x86matthew - AudioTransmit - Transmitting data between computers using audio

x86matthew.com/view_post

AudioTransmit - Transmitting data between computers using audio

02/04/2022

This proof-of-concept project demonstrates a method of transmitting data to another computer without using traditional networking. We will be encoding binary data into software-generated audio tones to communicate with the target computer via the speaker. The receiving computer will use the microphone to analyse the raw PCM audio samples to decode the binary data stream.

This project currently only supports one-way communication - there is a receiver program and a sender program. For testing purposes, it is possible to run both of these on the same computer. Most laptops have integrated speakers and microphones which makes them ideal for this project.

The sender program works as follows:

1. Data to transmit is passed to the command-line: AudioTransmit_Send.exe "A test string"
2. The program creates a blank temporary .wav file.
3. A sine-wave is generated in a specific frequency to indicate the start of a new data transmission. This tone is appended to the wave file.

4. Each data byte is converted into bits - an "on" or "off" tone is generated for each bit. A separate confirmation tone is sent after each bit value.
5. A final tone is appended to indicate the end of the transmission.
6. The temporary wave file is played through the computer speakers and then deleted.

The receiver program works as follows:

1. Start listening for data: AudioTransmit_Listen.exe
2. The program will access the microphone using the waveInOpen API.
3. The program will continuously record 1 second audio samples (16-bit mono at 8,000Hz sample rate).
4. Each sample will be analysed for "message tones" from our sender program. This works using the Goertzel algorithm to detect specific frequencies within the audio samples. The CalculateToneMagnitude function detects the "strength" of the target frequency within the current audio sample.
5. When a signal frequency is detected (above the minimum threshold value: TONE_THRESHOLD), the specified signal will be processed. The threshold value can be modified to allow for different levels of volume and background noise. Data transmission will begin when the "transmission start" tone is detected.
6. When each set of 8 bits have been received, they will be converted back into a byte and printed to the console.
7. The program will continue to read data bits until the "transmission end" tone is detected.
8. The receiver will now wait for the next message.

This project is very basic and doesn't perform any validation/checksum tests on the received data, but it does seem to be fairly reliable even with a small amount of background noise.

The default tones used in this project are listed below:

Transmission Start:	2000Hz
Data Bit 1:	600Hz
Data Bit 0:	800Hz
Transmission End:	1500Hz
Next Bit Confirmation:	1200Hz

The transmission speed is determined by the TONE_LENGTH_MS value in the sender program. The receiver program is able to automatically adjust to any transfer speed because consecutive tones are never equal by design. Each tone has a duration of 50ms by default - this seems to be reliable in my testing.

