Real-Time Protocol-based Audio Engine

Michał Konrad Rój

Real-Time Protocol-based Audio Engine

by Michał Konrad Rój

Abstract

This work describes a so-called *Real-Time Protocol-based Audio Engine* which is a set of procedures and threads responsible for proper real-time sound processing and sending it through the network. An H.323 terminal, capable to send and receive sounds, must meet many requirements. First, it must be able to process a sound signal in real-time. Then, it must use a standardized set of audio codecs to be correctly understood by the remote terminal (an H.323-compliant terminal it communicates with). Finally, it must send and receive the signal in a proper Real-Time Protocol (RTP) format. From the user's point of view, it should have simple, though parametrized, API. This would allow the user to create sound streams flexibly, depending on particular needs.

This work covers two fundamental aspects of the *Real-Time Protocol-based Audio Engine*: *Capturing and Playing Module*, responsible for proper real-time sound processing and *Audio Codec Module* responsible for encoding and decoding audio streams, as well as managing available audio codecs. The modules are flexible (e.g. ready to be expanded by new audio codecs). The modules include interfaces needed to exchange information with the other modules. The *Audio Codec Module* includes an G.711 codec, what makes this part of the terminal fully usable.

Notice that the sound equipment is beyond the scope of *ITU-T Rec. H.323*. The Recommendation specifies neither the algorithms how the equipment should work nor suggested mechanisms to be used. Consequently, the great parts of this work are the design of the author of this work.

Table of Contents

List of Figures

List of Examples

Chapter 1. Introduction

1.1. The goal of this work

The goal of this work is to create a *Real-Time Protocol-based Audio Engine* software module (abbr. *Audio Engine*).The Audio Engine module is a set of procedures responsible for handling real-time audio at the transmitter and receiver side. The Audio Engine at the transmitter delivers sound capturing, proper sound encoding, proper preparing real-time protocol stream, and finally, sending the audio stream to the network. The responsibility of the Audio Engine at the receiver is to acquire the audio stream properly, decode it, and then play the sound properly. All these procedures are working as the real-time procedures and their main goal in deliver a good-quality, countinuous audio signal, with minimal possible delay.

The software modules, described here are prepared to work with Microsoft Windows operating systems. But the way they were written, makes it possible to port the modules to another platform. Second, these software module were designed to be used as:

- A library with a strictly defined API.
- A simple standalone application that would be able to transmit speech from one PC to another.
- A component of an H.323 terminal, which had a simple and portable API.

This work covers two from the three submodules of the Audio Engine module: *Capturing and Playing Module*, responsible for proper real-time sound processing and *Audio Codec Module*, responsible for proper encoding and decoding sounds. The *RTP Module*, the third module, is not in the scope of this work.

1.2. PC Audio Tutorial

1.2.1. Sound devices

Sound device is an electronic gadget, commonly part of a personal computer, in charge of all jobs related to the sound. Sound devices can usually record and play sounds. When sound device can perform recording and playing sound simultaneously, it is called ’full-duplex’. Input devices

(e.g. microphone) and output ones (e.g. loudspeaker) are connected directly to a sound device. These devices work on analog signals. As a result of this fact, sound device must also be a medium between analog device and digital computer system. It converts the outgoing signal from digital form into analog one, and converts analog incoming signal into digital form, so it can be stored in computer memory. PC sound devices may work in various modes (coding styles, sampling rates, etc). They are controlled by operating system (have assigned I/O port and interrupt number), and usually communicate by the use of direct memory access.

Almost all contemporary operating systems deliver mechanisms of controlling sound devices over high-level programming interface. No knowledge about internals of sound card (device) are needed. OS driver does all low-level tasks for a programmer. For example multimedia unices, like Sun-OS or Linux, support `open-read-write` mechanisms on sound devices that are the part of the input-output Unix API [12].

There are plenty of sound libraries, developed for Microsoft Windows. Windows 95 and 98 seem to be the most popular multimedia platforms. Microsoft worked out at least three sound interfaces, which are commonly used.

1.2.2. PCM: Sampling and Quantization

Advantages of using digital audio are well-known: it can be stored, processed, duplicated and transmitted in a simple way without losing its quality. Like any other computer data, it can be transmitted as an ordinary data in packet-based computer networks with no additional features.

There are many ways of storing audio signal in digital form. The most popular one is Pulse Code Modulation (PCM). Signal stored in PCM have structure of a matrix (or a table). Every column of the matrix is called a sample. Every row is called a quantization level. Quantization levels are measured in bits. For example: if we say that there is a 16-bit quantization, we mean that there are 2^16 (2 to the power 16) quantization levels. We can also call it 16-bit samples, referencing to every sample in the matrix. The number of samples per second is a sampling rate, e.g. if we have 2-second signal stored in 10000 samples, we call it 5000 Hz (or 5kHz) sampling rate. The higher sampling rate and quantization levels the better quality and dynamic range of the sound. For CD-Audio stored in PCM form (16-bit quantization, 44.1 kHz sampling rate, stereo) we need almost 180 kB to store one second of sound. So the typical values in telephony are 8-bit quantization and 8 kHz sampling rate, generating 64 kbit/s bit-stream. The process of transforming audio analog signal into digital form (and vice-versa) is made by sound devices. Description of the process is beyond the scope of this work. For additional information, refer to [2].

1.2.3. Audio Codecs

Simply speaking coding is a way of transforming a signal from one form into another. Decoding is an opposite transformation to coding i.e., encoded form of the signal is transformed into the original (or similar to) original signal. The items that encode or decode signals (in hardware or software way) are called coders or decoders respectively. Codecs are all-in-one items that make both coding and decoding job. Audio codecs are items that work on audio signal. According to the definition above the transformation from electric signal to digital audio is also coding but is this chapter, term “coding” refers to transformation from one digital form into another.

There are many different reasons for using audio codecs: to make the signal less sensitive to interferences, to allow transporting in various networks, to encrypt the signal, etc. But the most important reason is saving the bandwidth. When recording with 8000 sampling rate and 16-bit quantization, 128 kbit/s bit-stream is generated. This requires quite a large bandwidth even for the commonest 10 Mbit/s LAN. Codecs which decrease a number of bits per second work like compressing programs. They get a block of samples (N bytes) and transform it into M-bytes (M is less N) block. Most of audio codecs are lossy ones because human ear cannot detect slight differences between genuine and decoded signal.

Codecs used in multimedia terminals must be standardized to be able to communicate with other terminals, which have been prepared by another producer. There are many standards defined by ITU-T recommendations and other international organizations.

For more information about audio coding refer [1], [2].

1.2.4. Audio in packet-based networks

Packet networks grow more and more popular in modern telecommunication. The idea of a packet network is that the endpoints are not connected directly (like for instance in case of telephone networks). Endpoints send their data in so-called packets (containing the address of the receiver in their header field, and small quantity of data). Every packet is carried separately over the transmission medium. This is quite an economic solution (no all the trail is reserved for the purposes of one particular connection) but it causes also some problems: packets may be exessively long stored by the network’s nodes (routers). Sometimes the packets may be lost (e.g. because the overflow of a router’s buffers). This is all the result of the fact that the most of packet-based networks (esp. IP networks) were designed for data transmitting purposes. Usually, data transmission does not require real-time mechanisms, the higher emphasis is set on reliability of the transmission media and protocols. The problem is lost packets was solved by developing reliable protocols (at the cost of delay). But the delay problem cannot be solved in a simple way. In some types of networks in cannot be solved at all.

Audio in packet based networks is transmitted in packets like an ordinary data. A frame (some amount of time long) is stored in a buffer and then it is transmitted into the network. The receiver of the audio packet stream must be ready for these two events:

- Packet may be lost; in this case receiver should make some procedures to supply the continuous audio signal.
- Packet may be delayed (jitter); to prevent it, special buffers are prepared at the receiver side. If the delay is too big these buffers may be also not enough.

The aspects above are more thoroughly described in [2] and [10].

1.3. Audio in H.323 Terminal

Unlike other H.323 entities, H.323 terminal is a really multimedia entity. I can support audio, video and data (chat-like and/or ftp-like) capabilities. Audio capabilities are the essential part of such a terminal. First, it allows multimedia computer to become a telephone-like device, device that mankind has been used to for decades. Second, this device is extremely cheap while using: usually it is a “Voice Over IP” device, so it requires just access to Internet to find itself in the global network.

