# Windows远程音频 源代码

## AudioHook.h

#include <stdio.h>

#define CINTERFACE

#include <Windows.h>

#include <detours.h>

#include <Audioclient.h>

#include <mmdeviceapi.h>

#include <io.h>

#include <process.h>

#define MAXLENGTH 76800

#define MAXSESSION 10

#define MAXPROCESS 40

#pragma comment(lib, "detours.lib")

struct AudioOfProcess{

int ProcessID;

int pLatest, pOldest, isEmpty;

BYTE AudioBuffer[MAXLENGTH];

void InitProcess();

};

struct AudioOfSession{

unsigned long SessionID;

int NumOfProcess;

AudioOfProcess ProcessAudio[MAXPROCESS];

void InitSession();

void DelProcess(int ProcessNum);

};

struct argument{

int numFrames;

BYTE\* pData;

int frameSize;

int SessionNum;

int ProcessNum;

};

class Sender{

int usingBuffer, filledSize;

UINT32 frameSize, packageSize;

void SendBuffer(int);

WAVEFORMATEXTENSIBLE wvFmt;

public:

argument arg;

Sender();

~Sender();

int getDataSize(int SN, int PN);

int getBytesPerMilli();

void InitArg();

void DelProcess();

void FillSenderBuffer(UINT32, BYTE\*);

void SetWAVFormat(WAVEFORMATEX\*);

void InitializeSender();

};

HRESULT STDMETHODCALLTYPE MyIACInitialize(IAudioClient \*,

AUDCLNT\_SHAREMODE,

DWORD,

REFERENCE\_TIME,

REFERENCE\_TIME,

const WAVEFORMATEX \*,

LPCGUID);

HRESULT STDMETHODCALLTYPE MyIARCGetBuffer(IAudioRenderClient \*,

UINT32,

BYTE \*\*);

HRESULT STDMETHODCALLTYPE MyIARCReleaseBuffer(IAudioRenderClient \*,

UINT32,

DWORD);

## AudioHook.cpp

#include "AudioHook.h"

HRESULT (STDMETHODCALLTYPE \*SysIACInitialize)(IAudioClient \*,

AUDCLNT\_SHAREMODE,

DWORD,

REFERENCE\_TIME,

REFERENCE\_TIME,

const WAVEFORMATEX \*,

LPCGUID) = NULL;

HRESULT (STDMETHODCALLTYPE \*SysIARCGetBuffer)(IAudioRenderClient \*,

UINT32,

BYTE \*\*) = NULL;

HRESULT (STDMETHODCALLTYPE \*SysIARCReleaseBuffer)(IAudioRenderClient \*,

UINT32,

DWORD) = NULL;

Sender \*mySender;

HANDLE hMutex[MAXSESSION][MAXPROCESS];

FILE\* log;

BOOL WINAPI DllMain(HINSTANCE hinst, DWORD dwReason, LPVOID reserved)

{

REFERENCE\_TIME hnsRequestedDuration = 10000000;

IMMDeviceEnumerator \*pEnumerator = NULL;

IMMDevice \*pDevice = NULL;

IAudioClient \*pAudioClient = NULL;

IAudioRenderClient \*pRenderClient = NULL;

WAVEFORMATEX \*pwfx = NULL;

HRESULT hr;

LONG error;

(void)hinst;

(void)reserved;

if (DetourIsHelperProcess()) {

return TRUE;

}

if (dwReason == DLL\_PROCESS\_ATTACH) {

mySender = new Sender;

mySender->InitArg();

hMutex[mySender->arg.SessionNum][mySender->arg.ProcessNum] = CreateMutex(NULL, FALSE, NULL);

CoInitialize(NULL);

hr = CoCreateInstance(

CLSID\_MMDeviceEnumerator, NULL,

CLSCTX\_ALL, IID\_IMMDeviceEnumerator,

(void\*\*)&pEnumerator);

hr = pEnumerator->lpVtbl->GetDefaultAudioEndpoint(pEnumerator,

eRender, eConsole, &pDevice);

hr = pDevice->lpVtbl->Activate(pDevice,

IID\_IAudioClient, CLSCTX\_ALL,

NULL, (void\*\*)&pAudioClient);

SysIACInitialize = pAudioClient->lpVtbl->Initialize;

hr = pAudioClient->lpVtbl->GetMixFormat(pAudioClient, &pwfx);

mySender->SetWAVFormat(pwfx);

hr = pAudioClient->lpVtbl->Initialize(pAudioClient,

AUDCLNT\_SHAREMODE\_SHARED,

0,

hnsRequestedDuration,

0,

pwfx,

NULL);

hr = pAudioClient->lpVtbl->GetService(pAudioClient,

IID\_IAudioRenderClient,

(void\*\*)&pRenderClient);

SysIARCGetBuffer = pRenderClient->lpVtbl->GetBuffer;

SysIARCReleaseBuffer = pRenderClient->lpVtbl->ReleaseBuffer;

DetourRestoreAfterWith();

printf("Detour start.\n");

DetourTransactionBegin();

DetourUpdateThread(GetCurrentThread());

DetourAttach(&(PVOID&)SysIACInitialize, MyIACInitialize);

error = DetourTransactionCommit();

DetourTransactionBegin();

DetourUpdateThread(GetCurrentThread());

DetourAttach(&(PVOID&)SysIARCGetBuffer, MyIARCGetBuffer);

error = DetourTransactionCommit();

DetourTransactionBegin();

DetourUpdateThread(GetCurrentThread());

DetourAttach(&(PVOID&)SysIARCReleaseBuffer, MyIARCReleaseBuffer);

error = DetourTransactionCommit();

}

else if (dwReason == DLL\_PROCESS\_DETACH) {

DetourTransactionBegin();