Transmitter code:

```
#include <stdio.h>
#include <windows.h>
```

```
#include <math.h>

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

#define AUDIO_BITS_PER_SAMPLE 16
#define AUDIO_SAMPLE_RATE 8000

#define TONE_LENGTH_MS 50

#define TRANSMISSION_START_FREQUENCY 2000
#define BIT_TONE_FREQUENCY_ON 600
#define BIT_TONE_FREQUENCY_OFF 800
#define TRANSMISSION_END_FREQUENCY 1500
#define BIT_TONE_FREQUENCY_NEXT 1200

struct WaveHeaderStruct
{
DWORD dwChunkID;
DWORD dwChunkSize;
DWORD dwFormat;

DWORD dwSubChunk1ID;
DWORD dwSubChunk1Size;
WORD wAudioFormat;
WORD wNumChannels;
DWORD dwSampleRate;
DWORD dwByteRate;
WORD wBlockAlign;
WORD wBitsPerSample;

DWORD dwSubChunk2ID;
DWORD dwSubChunk2Size;
};

FILE *pGlobal_WaveFile = NULL;
DWORD dwGlobal_TotalWaveDataLength = 0;
WaveHeaderStruct Global_WaveHeader;

DWORD InitialiseWaveFile(char *pFilePath)
{
// create output file
pGlobal_WaveFile = fopen(pFilePath, "wb");
if(pGlobal_WaveFile == NULL)
{
return 1;
}
```

```
// reset data length
dwGlobal_TotalWaveDataLength = 0;

// generate initial wave header
memset((void*)&Global_WaveHeader, 0, sizeof(Global_WaveHeader));
Global_WaveHeader.dwChunkID = 0x46464952;
Global_WaveHeader.dwChunkSize = 36;
Global_WaveHeader.dwFormat = 0x45564157;
Global_WaveHeader.dwSubChunk1ID = 0x20746D66;
Global_WaveHeader.dwSubChunk1Size = 16;
Global_WaveHeader.wAudioFormat = 1;
Global_WaveHeader.wNumChannels = 1;
Global_WaveHeader.dwSampleRate = AUDIO_SAMPLE_RATE;
Global_WaveHeader.dwByteRate = AUDIO_SAMPLE_RATE *
(AUDIO_BITS_PER_SAMPLE / 8);
Global_WaveHeader.wBlockAlign = AUDIO_BITS_PER_SAMPLE / 8;
Global_WaveHeader.wBitsPerSample = AUDIO_BITS_PER_SAMPLE;
Global_WaveHeader.dwSubChunk2ID = 0x61746164;
Global_WaveHeader.dwSubChunk2Size = 0;

// write header to file
fwrite((void*)&Global_WaveHeader, sizeof(Global_WaveHeader), 1, pGlobal_WaveFile);

return 0;
}

DWORD CloseWaveFile()
{
// move back to the start of the file
rewind(pGlobal_WaveFile);

// store total data length in wave header
Global_WaveHeader.dwChunkSize += dwGlobal_TotalWaveDataLength;
Global_WaveHeader.dwSubChunk2Size += dwGlobal_TotalWaveDataLength;

// write the updated header
fwrite((void*)&Global_WaveHeader, sizeof(Global_WaveHeader), 1, pGlobal_WaveFile);

// close file handle
fclose(pGlobal_WaveFile);

return 0;
}

DWORD GenerateTone(DWORD dwFrequency, DWORD dwDuration)
{
DWORD dwSampleCount = 0;
DWORD dwTotalSize = 0;
double dPeriod = 0;
WORD *pwSampleList = NULL;

// set initial values
dwSampleCount = (AUDIO_SAMPLE_RATE * dwDuration) / 1000;
dwTotalSize = dwSampleCount * sizeof(WORD);
dPeriod = AUDIO_SAMPLE_RATE / (double)dwFrequency;

// allocate memory for audio samples
pwSampleList = (WORD*)malloc(dwTotalSize);
if(pwSampleList == NULL)
{
return 1;
}

// generate sine wave in the specified frequency
for(DWORD i = 0; i < dwSampleCount; i++)
{
// store current sample
pwSampleList[i] = (WORD)(32767 * sin(2 * 3.14159 * ((double)i / dPeriod)));
}

// write audio samples to file
fwrite((void*)pwSampleList, dwTotalSize, 1, pGlobal_WaveFile);

// increase total length
dwGlobal_TotalWaveDataLength += dwTotalSize;

// free temporary memory
free(pwSampleList);

return 0;
}

DWORD TransmitByte(BYTE bByte)
{
DWORD dwCurrBit = 0;
BYTE bBits[8];

// convert byte to bits
dwCurrBit = 128;
for(DWORD i = 0; i < 8; i++)
{
if((bByte & dwCurrBit) != 0)
{
bBits[i] = 1;
}
else
{
bBits[i] = 0;
}

dwCurrBit /= 2;
}

// transmit bits
for(i = 0; i < 8; i++)
{
if(bBits[i] == 0)
{
// 0
if(GenerateTone(BIT_TONE_FREQUENCY_OFF, TONE_LENGTH_MS) != 0)
{
return 1;
}
}
else
{
// 1
if(GenerateTone(BIT_TONE_FREQUENCY_ON, TONE_LENGTH_MS) != 0)
{
return 1;
}
}

// end of current bit
if(GenerateTone(BIT_TONE_FREQUENCY_NEXT, TONE_LENGTH_MS) != 0)
{
return 1;
}
}

return 0;
}

DWORD TransmitAudioData(BYTE *pData, DWORD dwLength)
{
char szFileName[512];

// set temporary filename
memset(szFileName, 0, sizeof(szFileName));
strncpy(szFileName, "temp_output.wav", sizeof(szFileName) - 1);
```

```
// initialise wave file
if(InitialiseWaveFile(szFileName) != 0)
{
return 1;
}

// generate start tone (to indicate start of transmission)
if(GenerateTone(TRANSMISSION_START_FREQUENCY, TONE_LENGTH_MS) != 0)
{
return 1;
}

// transmit all bytes
for(DWORD i = 0; i < dwLength; i++)
{
// process current byte
if(TransmitByte(*(BYTE*)(pData + i)) != 0)
{
return 1;
}
}

// generate end tone (to indicate end of transmission)
if(GenerateTone(TRANSMISSION_END_FREQUENCY, TONE_LENGTH_MS) != 0)
{
return 1;
}

// close wave file
CloseWaveFile();

// play sound
if(PlaySound(szFileName, NULL, SND_SYNC) == 0)
{
return 1;
}

// delete temporary file
if(DeleteFile(szFileName) == 0)
{
return 1;
}

return 0;
}
```

```
int main(int argc, char *argv[])
{
BYTE *pData = NULL;
DWORD dwLength = 0;

printf("AudioTransmit_Send - www.x86matthew.com\n\n");

if(argc != 2)
{
printf("Usage: %s [data]\n\n", argv[0]);

return 1;
}

// get data and length
pData = (BYTE*)argv[1];
dwLength = strlen((char*)pData);

printf("Sending data...\n");

// transmit data
if(TransmitAudioData(pData, dwLength) != 0)
{
printf("Error: Failed to send data\n");

return 1;
}

printf("Sent %u bytes successfully\n", dwLength);

return 0;
}
```