According to [7] audio capabilities *must* be present in every H.323-compliant terminal. Video and data capabilities are optional. Thus, this terminal, as well as any other H.323-compliant terminal, supports audio.

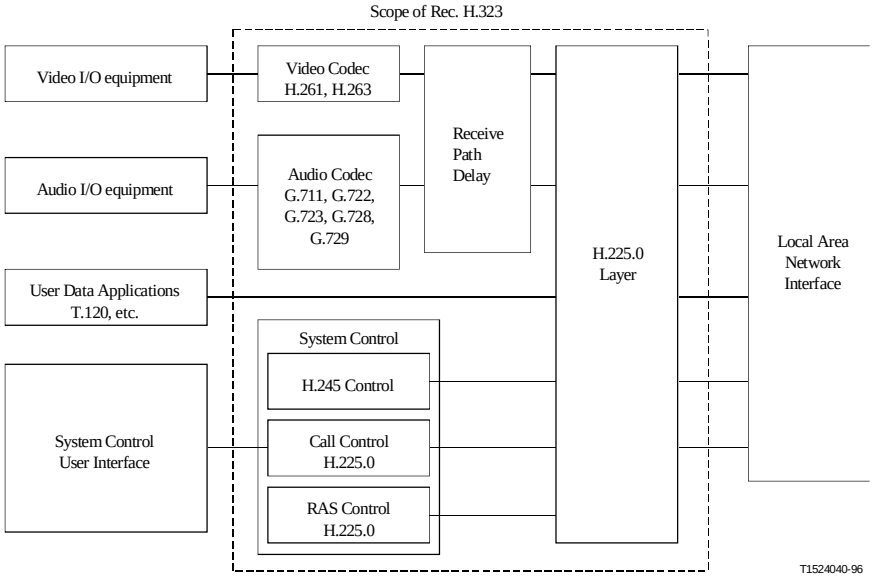

Figure 1-1. Inside the H.323 Terminal

Figure 1-1 (taken from the *ITU-T Rec. H.323*) shows the H.323 terminal and where is every functional block placed. The input audio stream starts in a microphone, then it is handled by the operation system, and then is processed by audio capturing module. These parts are outside the terminal and are represented by the *Audio I/O Equipment* in the figure. Then, audio stream must be encoded using a specific audio coder, and the encoded signal could be passed to the *Real-Time Protocol (RTP) Block* (H.225.0 Layer). The module prepares the frames and sends them to the network.

The output audio stream is a functional inversion of the input stream.

Although Audio I/O Equipment is a very system-dependent module, data streams generated by the the module must fit to the input of audio coders. Coders demand of Audio I/O Equipment to supply them with audio signal continuously, packed in frames. This signal is usually 8 or 16-bit, 8 or 16 kHz PCM. Such a signal must be generated by the sound device. Typically both participants are able to send and receive audio data, so sound device must be full-duplex. If not, the multimedia terminal can work just as a transmitter or by a receiver, never both of them. For some codec sets, playing sampling rate may differ from recording sampling rate, but not all sound devices support this capability. Consequently, before any audio capabilities are used (especially sent to the remote terminal by use of the control channel), the local sound system must be thoroughly examined.

For H.323 terminal [7] the following codecs are defined:

- G.711-based set of codecs. *This set of codecs is mandatory in every H.323 terminal.*
- G.722 codec.
- G.728
- G.729
- MPEG 1 audio
- G.723.1

The Recommendation allows to use the other codecs (not specified in the Recommendation) as well. They are must be also negotiated in the control channel as the “non-standard audio capability”.

H.323 terminals must be able to encode and decode audio signals using G.711 codec family, both A-law and U-law. Other audio codecs are optional, although recommended for some types of networks. So the minimal H.323-compliant terminal would be a G.711-audio terminal.

The H.323 requires to treat all the audio codecs as frame-oriented codecs (even those sample-oriented). If a codec is sample-oriented a frame is produced (by collecting some number of samples). The length of the frame must be 8 octets long. Many frames can be sent in one packet but one frame cannot be divided into many frames (for details refer [5], p. 12).

A receiver must be able to acquire 200 ms of sound in a single packet (this condition is introduced to set an appropriate receive buffer lenght).

If no sound is received for some time, receiver should repeat the latest received audio frame with a lower volume.

1.4. Windows Audio Mechanisms

The contemporary Microsoft Windows operating systems can be called “multimedia” systems. They support many sound and video features which now are standard parts of the OS. Among numerous sound interfaces available, there are three well-documented, having many examples, and freely distributed interfaces from Microsoft: Direct Sound, Multimedia Control Interface (MCI), and WaveForm Audio.

Direct Sound is a component of Microsoft DirectX - the set of object multimedia libraries. DirectX consist of several components used widely in modern computer games, graphical applications and multimedia programs. Direct Sound has many advanced capabilities including multichannel mixing and 3D sound. This system could be used successfully in multimedia terminal, but it has a few disadvantages that made the author give up:

- It is object library, so it requires object-oriented coding style in sound modules and C++ compiler.
- DirectX is not a standard element of early Windows NT and Windows 95 versions. If DirecX has not been installed in such systems, DirectX-dependent programs would not work.
- The latest version of DirectX for Windows NT does not support capturing (recording) of sound. Other mechanisms must be used to make full-duplex programs.

Multimedia Control Interface (MCI) is a standard component of Windows systems. It is very simple in use: programmer calls functions with char* parameter, which is a shell-like command. This is the only advantage of making use of this mechanism in multimedia terminal. It neither supports real-time continuous recoding nor playing, and it seems to be slower than system calls, because of parsing commands. It made the author give up.

In the author's opinion the most suitable mechanism in this case is the *Waveform Audio* mechanism, described in details in the next chapter.

1.5. Microsoft Windows *Waveform Audio*

The exhaustive information about WaveForm Audio can be found in [8]. Windows API mechanisms used here are described in [11].

WaveForm Audio is a standard member of Windows and Windows NT/2000. Even though all the routines are the components of a separate library (`winmm.dll`), these function set may be treated as the "sound system call set". As a basic set of functions, the module requires filling large structures. Consequently, the simplest program using this mechanisms would have more than fifty lines, but it gives programmer almost full control over sound devices. The control, that is portable among all the sound devices having drivers for the OS. *WaveForm Audio* was chosen by the author to be the engine of multimedia terminal's *Audio Engine*.

1.5.1. *WaveForm Audio* Overview

WaveForm Audio is the set of functions as well as data structures. To manage sound devices over *WaveForm Audio* one can use a kind of virtual device called waveform-audio device (further called just a "device"). There are two types of such devices: input device used while recording, and output device used while playing sounds. These devices are treated independently. Every device is represented by a special handle: of the type HWAVEOUT for the output device, and of the type

HWAVEIN for the input device. The variable of these types are used to identity the sound devices. The full list of Waveform functions and the example how they are used is in Appendix B.

WaveForm Audio contains mechanisms that should be used as the body of initial procedures. They allow to check if sound devices are present in the system, and to find out how many of them may be used. Programmer can get their capabilities and choose one of them. If none of them can be used, further procedures could not be performed.

Before usage, waveform-audio device must be opened. `waveInOpen` and `waveOutOpen` functions are needed to open input and output device. Parameters such as sampling rate, number of channels, bits per sample, the type of encoding, etc. are specified when calling these functions. The other parameter passed to the functions is the type of signallization between OS the sound functions. It can be:

Event mechanism
Callback mechanism
Thread Messages mechanism
Windows Messages mechanism
No signallization

If something goes wrong calling the functions (no free memory, bad parameters, not supported capabilities, ..) functions fail. Otherwise, the proper handle is returned and device can be controlled over this handle. The type of the returned handle is HWAVEIN for `waveInOpen` and HWAVEOUT for `waveOutOpen`.

1.5.2. Capturing

Capturing is organized as follows:

- Special headers are prepared (`waveInPrepareHeader`). This header contains a buffer (later filled with samples) which is prepared by user (e.g. by using `malloc`). Header contains also the buffer length and additional parameters.
- Prepared header is added to the system buffers (`waveInAddBuffer`). There can be many system buffers.
- Function `waveInStart` is called to start recording. In this exact moment OS gets first buffer from the set of system buffers and begins recording. When filling of the buffer is finished OS begins recording to the next buffer and so forth until all the system buffers are filled. After every buffer filled system also sends a signal to the application that buffer has been filled. Next buffer may be added to the system (`waveInAddBuffer`) to get continuous signal or another procedures, like copying or encoding may be performed after this signal.