DetourUpdateThread(GetCurrentThread());

DetourDetach(&(PVOID&)SysIACInitialize, MyIACInitialize);

error = DetourTransactionCommit();

DetourTransactionBegin();

DetourUpdateThread(GetCurrentThread());

DetourDetach(&(PVOID&)SysIARCGetBuffer, MyIARCGetBuffer);

error = DetourTransactionCommit();

DetourTransactionBegin();

DetourUpdateThread(GetCurrentThread());

DetourDetach(&(PVOID&)SysIARCReleaseBuffer, MyIARCReleaseBuffer);

error = DetourTransactionCommit();

CoUninitialize();

mySender->DelProcess();

CloseHandle(hMutex[mySender->arg.SessionNum][mySender->arg.ProcessNum]);

}

return TRUE;

}

## GUIDs.cpp

#include <Audioclient.h>

#include <mmdeviceapi.h>

extern const CLSID CLSID\_MMDeviceEnumerator = \_\_uuidof(MMDeviceEnumerator);

extern const IID IID\_IMMDeviceEnumerator = \_\_uuidof(IMMDeviceEnumerator);

extern const IID IID\_IAudioClient = \_\_uuidof(IAudioClient);

extern const IID IID\_IAudioRenderClient = \_\_uuidof(IAudioRenderClient);

## MyIACInitialize.cpp

#include "AudioHook.h"

extern Sender \*mySender;

extern HRESULT (STDMETHODCALLTYPE \*SysIACInitialize)(IAudioClient \*,

AUDCLNT\_SHAREMODE,

DWORD,

REFERENCE\_TIME,

REFERENCE\_TIME,

const WAVEFORMATEX \*,

LPCGUID);

HRESULT STDMETHODCALLTYPE MyIACInitialize(IAudioClient \*This,

AUDCLNT\_SHAREMODE ShareMode,

DWORD StreamFlags,

REFERENCE\_TIME hnsBufferDuration,

REFERENCE\_TIME hnsPeriodicity,

const WAVEFORMATEX \*pFormat,

LPCGUID AudioSessionGuid)

{

HRESULT hr = S\_OK;

hr = SysIACInitialize(This, ShareMode, StreamFlags, hnsBufferDuration, hnsPeriodicity, pFormat, AudioSessionGuid);

mySender->SetWAVFormat((WAVEFORMATEX\*)pFormat);

return hr;

}

## MyIARCGetReleaseBuffer.cpp

#include "AudioHook.h"

BYTE \*pBuffer = NULL;

extern Sender\* mySender;

extern HRESULT (STDMETHODCALLTYPE \*SysIARCGetBuffer)(IAudioRenderClient \*,

UINT32,

BYTE \*\*);

extern HRESULT (STDMETHODCALLTYPE \*SysIARCReleaseBuffer)(IAudioRenderClient \*,

UINT32,

DWORD);

HRESULT STDMETHODCALLTYPE MyIARCGetBuffer(IAudioRenderClient \*This,

UINT32 NumFramesRequested,

BYTE \*\*ppData)

{

HRESULT hr = S\_OK;

hr = SysIARCGetBuffer(This, NumFramesRequested, ppData);

pBuffer = \*ppData;

return hr;

}

HRESULT STDMETHODCALLTYPE MyIARCReleaseBuffer(IAudioRenderClient \*This,

UINT32 NumFramesWritten,

DWORD dwFlags)

{

HRESULT hr = S\_OK;

mySender->FillSenderBuffer(NumFramesWritten, pBuffer);

hr = SysIARCReleaseBuffer(This, NumFramesWritten, dwFlags);

return hr;

}

## Sender.cpp

#include "AudioHook.h"

#pragma data\_seg("SharedData")

BOOL AvailSession[MAXSESSION] = {0};

AudioOfSession SessionAudio[MAXSESSION] = {0};

WAVEFORMATEXTENSIBLE waveFormat = {0};

#pragma data\_seg()

#pragma comment(linker,"/SECTION:SharedData,RWS")

extern HANDLE hMutex[MAXSESSION][MAXPROCESS];

extern"C" WAVEFORMATEXTENSIBLE \_\_declspec(dllexport) getFormat() {

return waveFormat;

}

void AudioOfProcess::InitProcess(){

ProcessID = 0;

pLatest = 0; pOldest = 0; isEmpty = 1;

for(int i = 0; i < MAXLENGTH; i++)

AudioBuffer[i] = 0;

}

void AudioOfSession::InitSession(){

SessionID = 0;

NumOfProcess = 0;

for(int i = 0; i< MAXPROCESS; i++)

ProcessAudio[i].InitProcess();

}

void AudioOfSession::DelProcess(int ProcessNum){

NumOfProcess --;

ProcessAudio[ProcessNum] = ProcessAudio[NumOfProcess];

ProcessAudio[NumOfProcess].InitProcess();

}

Sender::Sender() {

}

void Sender::SetWAVFormat(WAVEFORMATEX \*pwfx) {

if (pwfx->wFormatTag == WAVE\_FORMAT\_EXTENSIBLE)

wvFmt = \*(WAVEFORMATEXTENSIBLE\*)pwfx;

else

wvFmt.Format = \*pwfx;

waveFormat = wvFmt;

frameSize = pwfx->nChannels \* pwfx->wBitsPerSample / 8;

}

void fillBuffer(void\*);

void Sender::InitArg(){

int curSession, curProcess;

unsigned long ProcessID;

unsigned long SessionID;

ProcessID = GetCurrentProcessId();

ProcessIdToSessionId(ProcessID, &SessionID);

arg.SessionNum = -1;

arg.ProcessNum = -1;

for(curSession = 0; curSession < MAXSESSION; curSession ++){ //search for this session

if(AvailSession[curSession] == TRUE && SessionAudio[curSession].SessionID == SessionID){

arg.SessionNum = curSession;

break;

