Revolution DSPTOOL.DLL

Version 1.00

The contents in this document are highly confidential and should be handled accordingly.

RVL-06-0034-001-B
Released: October 22, 2007

Contents

Code Examples

Revision History

Version	Date Revised	Item	Description
1.00	2006/03/01	-	First release by Nintendo of America Inc.

1 Description

DSPTOOL.DLL is a Win32 runtime dynamic linked library. It provides an API for encoding and decoding 16-bit PCM samples to and from the DSP-ADPCM compression format.

The DSP-ADPCM sample format provides (approximately) 3.5:1 compression and is proprietary to the Nintendo Revolution audio DSP. The audio DSP contains special hardware to decompress DSP-ADPCM samples for free.

This DLL is intended for developers who wish to create their own tools for generating and previewing DSP-ADPCM samples.

Note: This DLL is single-threaded.

2 Exported Functions

2.1 getBytesForAdpcmBuffer()

`getBytesForAdpcmBuffer()` calculates and returns the number of bytes needed to store a sample once it has been `DSP-ADPCM` encoded.

Note: The number of bytes will be a multiple of `DSP-ADPCM` frames, which are 8 bytes in length.

The actual length (in bytes) of the encoded sample may be less (see “getBytesForAdpcmSamples()” on page 6).

Use this function before encoding in order to allocate storage for the result.

```
U32 getBytesForAdpcmBuffer(u32 samples);
```

Arguments:

`u32 samples`

Number of (16-bit PCM) samples to be encoded.

Returns:

Number of bytes required to store a `DSP-ADPCM` encoded sample.

2.2 getBytesForAdpcmSamples()

`getBytesForAdpcmSamples()` returns the actual number of bytes occupied by a sample once it has been `DSP-ADPCM` encoded.

```
U32 getBytesForAdpcmSamples(u32 samples);
```

Arguments:

`u32 samples`

Number of (16-bit PCM) samples to be encoded.

Returns:

Actual length of the encoded sample (in bytes).

2.3 getBytesForPcmBuffer()

`getBytesForPcmBuffer()` calculates and returns the number of bytes needed to store a given sample once it has been decoded from `DSP-ADPCM` format.

```
U32 getBytesForPcmBuffer(u32 samples);
```

Arguments:

`u32 samples`

Number of samples to be decoded.

Returns:

Predicted length (in bytes) of a `DSP-ADPCM` sample to be decoded.

2.4 getBytesForPcmSamples()

`getBytesForPcmSamples()` returns the number of bytes to copy for the user application-specified number of samples.

```
U32 getBytesForPcmSamples(u32 samples);
```

Arguments:

`u32 samples`

Number of samples.

Returns:

Length (in bytes) for specified number of samples.

2.5 getSampleForAdpcmNibble()

`getSampleForAdpcmNibble()` returns the zero-based sample number for corresponding `ADPCM` nibble.

```
U32 getSampleForAdpcmNibble(u32 nibble);
```

Arguments:

`u32 nibble`

`ADPCM` nibble offset, including frame headers.

Returns:

Zero-based sample index for corresponding `ADPCM` sample.

2.6 getBytesForAdpcmInfo()

`getBytesForAdpcmInfo()` returns the size, in bytes, of the `ADPCMINFO` structure.

```
U32 getBytesForAdpcmInfo(u32 samples);
```

Arguments:

None.

Returns:

`sizeof(ADPCMINFO)`.

2.7 getNibbleAddress()

`getNibbleAddress()` calculates and returns the corresponding nibble for a sample at a given offset. For example, the 100th sample (counting from zero) corresponds to the 116th nibble (also counting from zero).

Note: The calculated nibble address already accounts for the 2-nibble frame headers of the `DSP-ADPCM` compression format.

Another example: the zeroth sample corresponds to nibble offset 2, which is the third nibble (counting from zero).

```
U32 getNibbleAddress(u32 samples);
```

Arguments:

`u32 samples`

The offset of 16-bit PCM address. Zero is the first sample; 100 is the 101st sample.

Returns:

Corresponding nibble *address*.