When in the capturing loop all headers are already prepared. So just `waveInAddBuffer` must be called to add the current buffer to the system buffer set.

The fragment of the capturing (recording) procedures is illustrated in Figure 1-2. In this version *Event* mechanism is used.

```
int i=0, No_SYSTEM_BUFFFERS;

if (waveInOpen(&hwi,
  0, (LPWAVEFORMATEX)&pwf,
        (DWORD) rip->eventh, 0, CALLBACK_EVENT )) {
 /* ... error handling  ... */
}

/* ... Preparing system buffers (sb) ... */

for(i=0; i<No_SYSTEM_BUFFERS; i++){

    sb[i]->lpData = (LPBYTE) malloc(system_buf_len);
    sb[i]->dwBufferLength = system_buf_len;
    sb[i]->dwBytesRecorded = 0;
    sb[i]->dwUser = 0;
    sb[i]->dwFlags = 0;
    sb[i]->dwLoops = 0;

 if(!sb[i]->lpData)
  /* ... error handling  ... */

        if (waveInPrepareHeader(hwi, sb[i],
  sizeof(WAVEHDR))) {
 /* ... error handling  ... */
        }
        if (waveInAddBuffer(hwi, sb[i],
  sizeof(WAVEHDR))) {
 /* ... error handling  ... */
 }
}

waveInStart(hwi);
ResetEvent(eventh);

while(1)
{
```

```
 WaitForSingleObject(eventh, INFINITE);
  /* ... sb[i] processing ... */
        if (ret = waveInAddBuffer(hwi, sb[i],
  sizeof(WAVEHDR))) {
  /* ... error handling ... */
 }

 if(i == No_SYSTEM_BUFFFERS - 1)
     i=0;
 else
     i++;
 if(stop_thread_flag)
     break;
}
```

Figure 1-2. The C-code of the capturing loop.

1.5.3. Playing

Playing can be organized in simpler way:

- Headers must also be prepared (`waveOutPrepareHeader`). Similar parameters are specified.
- Buffers (field of the header structure) are filled with blocks of samples.
- `waveOutPause` is called to tell OS that we do not want to start playing immediately after first `waveOutWrite`.
- `waveOutWrite` is called with the parameter pointing to buffer we would like to play. We can call this function more than once to supply continuous playing.
- `waveOutRestart` must be called to start playing.

The fragment of the playing loop is illustrated in Figure 1-3. In this version *Event* mechanism is used.

```
int i=0, No_SYSTEM_BUFFFERS, BUFFER_LEN;

if (waveOutOpen(&hwo,
  0 /*default*/, (LPWAVEFORMATEX)&pwf,
        (DWORD) eventh, 0, CALLBACK_EVENT )) {
 /* ... error handling */
}

/* .... allocation of sb buffers .... */

for(i=0; i<NoSYSTEM_BUFFERS; i++){ /*  header preparation  */

   sb[i]->lpData = (LPBYTE) malloc(BUFFER_LEN);
   sb[i]->dwBufferLength = system_buf_len;
   sb[i]->dwBytesRecorded = 0;
   sb[i]->dwUser = 0;
   sb[i]->dwFlags = 0;
   sb[i]->dwLoops = 1;

   if(!sb[i]->lpData)
 /* ... error handling  ... */

   if (waveOutPrepareHeader(rip->hwo,
 sb[i], sizeof(WAVEHDR))) {
 /* ... error handling */
   }
}

while(1)
{
/* ... process previous sound block ... */
/* ... prepare lacking frames if needed */

if (waveOutWrite(hwo, sb[i], sizeof(WAVEHDR))) {
 /* ... error handling ... */
}

if(i == No_SYSTEM_BUFFFERS-1)
    i=0;
else
    i++;

if(stop_thread_flag)
```

```
 break;

WaitForSingleObject(risp->eventh, INFINITE);
}
```

Figure 1-3. The C-code of the playing loop

1.5.4. Shutting Down

When we want to stop playing or recording we must:

- Call `waveOutReset` or `waveInReset` to free all system buffers (idle or being filled in the moment).
- Call `waveOutClose` or `waveInClose` to tell the operating system that devices are no longer needed. Other processes can access sound devices after this call.
- Call `waveOutUnprepareHeader` or `waveInUnprepareHeader` for every previously prepared header to unprepare them. After this call all data allocated for the use of buffers may be freed.

Chapter 2. Implementation

This chapter describes how Audio Modules are implemented in this project. The description includes *Capturing and Playing Module* for 32-bit Windows system, *Audio Codec Module* which claims to be portable, *Audio Queue Module*, the module used to exchange the audio frames between Audio and RTP modules (portable) and finally, the G.711-based codec implementation description (portable, too).

Audio Modules are very system-dependent modules. They must be designed and prepared taking to consideration the capabilities of the current OS and used audio library. If one is going to prepare the versions for the other operating systems and/or the other sound libraries, one should re-design and rewrite this module completely. This version of the *Audio Engine* is designed for Windows 95, 98, NT, and 2000 operating systems (NT/2000 are suggested) and based on the rules described in Section 2.1. The audio mechanism used by the project is *Waveform Audio*.

2.1. *Audio Engine* Architecture

Capturing and Playing Module is a part of an application (e.g. multimedia terminal), responsible for a proper capturing and playing real-time sounds. There are plenty of audio mechanisms prepared for various operating systems. But just a small subset of them is ready to be used in real-time applications. The real-time audio programming and the full-duplex programming requires some specific approach to the matter. Occasionally, programmer must make the most of the available device and the used programming libraries to prepare such a program. This chapter explains how to design a real-time application module.

2.1.1. Audio Streams

In multimedia terminal the two type of audio streams can be separated: an input stream and an

output stream. They should be treated independently.

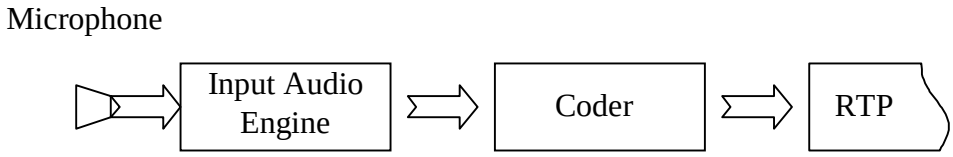

Figure 2-1. Input Audio Stream

In case of the input stream, as showed at Figure 2-1, the sound is taken from the microphone, sampled and prepared by the *Input Audio Engine*. Next, it is encoded and sent to the Real Time Protocol (RTP) process. The RTP process puts the encoded sound (payload) into the special frame and sends it to the network.

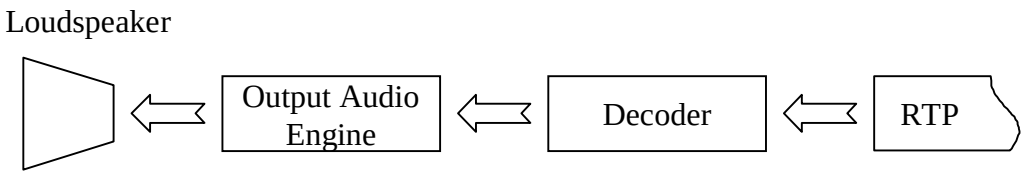

Figure 2-2. Output Audio Stream

In case of the output sound (Figure 2-2), the sound payload is taken from the RTP process, then decoded and sent to *Output Audio Engine*, responsible for continuous playing of the audio signal.

The RTP module is beyond the scope of this work. Coder Modules and Decoder Modules are very simple from the user point of view. They are usually procedures that get a block of data and return the other (“encoded” or “decoded”) data block. The remaining work is done by the Capturing and Playing Module. Luckily, low-leveled sampling is performed by OS and sound device. It is the Capturing and Playing Module that supports real-time capturing and playing of the sounds. It is Capturing and Playing Module that examines sound devices and their capabilities. Finally, it is Capturing and Playing Module that fills the empty space when sound frames are not received from the network.