}

else if(AvailSession[curSession] == FALSE && SessionAudio[curSession].SessionID != 0) //clear spare sessions

SessionAudio[curSession].InitSession();

}

if(curSession == MAXSESSION){ //if this is a new sesssion

for(curSession = 0; curSession < MAXSESSION; curSession++){

if(AvailSession[curSession] == FALSE){

AvailSession[curSession] = TRUE;

arg.SessionNum = curSession;

SessionAudio[curSession].SessionID = SessionID;

break;

}

}

}

if(arg.SessionNum >= 0){ //if sessions are not full

for(curProcess = 0; curProcess < SessionAudio[arg.SessionNum].NumOfProcess; curProcess ++){ //search for this process

if(ProcessID == SessionAudio[arg.SessionNum].ProcessAudio[curProcess].ProcessID){

arg.ProcessNum = curProcess;

break;

}

}

if(curProcess == SessionAudio[arg.SessionNum].NumOfProcess){ //if this is a new process

arg.ProcessNum = curProcess;

SessionAudio[arg.SessionNum].ProcessAudio[curProcess].ProcessID = ProcessID;

SessionAudio[arg.SessionNum].NumOfProcess ++;

}

}

}

void Sender::DelProcess(){

if(arg.SessionNum >=0 && arg.ProcessNum >= 0)

SessionAudio[arg.SessionNum].DelProcess(arg.ProcessNum);

}

void Sender::FillSenderBuffer(UINT32 numFrames, BYTE\* pData) {

arg.numFrames = numFrames;

arg.frameSize = frameSize;

arg.pData = pData;

if(arg.ProcessNum >= 0) //if processes are not full

\_beginthread(fillBuffer, 0, &arg);

}

void fillBuffer(void \*arg) {

struct argument \*a = (struct argument\*)arg;

int numBytes = a->numFrames \* a->frameSize;

int SN = a->SessionNum;

int PN = a->ProcessNum;

int pLatest = SessionAudio[SN].ProcessAudio[PN].pLatest;

int pOldest = SessionAudio[SN].ProcessAudio[PN].pOldest;

int isEmpty = SessionAudio[SN].ProcessAudio[PN].isEmpty;

BYTE\* audioBuffer = SessionAudio[SN].ProcessAudio[PN].AudioBuffer;

WaitForSingleObject(hMutex[SN][PN], INFINITE);

if(numBytes > MAXLENGTH){

numBytes = MAXLENGTH;

a->pData += numBytes - MAXLENGTH;

}

if (pLatest + numBytes <= MAXLENGTH)

memcpy(audioBuffer + pLatest, a->pData, numBytes);

else {

memcpy(audioBuffer + pLatest, a->pData, MAXLENGTH - pLatest);

memcpy(audioBuffer, a->pData + (MAXLENGTH - pLatest), numBytes - (MAXLENGTH - pLatest));

}

if (pOldest < pLatest || (pOldest == pLatest && isEmpty == 1)) {

if (pLatest + numBytes - MAXLENGTH > pOldest) {

SessionAudio[SN].ProcessAudio[PN].pLatest = SessionAudio[SN].ProcessAudio[PN].pOldest = pLatest + numBytes - MAXLENGTH;

SessionAudio[SN].ProcessAudio[PN].isEmpty = 0;

}

else {

SessionAudio[SN].ProcessAudio[PN].pLatest = (pLatest + numBytes) % MAXLENGTH;

if(numBytes != 0)

SessionAudio[SN].ProcessAudio[PN].isEmpty = 0;

}

}

else {

if (pLatest + numBytes > pOldest) {

SessionAudio[SN].ProcessAudio[PN].pLatest = SessionAudio[SN].ProcessAudio[PN].pOldest = (pLatest + numBytes) % MAXLENGTH;

}

else

SessionAudio[SN].ProcessAudio[PN].pLatest += numBytes;

}

ReleaseMutex(hMutex[SN][PN]);

}

int Sender::getDataSize(int SN, int PN) {

int pLatest = SessionAudio[SN].ProcessAudio[PN].pLatest;

int pOldest = SessionAudio[SN].ProcessAudio[PN].pOldest;

int isEmpty = SessionAudio[SN].ProcessAudio[PN].isEmpty;

if (pOldest < pLatest)

return (pLatest - pOldest);

else if (pOldest > pLatest)

return (pLatest + MAXLENGTH - pOldest);

else

return isEmpty?0:MAXLENGTH;

}

int Sender::getBytesPerMilli() {

return wvFmt.Format.nAvgBytesPerSec / 1000;

}

## getAudio.cpp

#include "AudioHook.h"

extern Sender \*mySender;

extern HANDLE hMutex[MAXSESSION][MAXPROCESS];

extern AudioOfSession SessionAudio[MAXSESSION];

extern BOOL AvailSession[MAXSESSION];

int getAll(int SessionNum, int ProcessNum, BYTE\* pAudio);

extern"C" int \_\_declspec(dllexport) delCurSession(){

int curSession;

unsigned long ProcessID, SessionID;

ProcessID = GetCurrentProcessId();

ProcessIdToSessionId(ProcessID, &SessionID);

for(curSession = 0; curSession < MAXSESSION; curSession ++){ //search for this session

if(AvailSession[curSession] == TRUE && SessionAudio[curSession].SessionID == SessionID){

AvailSession[curSession] = FALSE;

break;

}

}

return 0;

}

extern"C" int \_\_declspec(dllexport) getAudio(BYTE\* pAudio)

{

BYTE MainBuffer[MAXLENGTH] = {0};