Receiver code:

```
#include <stdio.h>
#include <windows.h>
#include <math.h>

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

#define AUDIO_BITS_PER_SAMPLE 16
#define AUDIO_SAMPLE_RATE 8000

#define TRANSMISSION_START_FREQUENCY 2000
#define BIT_TONE_FREQUENCY_ON 600
#define BIT_TONE_FREQUENCY_OFF 800
```

```
#define TRANSMISSION_END_FREQUENCY 1500
#define BIT_TONE_FREQUENCY_NEXT 1200

#define TONE_THRESHOLD 500

DWORD dwGlobal_TransmissionStarted = 0;
DWORD dwGlobal_BitsReceived = 0;
BYTE bGlobal_RecvBits[8];
DWORD dwGlobal_LastToneType = 0;

// 1 second audio buffer
BYTE bGlobal_AudioBuffer[AUDIO_SAMPLE_RATE * sizeof(WORD)];

double CalculateToneMagnitude(short *pSamples, DWORD dwSampleCount, DWORD
dwTargetFrequency)
{
DWORD dwK = 0;
double dScalingFactor = 0;
double dW = 0;
double dSine = 0;
double dCosine = 0;
double dCoeff = 0;
double dQ0 = 0;
double dQ1 = 0;
double dQ2 = 0;
double dMagnitude = 0;

// set initial values for goertzel algorithm
dScalingFactor = (double)dwSampleCount / 2.0;
dwK = (DWORD)(0.5 + (((double)dwSampleCount * (double)dwTargetFrequency) /
AUDIO_SAMPLE_RATE));
dW = (2.0 * 3.14159 * (double)dwK) / (double)dwSampleCount;
dSine = sin(dW);
dCosine = cos(dW);
dCoeff = 2.0 * dCosine;

// process all samples
for(DWORD i = 0; i < dwSampleCount; i++)
{
// process current sample
dQ0 = (dCoeff * dQ1) - dQ2 + pSamples[i];
dQ2 = dQ1;
dQ1 = dQ0;
}
```

```
// calculate magnitude
dMagnitude = (double)sqrtf((float)((dQ1 * dQ1) + (dQ2 * dQ2) - (dQ1 * dQ2 * dCoeff)));
dMagnitude /= 100;

return dMagnitude;
}

DWORD ProcessSamples()
{
BYTE bRecvByte = 0;
DWORD dwSamplesRemaining = 0;
DWORD dwCurrSampleIndex = 0;
DWORD dwCurrChunkSize = 0;
short *pCurrSamplePtr = NULL;
double dFrequencySignal_TransmissionStart = 0;
double dFrequencySignal_TransmissionStart2 = 0;
double dFrequencySignal_TransmissionEnd = 0;
double dFrequencySignal_NextBit = 0;
double dFrequencySignal_BitOn = 0;
double dFrequencySignal_BitOff = 0;
double dStrongestTone = 0;
DWORD dwStrongestToneType = 0;

// process current samples
dwSamplesRemaining = sizeof(bGlobal_AudioBuffer) / sizeof(WORD);
dwCurrSampleIndex = 0;
for(;;)
{
// check of all samples have been processed
if(dwSamplesRemaining == 0)
{
// finished
break;
}

// calculate current chunk size (25ms)
dwCurrChunkSize = (AUDIO_SAMPLE_RATE / 40);
if(dwSamplesRemaining < dwCurrChunkSize)
{
dwCurrChunkSize = dwSamplesRemaining;
}

// get current sample position
pCurrSamplePtr = (short*)&bGlobal_AudioBuffer[dwCurrSampleIndex * sizeof(WORD)];
```

```
// check if a transmission is already in progress
if(dwGlobal_TransmissionStarted == 0)
{
// no transmission - check if a new one is starting
dFrequencySignal_TransmissionStart = CalculateToneMagnitude(pCurrSamplePtr,
dwCurrChunkSize, TRANSMISSION_START_FREQUENCY);
if(dFrequencySignal_TransmissionStart >= TONE_THRESHOLD)
{
// new data transmission detected
dwGlobal_BitsReceived = 0;
dwGlobal_LastToneType = TRANSMISSION_START_FREQUENCY;