2.1.2. Capturing Loop

Very popular scenario of real-time sound processing are “capturing loop” and “playing loop”. Any loop is independent from the other loops. They may be organized as separate threads or even the separate processes. To simplify the matter, the capturing, encoding, decoding and playing procedures are called further threads.

```
while(EXIT_CONDITION == FALSE)
{
 TABLE_OF_SAMPLES = get N samples from ’INPUT SOUND DEVICE’;
 pass TABLE_OF_SAMPLES to ’CODER PROCEDURE’;
}
```

Figure 2-3. Capturing Thread

As shows the pseudo-code at Figure 2-3, the procedure just takes N number of samples from the device (using a given API) and passes it it to the audio coder procedure. The encoding thread can also be organized as a loop (as shown in Figure 2-4). The reason for separating the encoding thread was the time of encoding procedure: it takes some time, so it should not be called from inside the capturing thread (as organized in Figure 2-3). Looking at Figure 2-1 it would appear that audio coders operate on the flowing audio stream and generate the flowing one. But audio coders use a block of samples as their input and generate a bit-stream block. The simplest encoding function would have the prototype: `encode(void *input_data, void *output_data);`

```
while(EXIT_CONDITION == FALSE)
{
 SAMPLES = get TABLE_OF_SAMPLES from ’INPUT AUDIO ENGINE’;
 ENCODED_FRAME = encode SAMPLES;
 pass ENCODED_FRAME to ’RTP PROCEDURE’;
}
```

Figure 2-4. Encoding Thread

Unfortunately, the illustrated methods do not support the continuous signal. The capturing thread spends some time with passing the data to the coder procedure. Then it starts sampling next block (by calling “`get N samples from`”). Occasionally, the capturing thread can be suspended by the operation system after passing samples to the encoding thread but before starting sampling the next block. This would cause the unintentional break in the capturing. The designer of the module cannot afford to such a situation, so the above algorithm should be changed.

First, the next block should be started immediately after the previous one. This feature must be supplied by the operating system (OS). Windows 9x/NT systems go with such a feature. Before starting sampling, the user sends to the OS a number of buffers. Then, the sampling is started. The system fills every buffer and after completing the current one it begins filling the next buffer. The system sends also the signal to the user that the sampling of a certain data has been completed. The user can process it, and after all this work he adds the buffer to the system. Consequently, it can be filled again, and again.

Now, there is no need to separate the encoding thread from the capturing thread. In case of the contemporary personal computers, sampling takes much more time than encoding of the input block. This would save the designer from programming the inter-thread communication for capturing and encoding threads. This would also simplify the control over the threads. The new (safe) procedure is shown in Figure 2-5.

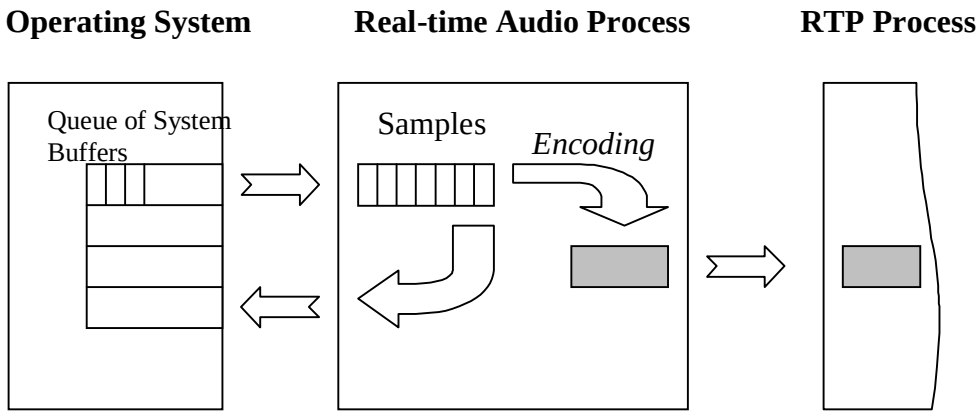

Figure 2-5. Safe Capturing Loop

The improved algorithm (with included encoding) would look like at Figure 2-6.

```
for (i = 1; i < NUMBER_OF_SYSTEM_BUFFERS; i++)
{
 put BUFFER[i] to 'OPERATING SYSTEM';
}
```

```
start capturing to buffers in 'OPERATING SYSTEM';

while(EXIT_CONDITION == FALSE)
{
 wait for the fill-up of the current block;
 SAMPLE_TABLE = get the recently filled buffer from 'OPERATING SYSTEM';
 ENCODED_TABLE = encode SAMPLE_TABLE;
 pass ENCODED_TABLE to 'RTP PROCEDURE';
 put SAMPLE_TABLE to 'OPERATING SYSTEM';
}
```

Figure 2-6. Improved Capturing Loop

As shown in Figure 2-6, the first phase of the capturing thread is the putting `NUMBER_OF_SYSTEM_BUFFERS` buffers to the operating system. This number should be at least two (to deliver the continuous audio signal), but three or more would be the good choice in some computer systems. The buffers are represented by structures (in C sense) containing the pointer to the memory allocated by the user. It contains also the length of the memory and some additional flags. A number of these flags are used by the sound library itself. The others may be used by the user (programmer). For instance one of them can be used to name a given buffer (set a unique value), and the other flag should be set to FALSE by the user, and system sets it to TRUE when the buffer has been filled. Using this system the user can recognize whether more than one buffer has been filled in the time of going through the capturing loop body (it may happen if the OS is full of busy processes).

In this implementation capturing thread is organized as separate thread. The threads is called by `start_recorder` routine that initiates data structures, especially headers (using `waveIn-PrepareHeader`), adds them to system buffers, runs `recorder_thread` as a new thread and returns a handle of the type snd_thread_id_ptr. By this handle user can control recording thread, especially kill it if no longer required (using `stop_recorder` procedure).

Procedure `recorder_thread` starts sampling and goes to the main capturing loop. At the beginning of the loop thread is being blocked. It is system who unblocks recording thread when a block of samples has been recorded. Then, recorder encodes block of samples using a specified audio coder. Encoded frame is put to queue owing by RTP thread. Empty header is thrown to the system (`waveInAddBuffer`) and the thread looks whether the stop condition is true (this makes thread call reset, stop and close audio device and free all allocated data). If not, thread jumps to the beginning of the loop i.e. it blocks itself. And so forth.

2.1.3. Playing Loop

Playing must be organized in a different way. First, it must support a buffer of frames ready to be played (an anti-jitter buffer). Second, it should be occasionally ready for fill-up of the lacking frames. The pseudo-code of such a procedure goes below:

```
while('ANTI-JITTER BUFFER' is not fully filled)
{
 ENCODED_TABLE = get from 'RTP PROCEDURE';
 SAMPLE_TABLE = decode ENCODED_TABLE;
 put SAMPLE_TABLE into 'ANTI-JITTER BUFFER';
}

start playing from 'ANTI-JITTER BUFFER';

while(EXIT CONDITION == FALSE)
{
 wait for the the current buffer buffer being finished;
 ENCODED_TABLE = get from 'RTP PROCEDURE';
 if (ENCODED_TABLE == EMPTY) then /* no buffer from RTP */
  SAMPLE_TABLE = prepare the virtual frame;
 else
  SAMPLE_TABLE = decode ENCODED_TABLE;
 put ENCODED_TABLE into 'ANTI-JITTER BUFFER');
}
```

Figure 2-7. Playing Thread

Figure 2-7 shows an *Output Audio Engine* procedure. The first part of the procedure stores a number of the incoming frames in the "anti-jitter buffer" (AJB). The number of the frames (or the number of milliseconds of the AJB) is passed as one of the procedure parameters. When the buffer is long enough, the playing is started. Working in a loop, the playing thread acquires the audio frame from the RTP thread (the RTP thread receives the frame directly from the network). If frame could not be acquired, a "virtual frame" is prepared by the special procedures (to supply the continuous audio signal). Otherwise the frame is decoded and passed the AJB. Then it is played from this buffer.

The algorithm described above shows just the main idea of the procedure. Thus it is very simplified here. First, samples should be not played directly from the anti-jitter buffer. Second, preparation of lacking frames is sometimes very complicated process, and usually it is not performed immediately after one delayed frame. Finally, the modern complex algorithms are being developed

to solve the lack of Quality of Service (QoS) in certain packet-based networks. These algorithms are not took into consideration here.