BYTE mixbuffer[MAXLENGTH] = {0};

float nTemp;

int numBytes = 0;

int MaxBytes = 0;

int curSession;

unsigned long ProcessID, SessionID;

int i;

ProcessID = GetCurrentProcessId();

ProcessIdToSessionId(ProcessID, &SessionID);

for(curSession = 0; curSession < MAXSESSION; curSession ++){ //search for this session

if(AvailSession[curSession] == TRUE && SessionAudio[curSession].SessionID == SessionID){

break;

}

}

for(i = 0; i < SessionAudio[curSession].NumOfProcess; i++){

numBytes = 0;

WaitForSingleObject(hMutex[curSession][i], INFINITE);

numBytes = getAll(curSession, i, mixbuffer);

ReleaseMutex(hMutex[curSession][i]);

for(int j = 0; j < numBytes; j += 4){

nTemp = (\*(float \*)&MainBuffer[j]) + (\*(float \*)&mixbuffer[j]); //mix the music

\*(float \*)&MainBuffer[j] = nTemp;

}

if(numBytes > MaxBytes)

MaxBytes = numBytes;

}

memcpy(pAudio, MainBuffer, MaxBytes);

if(MaxBytes > 0)

printf("get audio: %d bytes\n", MaxBytes);

return MaxBytes;

}

int getAll(int SessionNum, int ProcessNum, BYTE\* pAudio) {

int SN = SessionNum;

int PN = ProcessNum;

int pLatest = SessionAudio[SN].ProcessAudio[PN].pLatest;

int pOldest = SessionAudio[SN].ProcessAudio[PN].pOldest;

int isEmpty = SessionAudio[SN].ProcessAudio[PN].isEmpty;

BYTE\* audioBuffer = SessionAudio[SN].ProcessAudio[PN].AudioBuffer;

int numBytes = 0;

while(mySender->getDataSize(SessionNum, ProcessNum) >= 10 \* mySender->getBytesPerMilli() && numBytes < MAXLENGTH)

numBytes += 10 \* mySender->getBytesPerMilli();

if (pLatest > pOldest || (pLatest == pOldest && isEmpty == 1)) {

if (numBytes > pLatest - pOldest)

numBytes = pLatest - pOldest;

memcpy(pAudio, audioBuffer + pOldest, numBytes);

SessionAudio[SN].ProcessAudio[PN].pOldest += numBytes;

}

else {

if (numBytes > pLatest + MAXLENGTH - pOldest)

numBytes = pLatest + MAXLENGTH - pOldest;

if (pOldest + numBytes <= MAXLENGTH) {

memcpy(pAudio, audioBuffer + pOldest, numBytes);

SessionAudio[SN].ProcessAudio[PN].pOldest += numBytes;

}

else {

memcpy(pAudio, audioBuffer + pOldest, MAXLENGTH - pOldest);

memcpy(pAudio + MAXLENGTH - pOldest, audioBuffer, numBytes - (MAXLENGTH - pOldest));

SessionAudio[SN].ProcessAudio[PN].pOldest = pOldest + numBytes - MAXLENGTH;

}

}

if (SessionAudio[SN].ProcessAudio[PN].pOldest == SessionAudio[SN].ProcessAudio[PN].pLatest) SessionAudio[SN].ProcessAudio[PN].isEmpty = 1;

return numBytes;

}

## InjectDLL.cpp

#undef UNICODE

#include <vector>

#include <string>

#include <windows.h>

#include <Tlhelp32.h>

using std::vector;

using std::string;

int main(void)

{

vector<string>processNames; //Hold every process available

PROCESSENTRY32 pe32;

pe32.dwSize = sizeof(PROCESSENTRY32);

HANDLE hTool32 = CreateToolhelp32Snapshot(TH32CS\_SNAPALL, NULL); //Create the snapshot

BOOL bProcess = Process32First(hTool32, &pe32); //Call this and see if its valid

if(bProcess == TRUE)

{

while((Process32Next(hTool32, &pe32)) == TRUE) //While processes left to be enumerated

{

processNames.push\_back(pe32.szExeFile); //Save process name

if(strcmp(pe32.szExeFile, “ ”) == 0) //Add the process we want to inject to

{

printf("%d\n", pe32.th32ProcessID);

char\* FullPath = "D:\\Audio Cap and Play\\TestCaptureAudio\\Debug\\AudioHook2.dll";

HANDLE hProcess = OpenProcess(PROCESS\_ALL\_ACCESS, FALSE, pe32.th32ProcessID);

//printf("%x\n", hProcess);

LPVOID LoadLibraryAddr = (LPVOID)GetProcAddress(GetModuleHandle("Kernel32.dll"),

"LoadLibraryA"); //Get LoadLibraryA address

int cb = (1 + lstrlenA(FullPath)) \* sizeof(char);

LPVOID LLParam = (LPVOID)VirtualAllocEx(hProcess, NULL, cb,

MEM\_RESERVE | MEM\_COMMIT, PAGE\_READWRITE); //Allocate some memory for DLL string

if (LLParam == NULL) printf("Allocate failed.\n");

else printf("Allocate success.\n");

if (!WriteProcessMemory(hProcess, LLParam, FullPath, cb, NULL)) //Write it

printf("Write failed.\n");

else printf("Write success.\n");

HANDLE hRemoteThread = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD\_START\_ROUTINE)LoadLibraryAddr,

LLParam, 0, NULL); //New thread with LoadLibrary as start and our string as param

if (hRemoteThread == NULL) printf("Create Thread failed.\n");

else printf("Create Thread success.\n");

CloseHandle(hProcess);

}

}

}

CloseHandle(hTool32);

system("pause");

return 0;