2.8 getLoopContext()

`getLoopContext()` returns the `DSP-ADPCM` loop context at a given offset within a sample.

```
typedef struct
{

        // start context
        s16 coef[16];
        u16 gain;
        u16 pred_scale;
        s16 yn1;
        s16 yn2;

        // loop context
        u16 loop_pred_scale;
        s16 loop_yn1;
        s16 loop_yn2;

} ADPCMINFO;

void getLoopContext(
        u8              *src,           // location of ADPCM buffer in RAM
        ADPCMINFO       *cxt,           // location of adpcminfo
        u32             samples         // samples to desired context
);
```

Arguments:

`u8 *src`

This is a pointer to a buffer containing a `DSP-ADPCM` encoded sample.

`ADPCMINFO *cxt`

This is a pointer to an `ADPCMINFO` construct. The `getLoopContext()` function will place the loop context information into this construct.

`u32 samples`

This is the offset, in *samples*, of the loop start position.

Note: This loop start position is *not* a nibble address; it is the raw sample number at which a loop starts—in other words, it is the first sample to be played for the loop. Therefore, if the loop begins at the very first sample, the offset is zero. If the loop begins at the 101st sample, the offset is 100.

Returns:

None.

2.9 encode

`encode()` will compress a 16-bit PCM sample into the `DSP-ADPCM` format.

```
typedef struct
{

       // start context
       s16 coef[16];
       u16 gain;
       u16 pred_scale;
       s16 yn1;
       s16 yn2;

       // loop context
       u16 loop_pred_scale;
       s16 loop_yn1;
       s16 loop_yn2;

} ADPCMINFO;

void encode(
       s16         *src,   // location of source samples (16bit PCM signed little endian)
       u8          *dst,   // location of destination buffer
       ADPCMINFO   *cxt,   // location of adpcm info
       u32         samples // number of samples to encode
);
```

Arguments:

`s16 *src`

This is a pointer to a buffer containing a signed, little-endian, 16-bit PCM sample.

`u8 *dst`

This is a pointer to a buffer which will contain the DSP-ADPCM encoded output.

Note: Your application must allocate storage for this buffer. The size of this buffer must be calculated with `getBytesForAdpcmBuffer()`.

`ADPCMINFO *cxt`

Pointer to a structure of type `ADPCMINFO`. The `encode()` function will store the sample's coefficients and context information in this structure.

Note: Several of the parameters (`gain`, `yn1`, and `yn2`) will always be zero. These parameters are included in the structure to emphasize the fact that the corresponding registers in the DSP decoder hardware must be cleared before starting the decode process. Refer to the *Audio Library (AX)* document set in this guide for more details.

Note also that the loop context parameters will always be zero after calling `encode()`. To set these values, you must call `getLoopContext()` if the sample is indeed looped.

`u32 samples`

Number of samples to encode.

Returns:

None.

2.10 decode

`decode()` is used to decode a `DSP-ADPCM` sample into signed, little-endian, 16-bit PCM data.

```
typedef struct
{

       // start context
       s16 coef[16];
       u16 gain;
       u16 pred_scale;
       s16 yn1;
       s16 yn2;

       // loop context
       u16 loop_pred_scale;
       s16 loop_yn1;
       s16 loop_yn2;

} ADPCMINFO;

void encode(
       u8          *src,   // location of encoded source samples
       s16         *dst,   // location of destination buffer (16 bits / sample)
       ADPCMINFO   *cxt,   // location of adpcm info
       u32         samples // number of samples to decode
);
```

Arguments:

`u8 *src`

Pointer to a buffer containing `DSP-ADPCM` sample data.

`S16 *dst`

Pointer to a buffer which has been allocated for storing the uncompressed PCM data. The size of this buffer may be calculated by calling `getBytesForPcmBuffer()`.

`ADPCMINFO *cxt`

Pointer to a structure of type `ADPCMINFO`. This structure must contain the coefficient and initial state data which corresponds to the sample being decoded.

`u32 samples`

Number of samples to decode.

Returns:

None.

3 Usage

DSPTOOL.DLL is intended for use in Win32 applications. Here are some examples.

3.1 Initialization

The calling application must invoke DSPTOOL.DLL according to Win32 conventions.