Playing loop is organized as separate thread called *Playing Thread*. Playing Thread is started the `start_player` routine. This function prepares all data structures for playing thread e.g., headers, starts new thread and returns snd_thread_id_ptr. At first playing thread starts waiting for sufficient number of digital audio frames in queue (this number is parametrized value). This is made by blocking supported by the queue between RTP and playing thread. Then the waiting is finished playing thread gets a few frames from the queue, decodes them, sends them to the system (`waveOutWrite`) and runs playing (`waveOutRestart`). Then it blocks itself waiting for finish of playing audio frame. Every frame this thread is unblocked, gets next frame from the queue, decodes it and sends to system. If no frames are available (what means that something wrong must have happen - delay or even loss of RTP packet) than playing thread prepares frame using previous one.

It is clear that capturing and playing procedures should be organized separately. They should be implemented as separate processes or separate threads. The other reason to do so is that these procedures should have the special privileges : they are really real-time procedures, so any delay or stopping of these can cause serious effects. The privileges are very system-dependent and will not be described in this chapter.

2.2. Overview of this software

This software is made up of a set of so-called *modules*. Every module is responsible for selected problems. The modules are written in C programming language as several files. Until now the following modules have been designed:

- *Capturing and Playing Module,* called further **audio_engine**. Files: `audio_engine.h` and `audio_engine.c`.
- *Audio Queue Module,* called further **audio_queue**. Files: `audio_queue.h` and `audio_queue.c`.
- *Codec Management Module,* called further **codec_mngmnt**. Files: `codec_mngmnt.h` and `codec_mngmnt.c`.
- *G.711-Compatible Codec,* called further **g711**. Files: `g711.h` and `g711.c`.
- *RTP Module,* called further **rtp**. Files: `rtp.h` and `rtp.c`.

- *Virtual System Module*, called further **virtual_system**. Files: `virtual_system.h` and `virtual_system.c`.

Dependences between modules are as follows:

- **virtual_system** : Win32 or Linux
- **audio_queue** : **virtual_system**
- **codec_mngmnt** : **virtual_system**
- **g711** : **codec_mngmnt**
- **rtp** : **audio_queue**, **virtual_system**, Win32 (until now)
- **audio_engine** : **audio_queue**, **codec_mngmnt**, Win32

As shown, most of modules are dependent on the Virtual System Module (**virtual_system**), which is dependent on Windows (Win32) or GNU/Linux. Consequently, if the **virtual_system** was more portable (e.g., it would work under Spark on Sun OS), **audio_queue**, **codec_mngmnt** and **g711** would automatically be more portable.

The modules can be compiled under Linux with the **make** program (`Makefile` is included to this software). But as a result of the **audio_engine**'s dependency on Win32, this module would not work. Consequently, in this moment *only Win32 version of these audio modules works properly.*

Until now, no Windows project file (Visual C, Borland C, ..) was included to these software modules. Nevertheless, to compile it under VC++ you should:

- Create a new console project (File/New/Win 32 Console Application).
- Add all the mentioned files to the project and additionally `test_all.c` file (click on File-View bookmark in the Project Workspace window and then right-click on "<project_name> files"/Add Files to Project...).
- Add `winmm.lib` and `wsock32.lib` to the project (Project/Settings/Link and add them in Object/Library Modules dialog).
- Compile the project (F7).

2.3. Communication with RTP : *Audio Queue Module*

The *Capturing and Playing Module* must communicate with the RTP module. Recording thread sends encoded frames to the transmitting RTP, and the playing thread receives frames from the receiving RTP module. The module that allows this communication is the *Audio Queue* module. This module is optimized to store blocks of audio data (samples or audio frames). The queue, when initiated, is located between receiving/transmitting RTP thread and recorder/player thread. There may be many queues created, depending on how many audio engine threads are active. Recording thread and transmitting RTP must get the same queue pointer to work properly, and the same to playing thread and receiving RTP.

The queue contains two sets of buffers: *free* and *used*. Free buffers are those that are empty and can be filled with digital audio data. Used buffers are those containing digital audio data.

The main data structure is the frame_queue structure. This is a structure describing all features of the queue. A pointer to this structure (typedefed as fqp) is commonly used by all functions working with this queue. This pointer is also passed to the playing/recording thread and the RTP thread. There is no need of know the internals of this structure to use it properly.

The buf_struct structure describes a buffer (a single node of the queue).

```
struct buf_struct{

 char *payload;  /* digital audio data (payload) */
 timestp timestamp; /* timestamp */

 bsp next;  /* For internal use only */
};
```

The pointer to this structure is typedefed as bsp and it is returned by the get_buffer() and reserve_buffer() functions (described later).

All queue-functions are multi-thread safe, the data is protected by critical sections. These mechanisms are run with no user's effort, they are placed inside the routines described below.

- `prepare_queue` Creates a new queue object.

```
fqp prepare_queue(int payload_len, int buf_num);
```

This function initiates a queue with *`buf_num`* buffers and *`payload_len`* payload length. All data is allocated it the function body, not while using it. All buffers in queue are designated as "free" buffers. No "used" buffers exist.

- `reserve_buffer` Reserves a single buffer.

```
bsp reserve_buffer(fpq queue);
```

  This function reserves (locks) a single buffer from the set of “free” buffers. It returns a pointer to that buffer. If no buffers are available, this function blocks. A buffer, when reserved can be used only by someone who owes the returned pointer (i.e., the reserving thread). It is not accessible for others.

- `reserve_buffer_nb` A non-blocking version of the `reserve_buffer` function.

```
bsp reserve_buffer_nb(fpq queue);
```

  It returns NULL if no “free” buffers are available.

- `put_buffer` Puts the previously reserved buffer into queue.

```
bsp put_buffer(fpq queue, bsp buffer);
```

  `buffer` is the reserved buffer. This buffer is added to the set of “used” buffers (at the end of the FIFO inside).

- `get_buffer` Returns a buffer from the queue.

```
bsp get_buffer(fpq queue);
```

  This function gets the oldest buffer from the “used” buffers set (the first in the FIFO). If no buffers are available the function blocks.

- `get_buffer_nb` A non-blocking version of the `get_buffer` function.

```
bsp get_buffer_nb(fpq queue);
```

  It returns NULL if no “used” buffers are available.

- `get_buffer_when_notless` Gets a buffer on condition.

```
bsp get_buffer_when_notless(fpq queue, int num);
```

  The `get_buffer_when_notless` function gets a buffer if there are not less “used” buffers than *`num`* in the queue. This function a blocking one.

- `giveback_buffer` Returns a buffer to the queue.

```
bsp giveback_buffer(fpq queue, bsp buffer);
```

This function gives back a buffer returned by `get_` functions if no longer needed. Adds the buffer to the set of “free” buffers.

- `get_buffer_len` Returns the length of a buffer.

```
int  get_payload_len( fqp  queue );
```

Every buffer in the *queue* queue is the same length (specified when preparing the a queue). This function allows to check the length.

- `get_buffer_no` Returns the number of all the buffers in the queue.

```
int  get_buffer_no ( fqp  queue );
```

The returned value is a sum of “used”, “free” and “reserved” buffers.

- `turn_on_sorting` and `turn_off_sorting` Turn sorting on or off, respectively.

```
void  turn_on_sorting ( fqp  queue );
void  turn_off_sorting ( fqp  queue );
```

The `turn_on_sorting` function turns on the sorting of frames (according to the `timestamp` field). The `turn_off_sorting` function turns the sorting off. `turn_on_sorting` should be called before the queue *queue* is used.

2.4. Audio Codec Controlling : *Codec Management Module*

2.4.1. *Codec Management* Overview

A multimedia terminal may support multiple audio codec sets. For every logical channel there can be different codec than in other channels. This (management) module was developed to allow many codecs to be used simultaneously (e.g., while transmitting stereo signal, when the first stream may be coded using G.711 and the other using G.723.1), to make playing and recording threads unconscious of audio codecs used, and finally, to simplify process of selecting and changing audio codecs. Notice that if audio codec is compatible with the *Codec Management* module, it could not be used without his module. The way how to add a new codec is described in Section 2.4.3.