}

## AudioSender.h

#define WIN32\_LEAN\_AND\_MEAN

#include <Windows.h>

#include <stdio.h>

#include <winsock2.h>

#include <ws2tcpip.h>

#pragma comment(lib, "ws2\_32.lib")

#pragma comment(lib, "AudioHook2.lib")

#pragma comment(lib, "libfaac.lib")

typedef struct

{

BYTE \*in;

int channels;

int samplebytes;

int samplerate;

int bigendian;

int isfloat;

} pcmfile\_t;

#define FRAMELENGTH 8192

#define SERVER\_PORT "12345"

#define SERVER\_ADD "localhost"

extern"C" int \_\_declspec(dllexport) getAudio(BYTE\*);

## AudioSender.cpp

#include "AudioSender.h"

#include <faac.h>

#include <faaccfg.h>

#define MAXLENGTH 98304

BYTE get\_buffer[MAXLENGTH];

BYTE read\_buffer[FRAMELENGTH];

BYTE cir\_buffer[MAXLENGTH];

float encoder\_buffer[FRAMELENGTH];

BYTE buffer[MAXLENGTH];

faacEncHandle hEncoder;

unsigned long samplesInput, maxBytesOutput;

SOCKET ClientSocket = INVALID\_SOCKET, ListenSocket = INVALID\_SOCKET;

int InitEncoder();

int InitSocket();

pcmfile\_t sndf;

int pLatest = 0, pOldest = 0, isEmpty = 1;

size\_t wav\_read\_float32(pcmfile\_t \*sndf, float \*buf, size\_t num);

int getframe(BYTE\* pAudio);

int fillbuffer(BYTE \*pData, int numBytes);

int getDataSize();

LARGE\_INTEGER starttime2,endtime2,frequency;

int main() {

int numGet, iResult= 0;

int samplesRead, bytesEncode = 0, numFromCir;

int send\_offset;

while (1) {

InitEncoder();

InitSocket();

do {

numGet = 0;

numGet = getAudio(get\_buffer);

fillbuffer(get\_buffer, numGet);

send\_offset = 0;

while(getDataSize() >= 8192){

numFromCir = getframe(read\_buffer);

samplesRead = wav\_read\_float32(&sndf, encoder\_buffer, numFromCir/4);

if(samplesRead > 0){

if((bytesEncode = faacEncEncode(hEncoder, (int32\_t \*)encoder\_buffer, samplesRead, buffer + send\_offset, maxBytesOutput)) < 0)

printf("faacEncEncode Failed!\n");

send\_offset += bytesEncode;

}

}

if(bytesEncode > 0){

iResult = send(ClientSocket, (const char\*)buffer, send\_offset, 0);

}

} while (iResult >= 0);

printf("recv failed: %d\n", WSAGetLastError());

closesocket(ClientSocket);

faacEncClose(hEncoder);

WSACleanup();

}

return 0;

}

int InitEncoder(){

faacEncConfigurationPtr myFormat;

unsigned int sampleRate = 48000;

unsigned int nChannels = 2;

unsigned int mpegVersion = MPEG2;

unsigned int objectType = LOW;

unsigned int useMidSide = 1;

unsigned int useTns = 0;

unsigned int useLfe = 0;

unsigned int bitRate = 0;

unsigned int bandWidth = 16000;

unsigned int quantqual = 100;

unsigned int outputFormat = 1; //ADTS

unsigned int inputFormat = FAAC\_INPUT\_FLOAT;

unsigned int shortctl = SHORTCTL\_NORMAL;

sndf.bigendian = 0;

sndf.channels =2;

sndf.samplebytes = 4;

sndf.isfloat = 1;

sndf.samplerate = 48000;

sndf.in = read\_buffer;

if((hEncoder = faacEncOpen(sampleRate, nChannels, &samplesInput, &maxBytesOutput)) == NULL)

printf("faacEncOpen Failed!");

if((myFormat = faacEncGetCurrentConfiguration(hEncoder)) == NULL)

printf("faacEncGetCurrentConfiguration Failed!");

myFormat->mpegVersion = mpegVersion;

myFormat->aacObjectType = objectType;

myFormat->allowMidside = useMidSide;

myFormat->useTns = useTns;

myFormat->useLfe = useLfe;

myFormat->bitRate = bitRate;

myFormat->bandWidth = bandWidth;

myFormat->quantqual = quantqual;

myFormat->outputFormat = outputFormat;

myFormat->inputFormat = inputFormat;

myFormat->shortctl = shortctl;

if (!faacEncSetConfiguration(hEncoder, myFormat)) {

printf("Unsupported output format!\n");

}

return 0;

}

int InitSocket() {

int iResult;

WSADATA wsaData;

struct addrinfo \*result = NULL, \*ptr = NULL, hints;

iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

if (iResult != 0) {

printf("WSAStartup failed with error: %d\n", iResult);

return 1;

}

ZeroMemory( &hints, sizeof(hints) );

hints.ai\_family = AF\_INET;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP;

hints.ai\_flags = AI\_PASSIVE;

iResult = getaddrinfo(NULL, SERVER\_PORT, &hints, &result);

if (iResult != 0) {

printf("getaddrinfo failed: %d\n", iResult);

WSACleanup();

return 1;

}

ptr=result;

ListenSocket = socket(ptr->ai\_family, ptr->ai\_socktype, ptr->ai\_protocol);

if (ListenSocket == INVALID\_SOCKET) {

printf("Error at socket(): %ld\n", WSAGetLastError());

freeaddrinfo(result);

WSACleanup();

return 1;

}

iResult = bind( ListenSocket, result->ai\_addr, (int)result->ai\_addrlen);

if (iResult == SOCKET\_ERROR) {

printf("bind failed with error: %d\n", WSAGetLastError());

freeaddrinfo(result);

closesocket(ListenSocket);