dwGlobal_TransmissionStarted = 1;
}
}
else
{
// a transmission is already in progress - get next tone
dFrequencySignal_TransmissionStart = CalculateToneMagnitude(pCurrSamplePtr,
dwCurrChunkSize, TRANSMISSION_START_FREQUENCY);
dFrequencySignal_BitOn = CalculateToneMagnitude(pCurrSamplePtr, dwCurrChunkSize,
BIT_TONE_FREQUENCY_ON);
dFrequencySignal_BitOff = CalculateToneMagnitude(pCurrSamplePtr, dwCurrChunkSize,
BIT_TONE_FREQUENCY_OFF);
dFrequencySignal_NextBit = CalculateToneMagnitude(pCurrSamplePtr,
dwCurrChunkSize, BIT_TONE_FREQUENCY_NEXT);
dFrequencySignal_TransmissionEnd = CalculateToneMagnitude(pCurrSamplePtr,
dwCurrChunkSize, TRANSMISSION_END_FREQUENCY);

// check for the strongest tone
dStrongestTone = 0;
dwStrongestToneType = 0;
if(dFrequencySignal_TransmissionStart > dStrongestTone)
{
dStrongestTone = dFrequencySignal_TransmissionStart;
dwStrongestToneType = TRANSMISSION_START_FREQUENCY;
}
if(dFrequencySignal_BitOn > dStrongestTone)
{
dStrongestTone = dFrequencySignal_BitOn;
dwStrongestToneType = BIT_TONE_FREQUENCY_ON;
}
if(dFrequencySignal_BitOff > dStrongestTone)
{
dStrongestTone = dFrequencySignal_BitOff;
dwStrongestToneType = BIT_TONE_FREQUENCY_OFF;
```

```
}
if(dFrequencySignal_NextBit > dStrongestTone)
{
dStrongestTone = dFrequencySignal_NextBit;
dwStrongestToneType = BIT_TONE_FREQUENCY_NEXT;
}
if(dFrequencySignal_TransmissionEnd > dStrongestTone)
{
dStrongestTone = dFrequencySignal_TransmissionEnd;
dwStrongestToneType = TRANSMISSION_END_FREQUENCY;
}

// ensure at least one frequency is above the minimum threshold
if(dStrongestTone < TONE_THRESHOLD)
{
if(dwGlobal_BitsReceived != 0)
{
printf("\n** DATA CORRUPT - CANCELLED TRANSMISSION **\n");
}
dwGlobal_TransmissionStarted = 0;
}
else
{
// check if the tone has changed
if(dwStrongestToneType != dwGlobal_LastToneType)
{
// new tone detected
if(dwStrongestToneType == TRANSMISSION_START_FREQUENCY)
{
// found "transmission start" tone but already receiving data
if(dwGlobal_BitsReceived != 0)
{
printf("\n** DATA CORRUPT - CANCELLED TRANSMISSION **\n");
}

dwGlobal_TransmissionStarted = 0;
}
else if(dwStrongestToneType != BIT_TONE_FREQUENCY_NEXT)
{
// check if this is a data bit
if(dwStrongestToneType == BIT_TONE_FREQUENCY_ON || dwStrongestToneType ==
BIT_TONE_FREQUENCY_OFF)
{
if(dwGlobal_BitsReceived == 0)
{
```

```
// receiving first data bit
printf("** RECEIVING DATA **\n");
}

// check if the "on" or "off" bit tone is strongest
if(dwStrongestToneType == BIT_TONE_FREQUENCY_ON)
{
// received "1" bit
bGlobal_RecvBits[7 - (dwGlobal_BitsReceived % 8)] = 1;
}
else
{
// received "0" bit
bGlobal_RecvBits[7 - (dwGlobal_BitsReceived % 8)] = 0;
}

// wait for confirmation before reading next bit
dwGlobal_BitsReceived++;

// check if a full byte (8 bits) has been received
if(dwGlobal_BitsReceived % 8 == 0)
{
// convert bits to byte
bRecvByte = 0;
for(DWORD i = 0; i < 8; i++)
{
// convert current bit
bRecvByte |= (bGlobal_RecvBits[i] << i);
