
        Fs
        Nfft
        lastSample
    end
    methods
        function obj = CustomAudioRecorder(Fs,nBits,nChannels)
            obj = obj@audiorecorder(Fs,nBits,nChannels);
            obj.Fs = Fs;
            obj.TimerFcn = @obj.customTimerFcn;
        end
        function customTimerFcn(obj,src,eventData)
            audioData = getaudiodata(src);
            % Update number of samples since last time this function ran
            obj.Nfft = obj.TotalSamples - obj.lastSample;
            obj.lastSample = obj.TotalSamples;
            % Only process last Nfft samples
            audioData = audioData(end-obj.Nfft:end);
            audioData = fft(audioData,obj.Nfft);
            obj.Data = audioData;
            notify(obj,'NewAudioData');
        end
        function reset(obj)
            obj.lastSample = 0;
        end
    end
    methods (Static,Hidden)
        % Overide audiorecorder's private, hidden validateFcn.
        % Had to make this function static, don't really know why.
        function validateFcn(fcn)
        end
    end
    events
        NewAudioData
    end
end
```

Documentation for *CustomAudioPlayer* is not contained in this document.

C. Filters

All filters of the knife-alien sound processing engine are contained in a MATLAB package as to separate them from the rest of the knife-alien namespace. This avoids the problem of function name disambiguation with other functions in the knife-alien root tree.

The filters use an abstract superclass called *FilterClass* that defines properties common for all filters. These properties are:

- Fs – Sample rate
- UserData – Could be anything
- Next – A handle to the next filter in a chain of filters
- Prev – A handle to the previous filter in a chain of filters
- Data – The actual data
- Name – A textual representation of the filter

These properties are hidden so that they don't show up in the filter configuration box in the main GUI.

FilterClass also define an abstract function, filter(), that all subclasses of *FilterClass* have to implement. The only function implemented by *FilterClass* is eventHandler() which acts as wrapper to pass on data to filter().

See **??** for source code.

Listing 2. FilterClass.m: Superclass of all filters

```
classdef (Hidden=true) FilterClass < handle
    properties (Hidden=true)
        Fs
        UserData
        Next
        Prev
    end
    properties (SetAccess = protected,Hidden = true)
        Data
        Name
    end
    methods (Abstract=true)
        filteredData = filter(obj,data)
    end
    methods
        function eventHandler(obj,src,eventData)
            obj.filter(src.Data);
        end
    end
    events
        FilteringComplete
    end
end
```

Each filter's implementation of filter() has to call the filter() function of the next filter – obj.Next.filter() – for the filtering chain to work. It is also responsible for notifying FilteringComplete although it is not mandatory.

Each filter's constructor is also responsible for populating the Name property.

D. Save filtered data

Upon creation, knife-alien defines a series of listener callbacks for the FilteringComplete event of the last dummy filter. One of those callback functions is saveFilteredAudio() shown in **??**. This function appends data to the file recorded_audio.wav or creates it if it doesn't exist.

Listing 3. saveFilteredAudio.m: Append filtered audio to recorded_audio.wav

```
function saveFilteredAudio(obj,eventData)
audioData = ifft(obj.Data);
Y = 0;
if exist('recorded_audio.wav','file')
    Y = wavread('recorded_audio.wav');
end
Y = [Y; audioData];
wavwrite(Y,obj.Fs,'recorded_audio.wav');
```

E. Audio playback

Another listener callback function defined on startup is playFilteredAudio(). When data arrives to this function it is Fourier transformed and of type double. For audio playback this data has to be inversely transformed and converted to int16 to be compatible with the playAudio() function call. Only data of more than 512 samples are played to increase performance.

The UserData property of the last dummy filter contain the address of an C object needed by playAudio(). More on this in **??**.

Listing 4. playFilteredAudio.m: Inverse transform, convert and play data

```
function playFilteredAudio(obj,eventData)
if numel(obj.Data) > 512
    audioData = int16(ifft(obj.Data)*32767);
    playAudio(obj.UserData,audioData);
end
```

F. Update bottom graph

How the bottom graph is updated is beyond the scope of this document.

II. PulseAudio

PulseAudio is a sound system for POSIX system, meaning that it is a proxy for sound applications. It is an integral part of several popular Linux distributions and is used by various mobile devices.

PulseAudio is built on a server-client model, in which knife-alien is a client. A client connects to a PulseAudio server and lets the server mix the audio from several sources to a single output sound. This makes mixing audio from several applications easy. Each application can in turn have several streams of audio that is multiplexed by a context object through a single connection to the PulseAudio server.

The API for PulseAudio features two separate flavours: the simple API and the asynchronous API.

A. Simple API

For very basic audio output the simple API is enough. It features a reduced set of function calls aimed towards the very basic need of audio playback. It only supports a single stream per connection and has no handling of complex features like events, channel mappings and volume control.

Audio playback with the simple API is done in 3 steps

1) Connecting to the server
2) Transfer data
3) Play data

Note however that the simple API is a blocking API, i.e. does not return control until the audio has been played.

The 3 steps necessary to play back audio from MATLAB data can be seen in **??**.

Listing 5. playAudio.c: Playing audio using the simple API