WSACleanup();

return 1;

}

freeaddrinfo(result);

if ( listen( ListenSocket, SOMAXCONN ) == SOCKET\_ERROR ) {

printf( "Listen failed with error: %ld\n", WSAGetLastError() );

closesocket(ListenSocket);

WSACleanup();

return 1;

}

printf("Listening.\n");

ClientSocket = accept(ListenSocket, NULL, NULL);

if (ClientSocket == INVALID\_SOCKET) {

printf("accept failed: %d\n", WSAGetLastError());

closesocket(ListenSocket);

WSACleanup();

return 1;

}

printf("Accepted.\n");

}

size\_t wav\_read\_float32(pcmfile\_t \*sndf, float \*buf, size\_t num)

{

size\_t i = 0;

long int offset = 0;

unsigned char bufi[8];

if ((sndf->samplebytes > 8) || (sndf->samplebytes < 1))

return 0;

while (i<num) {

memcpy(bufi, sndf->in + offset, sndf->samplebytes);

offset += sndf->samplebytes;

if (sndf->isfloat){

buf[i] = (\*(float \*)&bufi) \* (float)32768;

}

i++;

}

return i;

}

int getframe(BYTE\* pAudio) {

//int numBytes = 10 \* mySender->getBytesPerMilli();

int numBytes = FRAMELENGTH;

if (pLatest > pOldest || (pLatest == pOldest && isEmpty == 1)) {

if (numBytes > pLatest - pOldest)

numBytes = pLatest - pOldest;

memcpy(pAudio, cir\_buffer + pOldest, numBytes);

pOldest += numBytes;

}

else {

if (numBytes > pLatest + MAXLENGTH - pOldest)

numBytes = pLatest + MAXLENGTH - pOldest;

if (pOldest + numBytes <= MAXLENGTH) {

memcpy(pAudio, cir\_buffer + pOldest, numBytes);

pOldest += numBytes;

}

else {

memcpy(pAudio, cir\_buffer + pOldest, MAXLENGTH - pOldest);

memcpy(pAudio + MAXLENGTH - pOldest, cir\_buffer, numBytes - (MAXLENGTH - pOldest));

pOldest = pOldest + numBytes - MAXLENGTH;

}

}

if (pOldest == pLatest) isEmpty = 1;

return numBytes;

}

int fillbuffer(BYTE \*pData, int numBytes) {

if(numBytes > MAXLENGTH){

numBytes = MAXLENGTH;

pData += numBytes - MAXLENGTH;

}

if (pLatest + numBytes <= MAXLENGTH)

memcpy(cir\_buffer + pLatest, pData, numBytes);

else {

memcpy(cir\_buffer + pLatest, pData, MAXLENGTH - pLatest);

memcpy(cir\_buffer, pData + (MAXLENGTH - pLatest), numBytes - (MAXLENGTH - pLatest));

}

if (pOldest < pLatest || (pOldest == pLatest && isEmpty == 1)) {

if (pLatest + numBytes - MAXLENGTH > pOldest) {

pLatest = pOldest = pLatest + numBytes - MAXLENGTH;

isEmpty = 0;

}

else

pLatest = (pLatest + numBytes) % MAXLENGTH;

}

else {

if (pLatest + numBytes > pOldest) {

pLatest = pOldest = (pLatest + numBytes) % MAXLENGTH;

isEmpty = 0;

}

else

pLatest += numBytes;

}

return numBytes;

}

int getDataSize() {

if (pOldest < pLatest)

return (pLatest - pOldest);

else if (pOldest > pLatest)

return (pLatest + MAXLENGTH - pOldest);

else

return isEmpty?0:MAXLENGTH;

}

## AudioReceiver.h

#define WIN32\_LEAN\_AND\_MEAN

#include <Windows.h>

#include <stdio.h>

#include "LibAudioPlayer.h"

#include <winsock2.h>

#include <ws2tcpip.h>

#include <faad.h>

#include <neaacdec.h>

#pragma comment(lib, "ws2\_32.lib")

#pragma comment(lib, "LibAudioPlayer.lib")

#pragma comment(lib, "libfaad.lib")

#define SERVER\_PORT "12345"

#define SERVER\_ADD "localhost"

## AudioReceiver.cpp

#include "AudioReceiver.h"

#include<Winbase.h>

#include <strsafe.h>

#define FRAMELENGTH 8192

#define MAXLENGTH 98304

BYTE recv\_buffer[MAXLENGTH];

BYTE decoder\_buffer[FRAMELENGTH];

void \*buffer;

SOCKET ConnectSocket;

NeAACDecHandle hAac;

NeAACDecFrameInfo hInfo;

int InitSocket();

int InitDecoder();

int get\_one\_ADTS\_frame(unsigned char\* buffer, size\_t buf\_size, unsigned char\* data ,size\_t\* data\_size);

void main() {

AudioPlayer \*ap = new AudioPlayer;

int numGet;

size\_t size;

unsigned char \*decoder\_offset;

unsigned long samplerate;

unsigned char channels;

InitSocket();

InitDecoder();

numGet = recv(ConnectSocket, (char\*)recv\_buffer, (6144/8)\*2\*5, 0);

if(get\_one\_ADTS\_frame(recv\_buffer, numGet, decoder\_buffer, &size) == 0)

if(NeAACDecInit(hAac, (unsigned char\*)decoder\_buffer, size, &samplerate, &channels) < 0){

printf("Error initializing decoder library.\n");

}

// Initialise the library using one of the initialization functions

ap->Start();

while (1) {

numGet = recv(ConnectSocket, (char\*)recv\_buffer, (6144/8)\*2\*10, 0); //maxBytesOutput