Playing and recording threads get a pointer to a codec structure as a parameter. This structure contains pointers to coding and decoding functions of particular codec as well as all parameters of a given codec. So *Capturing and Playing Module* thread just call the `snd_encode` or `snd_decode` function with a codec object (a pointer to a codec structure).

To select a particular codec (by the use of `snd_get_coder` and `snd_get_decoder`), a symbol of the codec is needed. The symbol type is the codec_symbol_t type (enumerated one). The following symbols are defined: `g711Alaw64k` , `g711Alaw56k` , `g711Ulaw64k` , `g711Ulaw56k` , `g722_64k` , `g722_56k` , `g722_48k` , `g7231` , `g728` , `g729` , `is11172AudioCapability` , `g729AnnexA` , `is13818AudioCapability` . Most of them are not implemented yet. The mentioned functions will return NULL if called with the symbol of an unimplemented codec . The structure that describes codecs is called codec_struct, and the pointer to this structure is defined p_cd.

2.4.2. *Codec Management* API

- `snd_init_codecs` Initiates all global structures for codecs, allocates global tables of codecs (coders and decoders).

```
int snd_init_codecs(void);
```

This function must be called BEFORE any other `snd_` function is used.

- `snd_get_coder` and `snd_get_decoder` Select coder/decoder from the global set of coders/decoders.

```
p_cd snd_get_coder(codec_symbol_t codec);
p_cd snd_get_decoder(codec_symbol_t codec);
```

When a `p_cd` pointer is returned by this call it can be used in other calls which require `p_cd` structure pointer. If coder/decoder cannot be selected this function returns NULL.

- `snd_get_first_coder` and `snd_get_first_decoder` Selects the first coder/decoder from the global set of codecs.

```
p_cd snd_get_first_coder(void);
p_cd snd_get_first_decoder(void);
```

These functions may be used when the user does not care about which codec will be chosen, or to make a scanning over the codecs (used with the `snd_get_next_` functions). The first case is a bad idea but until now this case is used in this project.

- `snd_get_next_coder` and `snd_get_next_decoder` Selects the next coder/decoder from the global set of coders/decoders.

```
p_cd snd_get_next_coder(p_cd codec);
p_cd snd_get_next_decoder(p_cd codec);
```

These functions select the following codec than the one specified in the *`codec`* parameter. This codec is returned by the functions. The current codec (*`codec`*) if de-selected (like after calling snd_unget_codec()). If no more codecs are available NULL is returned, and the current codec is left untouched (is not de-selected).

- `snd_unget_codec` De-selects a codec (coder or decoder).

```
int snd_unget_codec(p_cd codec);
```

This function frees all data connected with the codec.

- `snd_encode` and `snd_decode` Encode/decode audio data.

```
int snd_encode(p_cd codec, void *input_buffer, void *output_buffer);
int snd_decode(p_cd codec, void *input_buffer, void *output_buffer);
```

The parameters are as follows:

- *`codec`* in a codec to be used to encode/decode data.
- *`in_buffer`* The input buffer. Points to a block (vector of samples) to be encoded/decoded.
- *`out_buffer`* The output buffer. Points to a memory block allocated before, and is filled with encoded/decoded data after successful completion of the `snd_encode` or `snd_decode` function.

All additional parameters (e.g., the length of input/output buffers) are taken from the *`codec`* parameter by the encoding/decoding function.

- `snd_get_input_block_len` and `snd_get_output_block_len` Returns the length of the input/output data block.

```
int snd_get_input_block_len(p_cd codec);
int snd_get_output_block_len(p_cd codec);
```

This information is taken from the *`codec`* parameter.

- `snd_add_coder` and `snd_add_decoder` Add a new coder/decoder to the global set of coders/decoders.

```
int snd_add_coder(p_cd codec);
int snd_add_decoder(p_cd codec);
```

The *codec* parameter points to a codec object that is to be added. After successful completion of this function the codec can be used by audio threads.

- `snd_add_coders` and `snd_add_decoders` Add a set of coders/decoders to the global set of coders/decoders.

```
int snd_add_coders(p_cd *codec_p);
int snd_add_decoders(p_cd *codec_p);
```

The *codec_p* parameter points to a table of codecs finished with the NULL pointer.

- `snd_get_number_of_coders` and `snd_get_number_of_decoders` Returns the number of all coders/decoder ready to be used.

```
int snd_get_number_of_coders(void);
int snd_get_number_of_decoders(void);
```

These numbers are taken from the global codec structure, allocated by the snd_init_codecs() function.

Example 2-1 illustrates the possible scenario of using **codec_mngmnt**.

```
#include <stdio.h>
#include "codec_mngmnt.h"

codec_example()
{
 snd_init_codecs(); /* This must be called before any
      other snd_ function is called */

 snd_add_coders( g711_capabilities() );
 snd_add_decoders( g711_capabilities() );
    /* These add features from g711 module. Every particular
   codec module must be initited in a such way */

 printf("Number of coders: %d\n", snd_get_number_of_coders());
```

```
 printf("Number of decoders: %d\n", snd_get_number_of_decoders());


 printf("Getting decoders..\n");
 tmp_c = snd_get_first_decoder();
 if(!tmp_c)
   printf("Cannot get decoder (no decoders present?).");
 else
   printf("Decoder symbol: %s (%d), name : %s\n",
    tmp_c->string_symbol,
     tmp_c->symbol, tmp_c->name);

 while(tmp_c = snd_get_next_decoder(tmp_c))
   printf("Decoder symbol: %s (%d), name : %s\n",
    tmp_c->string_symbol,
     tmp_c->symbol, tmp_c->name);


 snd_unget_codec(tmp_c);

}
```

Example 2-1. Codec Management Example

2.4.3. How to add a new codec

To add a new codec to the group of supported codecs the following s.pdf must be performed:

- First, the codec must be written down. It must support the interface similar to: `encode(void *input, void *output, ...)` and `decode(void *input, void *output, ...)`. The first step is the transformation the above interface into:

```
void *CODEC_NAME_encode(p_cd coder_ptr, void *input_block, void
  *output_block);
void *CODEC_NAME_decode(p_cd decoder_ptr, void *input_block, void
  *output_block);
```

- Now codec must recognize the fields from p_cd *codec_ptr*. The structure is defined as follows:

```
struct codec_struct {
 enum codec_symbol_t symbol;
 char *string_symbol;
 char *name;  /* in case of named codec */
 int  bit_rate;  /* in bits per second */
 float  quality; /* scale: 0-5 */
 unsigned int dynamic : 1;
 unsigned int active : 1;
 unsigned int number : 6; /* up to 64 */
 int (*setparam)(p_cd, void *param_struct);
 int (*encode)(p_cd, void *input, void *output);
 int (*decode)(p_cd, void *input, void *output);
 struct local_params {
  void *param_struct;
  int   input_buf_len;
  int   output_buf_len;
 } *lp;
};
```

  So the parameters of the current coder or decoder are taken from this structure (especially from *lp*). Every codec type can support its own parameters in *lp->param_struct*. The length of the input buffer and output buffer can be stored in *ld->input_buf_len* and *ld->output_buf_len*, respectively.

- The function that fills the structure properly must be written down. symbol is the symbol of the codec. If this symbol cannot be found in enum codec_symbol_t table in `codec_mngmnt.h` it should be added manually to the list. *string_symbol* is used to show the readable name to the user. *name* may be used to add additional description for the codec; if not it should be filled with NULL. *bit_rate* is the bit-rate generated per second for encoded stream. *quality* is a quality of the sound (in the codec author's opinion). The scale if from 0 to 5. 64-bit G.711 would have this parameter well over 4, very-low-rate vocoders would have 2 or less. *dynamic* should be 0, *active* should be 1. *encode* should be set as the pointer to `encode` function prepared earlier, and *decode* function should be set as `decode` one. `setparam` function should be written by the author for the given codec (it sets a special parameters). The parameter structure pointer would be cast into void* and stored in *ld->param_struct*

- The function `p_cd *CODEC_NAME_capabilities` must be written down. Refer G.711 module (`g711.c`) for an example.

- The following lines should be added at the bottom of `snd_init_codecs` (in file `codec_mngmnt.c`):

  ```
  snd_add_coders(CODEC_NAME_capabilities());
  snd_add_decoders(CODEC_NAME_capabilities());
  ```

- The codec should be tested whether it can be selected, de-selected and re-selected. The codec test suite is included in `test_codec_management.c`. Feel free to modify it.