```
#include "mex.h"
#include <pulse/simple.h>

void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    const mxArray *Data = prhs[0];
    int16_t *data = (int16_t*)mxGetData(prhs[0]);
    size_t r = mxGetN(Data);

    // PA code
    pa_simple *s;
    pa_sample_spec ss;

    ss.format = PA_SAMPLE_S16LE;
    ss.channels = 1;
    ss.rate = 22050;

    s = pa_simple_new(NULL,            // Use the default server.
                      "knife-alien",   // Our application's name.
                      PA_STREAM_PLAYBACK,
                      NULL,            // Use the default device.
                      "Music",         // Description of our stream.
                      &ss,             // Our sample format.
                      NULL,            // Use default channel map
                      NULL,            // Use default buffering attributes.
                      NULL             // Ignore error code.
                      );

    pa_simple_write(s, data, (size_t)r, NULL);
}
```

Please note that this implementation is no longer available in the knife-alien source code.

B. Asynchronous API

This API allows full access to all of PulseAudio functionality, but is therefore also more complex. It is based around an asynchronous event loop, or main loop, and PulseAudio offers 3 different implementations of this loop. For knife-alien the *Main Loop* implementation was chosen for fast prototyping.

When an event loop implementation has been chosen, a context object has to be created. This context will multiplex events, commands and streams to the server. It is unnecessary for more than 1 context unless connections to multiple servers are wanted. Once the context is created it has to be connected to the server.

See **??** for how an event loop implementation is chosen and a context is created and connected to a PulseAudio server.

Listing 6. initPulseaudio.c: Choose event loop, creating context and connect

```
    pa_ptrs.pa_ml = pa_mainloop_new();
    pa_ptrs.pa_mlapi = pa_mainloop_get_api(pa_ptrs.pa_ml);
    pa_ptrs.pa_ctx = pa_context_new(pa_ptrs.pa_mlapi, "knife-alien");

    // This function connects to the pulse server
    pa_context_connect(pa_ptrs.pa_ctx, NULL, 0, NULL);
```

All audio is transferred in a stream. For knife-alien a single stream is sufficient for playing the filtered audio. An object representing the stream has to be created both on the client as well as on the server. This is shown in **??**.

Listing 7. initPulseaudio.c: Create stream on both client and server

```
    pa_ptrs.pa_s = pa_stream_new(pa_ptrs.pa_ctx,"knife-alien",&pa_ptrs.pa_ss,NULL);
    pa_stream_connect_playback(pa_ptrs.pa_s, // The stream to connect to a sink
            NULL, // Name of the sink to connect to, or NULL for default
            NULL, // Buffering attributes, or NULL for default
            PA_STREAM_NOFLAGS, // Additional flags, or 0 for default
            NULL, // Initial volume, or NULL for default
            NULL // Synchronize this stream with the specified one, or NULL for a
                 standalone stream
            );
```

The stream can now be used to transfer data to the PulseAudio server, but the data will not be played until the stream is drained. The pa_stream_drain() function does this and returns a pa_operation pointer that can be used to monitor the state of the drain operation.

Listing 8. playAudio.c: Transfer and play audio

```
    // Play
    pa_stream_write(pa_ptrs.pa_s,data,r,NULL,0,PA_SEEK_RELATIVE);
    o=pa_stream_drain(pa_ptrs.pa_s,0,NULL);
    while(pa_operation_get_state(o) != PA_OPERATION_DONE) {
        pa_mainloop_iterate(pa_ptrs.pa_ml,1,NULL);
    }
    pa_operation_unref(o);
```

Last but not least it is important to mention that nothing happens unless the event loop is advanced, made possible by the pa_mainloop_iterate() function call.

When the application using PulseAudio is closed it is important to destroy the stream and disconnect the context connection. This is handled by destroyPulseaudio.c as seen in **??**.

Listing 9. destroyPulseaudio.c: Destroy and disconnect stream and context