// Loop until decoding finished

if(numGet > 0){

decoder\_offset = recv\_buffer;

while(get\_one\_ADTS\_frame(decoder\_offset, numGet, decoder\_buffer, &size) == 0){

buffer = NeAACDecDecode(hAac, &hInfo, (unsigned char\*)decoder\_buffer, size);

if ((hInfo.error == 0) && (hInfo.samples > 0)){

ap->LoadData(hInfo.samples \* 4, (BYTE \*)buffer);

printf("LoadData: %d\n", hInfo.samples \* 4);

} else if (hInfo.error != 0) {

printf("NeAACDecDecode Failed!\n");

printf("Error: %s\n", NeAACDecGetErrorMessage(hInfo.error));

NeAACDecClose(hAac);

}

numGet -= size;

decoder\_offset += size;

}

}

}

NeAACDecClose(hAac);

}

int InitSocket() {

WSADATA wsaData;

int iResult;

struct addrinfo \*result = NULL, \*ptr = NULL, hints;

ConnectSocket = INVALID\_SOCKET;

iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);

if (iResult != 0) {

printf("WSAStartup failed with error: %d\n", iResult);

return 1;

}

ZeroMemory( &hints, sizeof(hints) );

hints.ai\_family = AF\_INET;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP;

iResult = getaddrinfo(SERVER\_ADD, SERVER\_PORT, &hints, &result);

if (iResult != 0) {

printf("getaddrinfo failed: %d\n", iResult);

WSACleanup();

return 1;

}

ptr=result;

ConnectSocket = socket(ptr->ai\_family, ptr->ai\_socktype, ptr->ai\_protocol);

if (ConnectSocket == INVALID\_SOCKET) {

printf("Error at socket(): %ld\n", WSAGetLastError());

freeaddrinfo(result);

WSACleanup();

return 1;

}

// Connect to server.

iResult = connect( ConnectSocket, ptr->ai\_addr, (int)ptr->ai\_addrlen);

if (iResult == SOCKET\_ERROR) {

closesocket(ConnectSocket);

ConnectSocket = INVALID\_SOCKET;

}

freeaddrinfo(result);

}

int InitDecoder(){

NeAACDecConfigurationPtr conf;

unsigned int sampleRate = 48000;

unsigned int nChannels = 2;

// Open the library

hAac = NeAACDecOpen();

// Get the current config

conf = NeAACDecGetCurrentConfiguration(hAac);

conf->defObjectType = LC;

conf->defSampleRate = sampleRate;

conf->outputFormat = FAAD\_FMT\_FLOAT;

// Set the new configuration

if(NeAACDecSetConfiguration(hAac, conf) == 0)

printf("NeAACDecSetConfiguration Failed!");

return 0;

}

int get\_one\_ADTS\_frame(unsigned char\* buffer, size\_t buf\_size, unsigned char\* data ,size\_t\* data\_size)

{

size\_t size = 0;

if(!buffer || !data || !data\_size )

{

return -1;

}

while(1)

{

if(buf\_size < 7 )

{

return -1;

}

if((buffer[0] == 0xff) && ((buffer[1] & 0xf0) == 0xf0) )

{

size |= ((buffer[3] & 0x03) <<11); //high 2 bit

size |= buffer[4]<<3; //middle 8 bit

size |= ((buffer[5] & 0xe0)>>5); //low 3bit

break;

}

--buf\_size;

++buffer;

}

if(buf\_size < size)

{

return -1;

}

memcpy(data, buffer, size);

\*data\_size = size;

return 0;

}

## LibAudioPlayer.h

#include <Windows.h>

#include <AudioClient.h>

#include <mmdeviceapi.h>

#include <process.h>

#include <stdio.h>

#define MAXLENGTH 98304

class AudioPlayer {

public:

LARGE\_INTEGER starttime4, endtime4;

CRITICAL\_SECTION dataProcessSection;

int playThreshold, lowerLimit;

DWORD flags;

IAudioClient \*pAudioClient;

IAudioRenderClient \*pRenderClient;

IMMDeviceEnumerator \*pEnumerator;

IMMDevice \*pDevice;

REFERENCE\_TIME hnsRequestedDuration;

REFERENCE\_TIME hnsActualDuration;

UINT32 bufferFrameCount;

WAVEFORMATEXTENSIBLE wvFmt;

BYTE\* audioBuffer;

int pOldest, pLatest, isEmpty;

HRESULT GetData(UINT32, BYTE\*);

void RunThread();

int getDataSize();

AudioPlayer();

HRESULT Start();

HRESULT LoadData(int numBytes, BYTE\* pData);

~AudioPlayer();

};

## LibAudioPlayer.cpp

#include "LibAudioPlayer.h"

#define SAFE\_RELEASE(punk) \

if ((punk) != NULL) \

{ (punk)->Release(); (punk) = NULL; }

const CLSID CLSID\_MMDeviceEnumerator = \_\_uuidof(MMDeviceEnumerator);

const IID IID\_IMMDeviceEnumerator = \_\_uuidof(IMMDeviceEnumerator);

const IID IID\_IAudioClient = \_\_uuidof(IAudioClient);

const IID IID\_IAudioRenderClient = \_\_uuidof(IAudioRenderClient);

AudioPlayer::AudioPlayer() {

HRESULT hr;

WAVEFORMATEX \*pwfx;

pAudioClient = NULL;

pRenderClient = NULL;

pEnumerator = NULL;

pDevice = NULL;

flags = 0;

hnsRequestedDuration = 1000000;

audioBuffer = new BYTE[MAXLENGTH];

pLatest = 0;

pOldest = 0;

isEmpty = 1;