2.5. *Capturing and Playing Module*

If the user is going to start recording procedures, he must first prepare two objects: a queue and a codec. The queue object is a pointer to the queue structure, where encoded data is stored for RTP. The codec object is a pointer to a codec structure, containing all coding parameters and functions. These objects (a queue and a codec) are passed to a capturing thread.

If user is about to start playing procedures, `start_player` is called. Queue and codec are passed to this function. In this case 'queue' means the queue structure, from which encoded frames are taken. These frames are put there by RTP thread. Codec is a pointer to decoder structure, used while audio decoding. The third parameter is a number of buffered frames in queue. Playing thread is a part of receiver - it must take care whether signal is played continuously. The length of the buffer is taken from the queue structure. The more frames are buffered, the bigger delay, but also the less probability of losing continuousity of the signal. This value is constant. Until now, no adaptive procedures have been prepared. To stop the playing thread `stop_player` is used.

The API is as follows:

- `check_sound_devices` Examines the local sound system.

```
int check_sound_devices (void);
```

  This function examines if the sound device(s) is (are) present and configured in the OS. Then it checks whether it supports 8kHz/16bit sampling and playing (for wideband codecs 16kHz should be added). Finally, the full-duplex is examined. If all test are successful, 0 is returned. Otherwise this function returns a non-zero value.

- `start_recorder` Runs a capturing thread.

```
snd_thread_id_ptr start_recorder (fqp queue_ptr, p_cd coder_ptr, int
  system_buf_len, int system_buf_num);
```

Four parameters are passed to this function:

- *queue_ptr* is a pointer to a queue object prepared earlier.
- *codec_ptr* is a pointer to an audio coder created and configured earlier.
- *system_buf_len* specifies the length of every system buffer.
- *system_buf_num* specifies the number of system buffers (the more the better).

The start_recorder function returns a special handler (of type snd_thread_id_ptr), which can be used to control the capturing thread. If the returned value is not NULL, success is assumed. From the user point of view start_recorder is a non-blocking function. It creates a new thread for the capturing procedures.

- start_player Runs a playing thread.

```
snd_thread_id_ptr start_player (fqp queue_ptr, p_cd decoder_ptr, int
  system_buf_len, int system_buf_num, int buffered_frames);
```

Five parameters are passed to this function (four of them are similar the the start_recorder's parameters):

- *queue_ptr* is a pointer to a queue object prepared earlier.
- *codec_ptr* is a pointer to an audio decoder created and configured earlier.
- *system_buf_len* specifies the length of every system buffer.
- *system_buf_num* specifies the number of system buffers (the more the better).
- *buffered_frames* specifies the initial number of buffered frames in the receiver (anti-jitter buffer length). This number can vary in time of continuous playing.

This function creates a playing thread. It receives audio frames from an RTP thread (by the use of the audio queue passed to the function). First, it waits for *buffered_frames* number of received frames. Then it starts the commonplace *get-decode-play* procedure.

- stop_sound_thread Stops a capturing/recording thread.

```
int stop_sound_thread (snd_thread_id_ptr sound_thread);
```

This function is called if a capturing/playing thread if no longer needed. It stops capturing/playing, frees all allocated data, closes input/output sound device, and kills the sound thread.

2.6. Implementation of G.711 codec

The *ITU-T Rec. G.711* presents two PCM audio codecs called A-law and U-law. They both transform linear PCM signal into logarithmic PCM. They both operate on single samples.

A-law uses 13-bit linear PCM vector and transforms it into 8-bit logarithmic PCM vector while encoding process. U-law uses 14-bit linear PCM, transforming it into 8-bit. Non-professional sound devices cannot generate nether 13 nor 14-bit samples. In this implementation 16-bit samples are passed and the input of coder. Every sample is converted into 13 or 14-bit sample by cutting off the less significant bits.

Coding process does not require logarithmic calculations. *ITU-T Rec. G.711* [3] defines tables, that are used while coding 7and decoding process. Although A-law and U-law operate on the same algorithms, they use slightly different tables. A-law is used as an example in this implementation of G.711 codec.

14-bit input signal for A-law coder can take value from between -4096 and 4096. This range is divided into several segments having various length. And each segment is divided into equal length intervals with interval size characteristic for each interval. The sum of all intervals in all segments is equal 255 (to fit it into one byte). For greater values of the input signal, the intervals are bigger, e.g. 128 for intervals from between 2048 and 4096, so bigger values are encoded with bigger coding error. While encoding a sample, first the segment is determined. Next, appropriate interval is this segment must be determined (it is called offset). Next, sum of all intervals in less segments is added to this offset and this is the 8-bit encoded value. While decoding, appropriate segment and appropriate interval are determined. The output value is always treated as if it was the value from the middle of the interval (the average value) and in this way original value is restored.

This codec is compatible with codec management module. After selection of this codec functions `snd_encode`, `snd_decode` are used to transform the audio signal. But first the length of the input buffer must be set, using auxiliary `snd_` functions.

Bibliography

[1] Adam Drozdek, *Wprowadzenie do kompresji danych*, WNT, Warszawa, 1997.

[2] Jerry D. Gibson, Toby Berger, Tom Lookabaugh, Dave Lindbergh, and Richard L. Baker, *Digital Compression for Multimedia: Principles and standards*, 1998.

[3] *CCITT Recommendation G.711: Pulse Code Modulation (PCM) of voice frequencies*, 1988.

[4] *ITU-T Recommendation G.723.1: Speech coders: Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s*, 1996.

[5] *ITU-T Recommendation H.225: Call signalling protocols and media stream packetization for packet based multimedia communication system*, 1998.

[6] *ITU-T Recommendation H.245: Control protocol for multimedia communication*, 07/97.

[7] *ITU-T Recommendation H.323: Packet-based multimedia communications systems*, 02/98.

[8] *Microsoft Developer Studio Documentation (taken from VC++): Platform Software Development Kit Documentation*.

[9] *RTP: A Transport Protocol for Real-Time Applications*, 1996, H. Schulzrinne, et al..

[10] Mahbub Hassan, Alfandika Nayandoro, Mohammed Atiquzzaman, *Internet Telephony:Services, Technical Challenges, and Products*, IEEE Communications Magazine, p. 96-102, April 2000.

[11] Charles Petzoldt, Paul Yao, *Programowanie w Windows 95: Kompletny przewodnik programisty po API Windows 95*, 1997.

[12] Hannu Savolainen, *Open Sound System - Audio Programming: http://www.opensound.com/pguide*.

Appendix A. Virtual System API

This module is a set of virtual “system calls”. This is an ANSI-C code, but the ANSI standard C library does not support some useful features of modern OSes (e.g. no multi-thread support is present). To make modules free of system dependent system calls (i.e. to make the project more portable) module “Virtual System API” is used in any other module when it must use non-ANSI features. The following structures are present:

To use shared data the following structures were prepared: semaphore_struct and ssp, defined as

```
typedef struct semaphore_struct *ssp;
```

These structures are used to manage a “semaphore structure”. The variable of the type ssp is returned by function creating this structure and used later to protect some shared data. There is no need to introduce the field of the structure. *In Windows environment they are implemented using Mutexes and Events.*

To manage multiple threads the following structures are used: thread_id and tidp, defined as:

```
typedef struct thread_id *tidp;
```

The following API has been prepared:

- `prepare_ss` Allocates a new semaphore structure object.

```
ssp prepare_ss(void);
```

Prepares a new semaphore structure, returns pointer to newly allocated structure or NULL if no data can be allocated.

- `lockit` Locks a semaphore.

```
void lockit(ssp ss_object);
```

Locks the semaphore that is pointed by *`ss_object`*. If someone has already locked the semaphore, this functions blocks. *No processes using `lock`/`unlock` pair can come into critical section.*

- `unlockit` Unlocks a semaphore.

```
void unlockit(ssp ss_object);
```

Unlocks the semaphore that pointed by the *ss_object* semaphore object. If there is one blocked thread on this semaphore, it is unblocked and gets into the critical section. If multiple threads are blocked on the semaphore, one of them is unblocked.

- `checklock` Checks whether a semaphore is locked.

```
void checklock(ssp ss_object);
```

Returns 1 if semaphore is already locked or 0 if not.