```
    // Shutdown everything
    pa_stream_disconnect(pa_ptrs.pa_s);
    pa_context_disconnect(pa_ptrs.pa_ctx);
    pa_context_unref(pa_ptrs.pa_ctx);
    pa_mainloop_free(pa_ptrs.pa_ml);
    mexPrintf("Pulseaudio destroyed\n");
}
```

C. Sharing C objects between function calls

The pa_ptrs struct shown in the above C examples and in **??** is declared static to make it persistent between function calls. The address of the struct is returned from initPulseAudio.c and is the first argument to both playAudio.c and destroyPulseaudio.c.

Listing 10. initPulseaudio.c: Definition of pa_ptrs

```
static struct pa_settings {
    // Define our pulse audio loop and connection variables
    pa_mainloop *pa_ml;
    pa_mainloop_api *pa_mlapi;
    pa_context *pa_ctx;
    pa_stream *pa_s;
    pa_sample_spec pa_ss;
}pa_ptrs;
```

?? and **??** show how the pa_ptrs struct is shared between function calls.

Listing 11. initPulseaudio.c: Save object address as return value

```
    plhs[0]=mxCreateNumericMatrix(1,1,mxUINT64_CLASS,mxREAL);
    uint64_t *ptr = (uint64_t)mxGetData(plhs[0]);
    *ptr = &pa_ptrs;
```

Listing 12. playAudio.c: Retrieve object from input argument

```
    // Get PA settings
    uint64_t ptr = *(uint64_t*)(mxGetData(prhs[0]));
    pa_settings_t pa_ptrs = *(struct pa_settings*)(ptr);
```

D. Integrating the asynchronous API in knife-alien

Since the asynchronous API demands both initialization and destroying of objects, initPulseaudio() and destroyPulseaudio() are added to main_GUI_OpeningFcn() and closeFcn() respectively. The memory address returned by initPulseaudio() is saved to the UserData property of the last dummy filter for later use by playAudio() and destroyPulseaudio(). See **??**, **??** and **??** for how PulseAudio is used in knife-alien.

Listing 13. main_GUI.m: Initializing PulseAudio and saving object address

```
    % Connect to pulseaudio deamon
    handles.pa_ptr = initPulseaudio;

    dummy = Filters.DummyFilter;
    firstDummy = Filters.DummyFilter;
    firstDummy.Next = dummy;
    dummy.Prev = firstDummy;
    set(firstDummy,'StemHandle',stemHandle);
    set(dummy,'StemHandle',stemHandle2);
    set(firstDummy,'Fs',fs);
    set(dummy,'Fs',fs);
    set(dummy,'UserData',handles.pa_ptr);
```

Listing 14. closeFcn.m: Destroying PulseAudio connection

```
destroyPulseaudio(handles.pa_ptr);
```

Miscellaneous

All C files were compiled as shown in **??**.

Listing 15. How files were compiled

```
mex -lpulse <file>
```

Many lines of code has been truncated for readability. Please refer to http://0pointer.de/lennart/projects/pulseaudio/doxygen/ for the complete PulseAudio API documentation.

III. Conclusion and further enhancements

The implementation of PulseAudio in knife-alien presented in this report is not efficient enough to produce audio output that is smooth and with as little latency as possible.

A non-blocking approach of calling PulseAudio routines is needed to avoid waiting for the sound to be played before returning control to MATLAB. Possibilities include threading, choosing a different PulseAudio event loop implementation or playing the sound in a different process completely.

Appendix

Using PulseAudio with Simulink

SIMULINK uses a different function layout than MATLAB. In MATLAB, a single function is defined, whilst in SIMULINK, a functional block is defined. For this block a number of input and output ports need to be defined, as well as any additional input parameters.

Since SIMULINK is a iterative simulation tool it defines functions that will be run at the start, during and end of a simulation. This makes it possible to combine the source code of initPulseaudio.c, playAudio.c and destroyPulseaudio.c into one single file – playAudio_s.c.

?? display how the number of input ports are set. It also sets the expected type of input data and forces the data to be continuous. If the data is not explicitly set to be continuous then the code in **??** would not function.

Likewise the number of output ports are set on line 145

Listing 16. playAudio_s.c: Defining input ports