InitializeCriticalSection(&dataProcessSection);

hr = CoInitialize(NULL);

hr = CoCreateInstance(

CLSID\_MMDeviceEnumerator, NULL,

CLSCTX\_ALL, IID\_IMMDeviceEnumerator,

(void\*\*)&pEnumerator);

hr = pEnumerator->GetDefaultAudioEndpoint(

eRender, eConsole, &pDevice);

hr = pDevice->Activate(

IID\_IAudioClient, CLSCTX\_ALL,

NULL, (void\*\*)&pAudioClient);

hr = pAudioClient->GetMixFormat(&pwfx);

wvFmt = \*(WAVEFORMATEXTENSIBLE\*)pwfx;

hr = pAudioClient->Initialize(

AUDCLNT\_SHAREMODE\_SHARED,

0,

hnsRequestedDuration,

0,

pwfx,

NULL);

// Get the actual size of the allocated buffer.

hr = pAudioClient->GetBufferSize(&bufferFrameCount);

hr = pAudioClient->GetService(

IID\_IAudioRenderClient,

(void\*\*)&pRenderClient);

}

void runThread(void\*);

HRESULT AudioPlayer::Start() {

\_beginthread(runThread, 0, (void\*)this);

return S\_OK;

}

void runThread(void \*audioPlayer) {

UINT32 numFramesAvailable;

UINT32 numFramesPadding;

UINT32 size;

HRESULT hr;

BYTE \*pData;

AudioPlayer \*ap = (AudioPlayer\*)audioPlayer;

LARGE\_INTEGER frequency;

EnterCriticalSection(&(ap->dataProcessSection));

hr = ap->pAudioClient->GetCurrentPadding(&numFramesPadding);

numFramesAvailable = (ap->bufferFrameCount - numFramesPadding < ap->getDataSize()) ?ap->bufferFrameCount - numFramesPadding : ap->getDataSize();

hr = ap->pRenderClient->GetBuffer(numFramesAvailable, &pData);

ap->GetData(numFramesAvailable, pData);

hr = ap->pRenderClient->ReleaseBuffer(numFramesAvailable, ap->flags);

LeaveCriticalSection(&(ap->dataProcessSection));

hr = ap->pAudioClient->Start(); // Start playing.

// Each loop fills about half of the shared buffer.

while (TRUE){

if(ap->getDataSize() >= 3840){

EnterCriticalSection(&(ap->dataProcessSection));

hr = ap->pAudioClient->GetCurrentPadding(&numFramesPadding);

numFramesAvailable = (ap->bufferFrameCount - numFramesPadding < ap->getDataSize()) ?ap->bufferFrameCount - numFramesPadding : ap->getDataSize();

hr = ap->pRenderClient->GetBuffer(numFramesAvailable, &pData);

ap->GetData(numFramesAvailable, pData);

hr = ap->pRenderClient->ReleaseBuffer(numFramesAvailable, ap->flags);

LeaveCriticalSection(&(ap->dataProcessSection));

}

}

hr = ap->pAudioClient->Stop();

}

AudioPlayer::~AudioPlayer() {

DeleteCriticalSection(&dataProcessSection);

SAFE\_RELEASE(pEnumerator)

SAFE\_RELEASE(pDevice)

SAFE\_RELEASE(pAudioClient)

SAFE\_RELEASE(pRenderClient)

CoUninitialize();

}

HRESULT AudioPlayer::GetData(UINT32 numFramesAvailable, BYTE\* pData) {

int numBytes = numFramesAvailable \* wvFmt.Format.nChannels \* wvFmt.Format.wBitsPerSample / 8;

if (pLatest > pOldest) {

if (numBytes > pLatest - pOldest)

numBytes = pLatest - pOldest;

memcpy(pData, audioBuffer + pOldest, numBytes);

pOldest += numBytes;

}

else {

if (numBytes > pLatest + MAXLENGTH - pOldest)

numBytes = pLatest + MAXLENGTH - pOldest;

if (pOldest + numBytes <= MAXLENGTH) {

memcpy(pData, audioBuffer + pOldest, numBytes);

pOldest += numBytes;

}

else {

memcpy(pData, audioBuffer + pOldest, MAXLENGTH - pOldest);

memcpy(pData + MAXLENGTH - pOldest, audioBuffer, numBytes - (MAXLENGTH - pOldest));

pOldest = pOldest + numBytes - MAXLENGTH;

}

}

if (pOldest == pLatest) isEmpty = 1;

return S\_OK;

}

HRESULT AudioPlayer::LoadData(int numBytes, BYTE\* pData) {

EnterCriticalSection(&dataProcessSection);

if (pLatest + numBytes <= MAXLENGTH)

memcpy(audioBuffer + pLatest, pData, numBytes);

else {

memcpy(audioBuffer + pLatest, pData, MAXLENGTH - pLatest);

memcpy(audioBuffer, pData + (MAXLENGTH - pLatest), numBytes - (MAXLENGTH - pLatest));

}

if (pOldest < pLatest || (pOldest == pLatest && isEmpty == 1)) {

if (pLatest + numBytes - MAXLENGTH > pOldest) {

pLatest = pOldest = pLatest + numBytes - MAXLENGTH;

isEmpty = 0;

}

else{

pLatest = (pLatest + numBytes) % MAXLENGTH;

if(numBytes != 0)

isEmpty = 0;

}

}

else {

if (pLatest + numBytes > pOldest) {

pLatest = pOldest = (pLatest + numBytes) % MAXLENGTH;

}

else

pLatest += numBytes;

}

LeaveCriticalSection(&dataProcessSection);

return S\_OK;

}

int AudioPlayer::getDataSize() {

if (pOldest < pLatest)

return (pLatest - pOldest);

else if (pOldest > pLatest)

return (pLatest + MAXLENGTH - pOldest);

else

return isEmpty?0:MAXLENGTH;

}