- `wait_for_resource` Starts waiting for a “new resource”.

```
void wait_for_resource(ssp ss_object);
```

Blocks waiting for an event (called here a “new resource”). This function must be called when inside critical section (`lockit` called before) because it unlocks this semaphore to allow the other thread go inside critical section to call `new_resource` (described below).

- `new_resource` Signals a “new resource”.

```
void new_resource(ssp ss_object);
```

If any other thread is waiting for a “new resource” (it called `wait_for_resource` function) this functions unblocks the waiting thread. Otherwise it does nothing.

- `make_thread` Creates a new thread.

```
tidp make_thread(fun_ptr function, void *parameter);
```

Creates new thread with function *function* and parameters passed to it in *param*. Returns pointer to this thread or NULL if it cannot be created.

- `kill_thread` Kills (terminates) a running thread.

```
tidp kill_thread(tidp thread_ID);
```

This function is called from the outside of the thread. This functions should be used with consciousness what it does in a given OS. It must not be used with threads having critical sections.

- `exit_thread` Terminates a thread.

```
void exit_thread(int retval);
```

This function is called from the inside of the thread, which wants to terminate itself (exit). *retval* is a return value.

- `fallasleep` Portable sleep().

```
void fallasleep(int seconds);
```

Just falls asleep for *seconds* of seconds.

- `put_message` Prints a message.

```
void put_message(const char *seconds ...);
```

Prints the message *s*. This function works like `printf` so multiple arguments are possible. In this, console version it calls `printf`.

- `put_debug_message` Prints a message when debugging.

```
void put_debug_message(const char *seconds ...);
```

Prints message only when `_DEBUG` is defined.

Appendix B. WaveForm API

This is the full Waveform Audio API.

```
MMRESULT waveInAddBuffer ( HWAVEIN hwi , LPWAVEHDR pwh , UINT
cbwh );
MMRESULT waveInClose ( HWAVEIN hwi );
MMRESULT waveInGetDevCaps ( UINT uDeviceID , LPWAVEINCAPS pwic ,
UINT cbwic );
MMRESULT waveInGetErrorText ( MMRESULT mmrError , LPSTR pszText ,
UINT cchText );
MMRESULT waveInGetID ( HWAVEIN hwi , LPUINT puDeviceID );
UINT waveInGetNumDevs ( VOID );
MMRESULT waveInGetPosition ( HWAVEIN hwi , LPMMTIME pmmt , UINT
cbmmt );
DWORD waveInMessage ( HWAVEIN hwi , UINT uMsg , DWORD dwParam1 ,
DWORD dwParam2 );
MMRESULT waveInOpen ( LPHWAVEIN phwi , UINT uDeviceID ,
LPWAVEFORMATEX pwfx , DWORD dwCallback , DWORD dwCallbackInstance
);
MMRESULT waveInPrepareHeader ( HWAVEIN hwi , LPWAVEHDR pwh , UINT
cbwh );
CALLBACK waveInProc ( HWAVEIN hwi , UINT uMsg , DWORD dwInstance
, DWORD dwParam1 , DWORD dwParam2 );
MMRESULT waveInReset ( HWAVEIN hwi );
MMRESULT waveInStart ( HWAVEIN hwi );
MMRESULT waveInStop ( HWAVEIN hwi );
MMRESULT waveInUnprepareHeader ( HWAVEIN hwi , LPWAVEHDR pwh ,
UINT cbwh );
```

```
MMRESULT waveOutBreakLoop ( HWAVEOUT hwo );
MMRESULT waveOutClose ( HWAVEOUT hwo );
MMRESULT waveOutGetDevCaps ( UINT uDeviceID , LPWAVEOUTCAPS pwoc ,
UINT cbwoc );
MMRESULT waveOutGetErrorText ( MMRESULT mmrError , LPSTR pszText ,
UINT cchText );
MMRESULT waveOutGetID ( HWAVEOUT hwo , LPUINT puDeviceID );
UINT waveOutGetNumDevs ( VOID );
MMRESULT waveOutGetPitch ( HWAVEOUT hwo , LPDWORD pdwPitch );
```

```
 MMRESULT  waveOutGetPlaybackRate ( HWAVEOUT  hwo ,  LPDWORD  pdwRate
);
 MMRESULT  waveOutGetPosition ( HWAVEOUT  hwo ,  LPMMTIME  pmmt ,  UINT
 cbmmt );
 MMRESULT  waveOutGetVolume ( HWAVEOUT  hwo ,  LPDWORD  pdwVolume );
 DWORD  waveOutMessage ( HWAVEOUT  hwo ,  UINT  uMsg ,  DWORD  dwParam1
,  DWORD  dwParam2 );
 MMRESULT  waveOutOpen ( LPHWAVEOUT  phwo ,  UINT  uDeviceID ,
 LPWAVEFORMATEX  pwfx ,  DWORD  dwCallback ,  DWORD  dwCallbackInstance
);
 MMRESULT  waveOutPause ( HWAVEOUT  hwo );
 MMRESULT  waveOutPrepareHeader ( HWAVEOUT  hwo ,  LPWAVEHDR  pwh ,
 UINT  cbwh );
 CALLBACK  waveOutProc ( HWAVEOUT  hwo ,  UINT  uMsg ,  DWORD
 dwInstance ,  DWORD  dwParam1 ,  DWORD  dwParam2 );
 MMRESULT  waveOutReset ( HWAVEOUT  hwo );
 MMRESULT  waveOutSetPitch ( HWAVEOUT  hwo ,  DWORD  dwPitch );
 MMRESULT  waveOutSetPlaybackRate ( HWAVEOUT  hwo ,  DWORD  dwRate );
 MMRESULT  waveOutSetVolume ( HWAVEOUT  hwo ,  DWORD  dwVolume );
 MMRESULT  waveOutUnprepareHeader ( HWAVEOUT  hwo ,  LPWAVEHDR  pwh ,
 UINT  cbwh );
 MMRESULT  waveOutWrite ( HWAVEOUT  hwo ,  LPWAVEHDR  pwh ,  UINT  cbwh
);
```

The following figure (Figure B-1) shows how input device is opened, and how the first buffers are sent to the OS. This code is taken from the Sound Engine module, prepared by the author.

```
pwf.wBitsPerSample= 16;
pwf.wf.nChannels  = 1;
pwf.wf.nSamplesPerSec = 8000;

pwf.wf.wFormatTag = WAVE_FORMAT_PCM;
pwf.wf.nBlockAlign =
  pwf.wf.nChannels * pwf.wBitsPerSample / 8;
pwf.wf.nAvgBytesPerSec =
  pwf.wf.nSamplesPerSec * pwf.wf.nBlockAlign;

if (waveInOpen(&rip->hwi,
 /*WAVE_MAPPER*/ 0, (LPWAVEFORMATEX)&pwf,
       (DWORD) rip->eventh, 0, CALLBACK_EVENT )) {
 printf("Couldn't open sound device. Leaving..\n");
```

```
 goto problem;
}

/* Preparing system buffers */

sb = (WAVEHDR**) malloc(sizeof(WAVEHDR**) * system_buf_num);
if(!sb)
 goto problem;

   for (i = 0; i < system_buf_num; i++)
       sb[i] = NULL;

   for (i = 0; i < system_buf_num; i++) {

 count = i;
       sb[i] = (WAVEHDR*) malloc(sizeof(WAVEHDR));

       if (sb[i] == NULL) {
        put_debug_message("malloc() error!\n");
  goto problem;
 }

       sb[i]->lpData = (LPBYTE) malloc(system_buf_len);
       sb[i]->dwBufferLength = system_buf_len;
       sb[i]->dwBytesRecorded = 0;
       sb[i]->dwUser = 0;
       sb[i]->dwFlags = 0;
       sb[i]->dwLoops = 0;

 if(!sb[i]->lpData)
  goto problem;

       if (waveInPrepareHeader(rip->hwi, sb[i], sizeof(WAVEHDR))) {
  put_debug_message("waveInPrepareHeader problem!\n");
  goto problem;
       }
       if (waveInAddBuffer(rip->hwi, sb[i], sizeof(WAVEHDR))) {
           put_debug_message("waveInAddBuffer problem!\n");
  goto problem;
       }
   }
```

Figure B-1. A fragment of the capturing procedure