```
133     if (!ssSetNumInputPorts(S, 1)) return;
134     ssSetInputPortWidth(S, 0, 1);
135     ssSetInputPortRequiredContiguous(S, 0, true); /*direct input signal access*/
136     ssSetInputPortDataType(S, 0, SS_INT16);
137     /*
138      * Set direct feedthrough flag (1=yes, 0=no).
139      * A port has direct feedthrough if the input is used in either
140      * the mdlOutputs or mdlGetTimeOfNextVarHit functions.
141      * See matlabroot/simulink/src/sfuntmpl_directfeed.txt.
142      */
143     ssSetInputPortDirectFeedThrough(S, 0, 0);
144
145     if (!ssSetNumOutputPorts(S, 0)) return;
```

The initialization process earlier contained in initPulseaudio.c is now confined in the function mdlStart(), as shown in **??**. This function will only run once at the start of a simulation.

Listing 17. playAudio_s.c: Initializing PulseAudio

```
#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
  /* Function: mdlStart =======================================================
   * Abstract:
   *    This function is called once at start of model execution. If you
   *    have states that should be initialized once, this is the place
   *    to do it.
   */
  static void mdlStart(SimStruct *S)
  {
    // We'll need these state variables to keep track of our requests
    int pa_ready = 0;
    int pa_conn = 0;
    int fork_id = 0;

    // Create a mainloop API and connection to the default server
    pa_ptrs.pa_ss.format = PA_SAMPLE_S16NE;
    pa_ptrs.pa_ss.rate = 22050;
    pa_ptrs.pa_ss.channels = 1;
    pa_ptrs.pa_ml = pa_mainloop_new();
    pa_ptrs.pa_mlapi = pa_mainloop_get_api(pa_ptrs.pa_ml);
    pa_ptrs.pa_ctx = pa_context_new(pa_ptrs.pa_mlapi, "test");

    // This function connects to the pulse server
    pa_context_connect(pa_ptrs.pa_ctx, NULL, 0, NULL);

    // This function defines a callback so the server will tell us it's state.
    // Our callback will wait for the state to be ready.  The callback will
    // modify the variable to 1 so we know when we have a connection and it's
    // ready.
    // If there's an error, the callback will set pa_ready to 2
    pa_context_set_state_callback(pa_ptrs.pa_ctx, pa_state_cb, &pa_ready);

    for (int i=0;pa_ready == 0 && i<1000;i++) {
        pa_mainloop_iterate(pa_ptrs.pa_ml, 1, NULL);
    }

    if (pa_ready == 2 || pa_ready == 0) {
            pa_context_disconnect(pa_ptrs.pa_ctx);
            pa_context_unref(pa_ptrs.pa_ctx);
            pa_mainloop_free(pa_ptrs.pa_ml);
            ssPrintf("Error");
            return;
    }

    // At this point, we're connected to the server and ready to make
        // requests
    ssPrintf("Context_connected_to_PA-daemon\n");


    pa_ptrs.pa_s = pa_stream_new(pa_ptrs.pa_ctx,"FooStream",&pa_ptrs.pa_ss,NULL);
    pa_stream_connect_playback(pa_ptrs.pa_s, // The stream to connect to a sink
            NULL, // Name of the sink to connect to, or NULL for default
```

```
            NULL,  // Buffering attributes, or NULL for default
            PA_STREAM_NOFLAGS,  // Additional flags, or 0 for default
            NULL,  // Initial volume, or NULL for default
            NULL  // Synchronize this stream with the specified one, or NULL for a
                    standalone stream
            );
    pa_stream_set_state_callback(pa_ptrs.pa_s,pa_stream_cb,&pa_conn);

    for (int i=0;pa_conn == 0 && i<1000;i++) {
        pa_mainloop_iterate(pa_ptrs.pa_ml,1,NULL);

    }

    if (pa_conn == 2 || pa_conn == 0) {
            pa_stream_disconnect(pa_ptrs.pa_s);
            pa_context_disconnect(pa_ptrs.pa_ctx);
            pa_context_unref(pa_ptrs.pa_ctx);
            pa_mainloop_free(pa_ptrs.pa_ml);
            ssPrintf("Error");
            return;
    }

    ssPrintf("Stream connected to PA-daemon\n");
}
#endif /*  MDL_START */
```

For each iteration of a simulation, mdlUpdate() is called. That is where the code from playAudio.c is placed.

Listing 18. playAudio_s.c: Retrieving and playing audio data

```
#define MDL_UPDATE  /* Change to #undef to remove function */
#if defined(MDL_UPDATE)
  /* Function: mdlUpdate ======================================================
   * Abstract:
   *    This function is called once for every major integration time step.
   *    Discrete states are typically updated here, but this function is useful
   *    for performing any tasks that should only take place once per
   *    integration step.
   */
  static void mdlUpdate(SimStruct *S, int_T tid)
  {
    // Declare some variables used later
    pa_operation *o;
    size_t writableSize;