}

// print current byte
printf("%c", bRecvByte);
}
}
else if(dwStrongestToneType == TRANSMISSION_END_FREQUENCY)
{
// end of transmission
if(dwGlobal_BitsReceived != 0)
{
printf("\n** END OF TRANSMISSION (received %u bytes) **\n", dwGlobal_BitsReceived /
8);
}
dwGlobal_TransmissionStarted = 0;
}
}
```

```
// store last tone type
dwGlobal_LastToneType = dwStrongestToneType;
}
}
}

// update values for next chunk
dwSamplesRemaining -= dwCurrChunkSize;
dwCurrSampleIndex += dwCurrChunkSize;
}

return 0;
}

int main()
{
HWAVEIN hWave = NULL;
WAVEHDR WaveHeaderData;
WAVEFORMATEX WaveFormatData;
DWORD dwRetnVal = 0;
HANDLE hWaveEvent = NULL;

printf("AudioTransmit_Listen - www.x86matthew.com\n\n");

// create event
hWaveEvent = CreateEvent(NULL, 1, 0, NULL);
if(hWaveEvent == NULL)
{
return 1;
}

// set wave format data
memset((void*)&WaveFormatData, 0, sizeof(WaveFormatData));
WaveFormatData.wFormatTag = WAVE_FORMAT_PCM;
WaveFormatData.wBitsPerSample = AUDIO_BITS_PER_SAMPLE;
WaveFormatData.nChannels = 1;
WaveFormatData.nSamplesPerSec = AUDIO_SAMPLE_RATE;
WaveFormatData.nAvgBytesPerSec = AUDIO_BITS_PER_SAMPLE *
(AUDIO_BITS_PER_SAMPLE / 8);
WaveFormatData.nBlockAlign = AUDIO_BITS_PER_SAMPLE / 8;
WaveFormatData.cbSize = 0;

// open wave handle
if(waveInOpen(&hWave, WAVE_MAPPER, &WaveFormatData, (DWORD)hWaveEvent,
0, CALLBACK_EVENT | WAVE_FORMAT_DIRECT) != MMSYSERR_NOERROR)
{
```

```
CloseHandle(hWaveEvent);
return 1;
}

for(;;)
{
// set wave header data
memset((void*)&WaveHeaderData, 0, sizeof(WaveHeaderData));
WaveHeaderData.lpData = (LPSTR)bGlobal_AudioBuffer;
WaveHeaderData.dwBufferLength = sizeof(bGlobal_AudioBuffer);
WaveHeaderData.dwBytesRecorded = 0;
WaveHeaderData.dwUser = 0;
WaveHeaderData.dwFlags = 0;
WaveHeaderData.dwLoops = 0;

// prepare wave header
if(waveInPrepareHeader(hWave, &WaveHeaderData, sizeof(WaveHeaderData)) !=
MMSYSERR_NOERROR)
{
// error
CloseHandle(hWaveEvent);
waveInClose(hWave);

return 1;
}

// add wave input buffer
if(waveInAddBuffer(hWave, &WaveHeaderData, sizeof(WaveHeaderData)) !=
MMSYSERR_NOERROR)
{
// error
CloseHandle(hWaveEvent);
waveInClose(hWave);

return 1;
}

// reset event
ResetEvent(hWaveEvent);

// start recording
if(waveInStart(hWave) != MMSYSERR_NOERROR)
{
// error
CloseHandle(hWaveEvent);
waveInClose(hWave);
return 1;
}

// wait until recording has finished
for(;;)
{
// wait for event to fire
WaitForSingleObject(hWaveEvent, INFINITE);

// check if sample has finished recording
if(WaveHeaderData.dwFlags & WHDR_DONE)
{
// finished
break;
}
}

// unprepare wave header
if(waveInUnprepareHeader(hWave, &WaveHeaderData, sizeof(WAVEHDR)) !=
MMSYSERR_NOERROR)
{
// error
CloseHandle(hWaveEvent);
waveInClose(hWave);

return 1;
}

// process audio samples
if(ProcessSamples() != 0)
{
// error
CloseHandle(hWaveEvent);
waveInClose(hWave);

return 1;
}
}

// close wave handle
waveInClose(hWave);

// close event
CloseHandle(hWaveEvent);

return 0;
}
```