    // Get data
    if (ssGetInputPortDataType(S,0) != SS_INT16) {
        ssPrintf("Wrong input type");
        return;
    }
    const int16_t *data = ssGetInputPortSignal(S,0);
    size_t r=ssGetInputPortWidth(S,0);

    // Determine how much we can put in buffer
    writableSize=pa_stream_writable_size(pa_ptrs.pa_s);
    if (writableSize < r)
        r=writableSize;
    // Play
    pa_stream_write(pa_ptrs.pa_s,data,r,NULL,0,PA_SEEK_RELATIVE);
    o=pa_stream_drain(pa_ptrs.pa_s,0,NULL);
    while(pa_operation_get_state(o) != PA_OPERATION_DONE) {
        pa_mainloop_iterate(pa_ptrs.pa_ml,1,NULL);
    }
    pa_operation_unref(o);
  }
#endif /* MDL_UPDATE */
```

When a simulation ends, the mdlTerminate() function is called and all PulseAudio objects are destroyed. See **??** for the code.

Listing 19. playAudio_s.c: Destroy and disconnect stream and context

```
/* Function: mdlTerminate =====================================================
 * Abstract:
 *    In this function, you should perform any actions that are necessary
 *    at the termination of a simulation.  For example, if memory was
 *    allocated in mdlStart, this is the place to free it.
 */
static void mdlTerminate(SimStruct *S)
{
    pa_stream_disconnect(pa_ptrs.pa_s);
    pa_context_disconnect(pa_ptrs.pa_ctx);
    pa_context_unref(pa_ptrs.pa_ctx);
    pa_mainloop_free(pa_ptrs.pa_ml);
    mexPrintf("Pulseaudio destroyed\n");
}
```

This code is now ready to be used with SIMULINK. It is compiled according to **??**.

To use this code in a SIMULINK simulation, the SIMULINK block 'S-Function' is used. For recording audio the 'From Audio Device' block of the *DSP System Toolbox* was used. That block outputs audio data in the form of SIMULINK frames so an 'Unbuffer' block has to be used to serialize the data. A 'Reshape' block is used to remove empty dimensions.

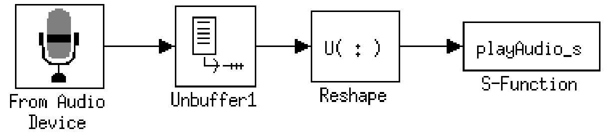

Fig. 2. SIMULINK block diagram

The 'From Audio Device' block need to be configured according to **??** and in the 'S-Function' block, 'S-function name' is set to *playAudio_s*.

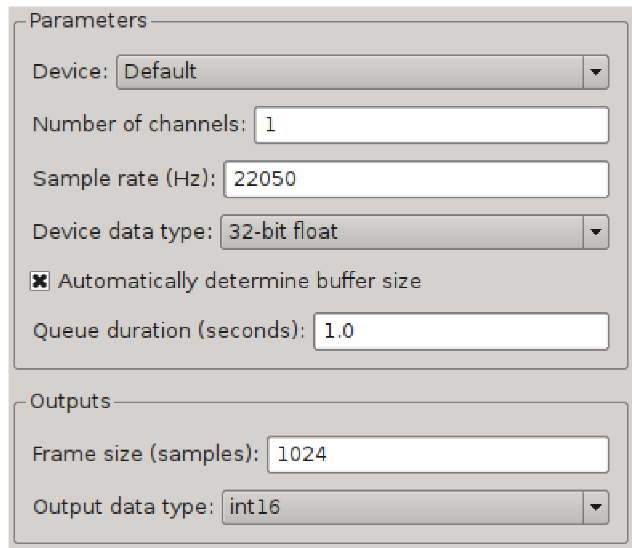

Fig. 3. Parameters of 'From Audio Device'

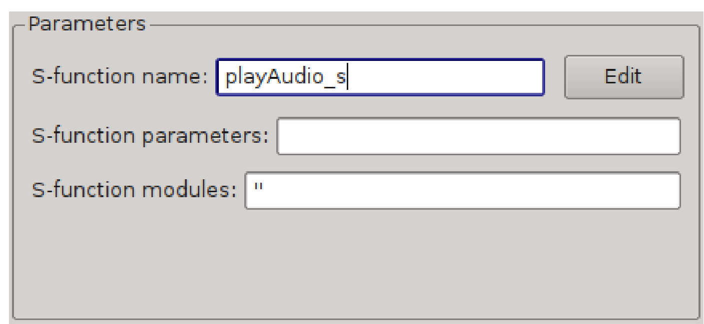

Fig. 4. Parameters of 'S-Function'