Code 3–1 Invoking DSPTOOL.DLL

```
static HINSTANCE hDll;

typedef u32 (*lpFunc1)(u32);
typedef u32 (*lpFunc2)(void);
typedef void (*lpFunc3)(s16*, u8*, ADPCMINFO*, u32);
typedef void (*lpFunc4)(u8*, s16*, ADPCMINFO*, u32);
typedef void (*lpFunc5)(u8*, ADPCMINFO*, u32);

lpFunc1 getBytesForAdpcmBuffer;
lpFunc1 getBytesForAdpcmSamples;
lpFunc1 getBytesForPcmBuffer
lpfunc1 getBytesForPcmSamples;
lpfunc1 getSampleForAdpcmNibble;
lpfunc1 getNibbleAddress;
lpFunc2 getBytesForAdpcmInfo;
lpFunc3 encode;
lpFunc4 decode;
lpFunc5 getLoopContext;


/*---------------------------------------------------------------------------*/
void clean_up(void)
{
    if (hDll)
       FreeLibrary(hDll);
}


/*---------------------------------------------------------------------------*/
int getDll(void)
{
    hDll = LoadLibrary("dspadpcm.dll");

    if (hDll)
    {
        if (!(getBytesForAdpcmBuffer =
                (lpFunc1)GetProcAddress(
                            hDll,
                            "getBytesForAdpcmBuffer"
                            ))) return 1;

        if (!(getBytesForAdpcmSamples =
                (lpFunc1)GetProcAddress(
                            hDll,
                            "getBytesForAdpcmSamples"
                            ))) return 1;

        if (!(getBytesForPcmBuffer =
                (lpFunc1)GetProcAddress(
                            hDll,
                            "getBytesForPcmBuffer"
                            ))) return 1;

        if (!(getBytesForPcmSamples =
```

```
                (lpFunc1)GetProcAddress(
                            hDll,
                            "getBytesForPcmSamples"
                            ))) return 1;

        if (!(getNibbleAddress =
                (lpFunc1)GetProcAddress(
                            hDll,
                            "getNibbleAddress"
                            ))) return 1;

        if (!(getSampleForAdpcmNibble =
                (lpFunc1)GetProcAddress(
                            hDll,
                            "getSampleForAdpcmNibble"
                            ))) return 1;

        if (!(getBytesForAdpcmInfo =
                (lpFunc2)GetProcAddress(
                            hDll,
                            "getBytesForAdpcmInfo"
                            ))) return 1;

        if (!(encodeLittleEndian =
                (lpFunc3)GetProcAddress(
                            hDll,
                            "encode"
                            ))) return 1;

        if (!(encodeBigEndian =
                (lpFunc4)GetProcAddress(
                            hDll,
                            "decode"
                            ))) return 1;

        if (!(getLoopContext =
                (lpFunc5)GetProcAddress(
                            hDll,
                            "getLoopContext"
                            ))) return 1;

        return(0);
    }

    return(1);
}


/*---------------------------------------------------------------------------*/
void main (void)
{
       if (getDll)
       {
               clean_up();
               exit(1);
       }

// do stuff here

       clean_up();
}
```

3.2 Encoding

The encoding function assumes 16-bit, little-endian PCM data (as used by Windows *.WAV files). If an application wishes to encode big-endian data, it must reverse the endianness prior to encoding.

Code 3–2 Encoding (pseudocode example)

```
//... loaded DCPADPCM.DLL


//... put some PCM buffer in memory, reverse the endian if you have to

u8 *adpcm = (u8*)malloc(getBytesForAdpcmBuffer(samplesToEncode));

if (adpcm)
{
        ADPCMINFO adpcminfo;

        // ok.. lets encode it!
        encode(source, adpcm, &adpcminfo, samplesToEncode);

        // get ADPCM loop context if sample is looped
        if (samplesToLoopStart)
                getLoopContext(adpcm, &adpcminfo, samplesToLoopStart);

        // see how many bytes to store the encoded data stream
        u32 nBytesToStore = getBytesForAdpcmSamples(samplesToEncode);

         ... store encoded ADPCM data stream to file

         ... store ADPCM context to file

         u32 nibbleStartOffset           = getNibbleAddress(0);
         u32 nibbleLoopStartOffset       = getNibbleAddress(samplesToLoopStart);
         u32 nibbleEndAddress            = getNibbleAddress(samplesToEncode);

         ... store nibble addressing to file

         // don't need the ADPCM buffer anymore
         free(adpcm);
}

... continue
```

3.3 Decoding

Decoding results in 16-bit, little-endian PCM output data.

Code 3–3 Decoding (pseudocode example)

```
... loaded DCPADPCM.DLL


... put some ADPCM buffer and corresponding ADPCMINFO in memory,
    ADPCM is byte ordered.. not endian sensitive.

s16 *pcm = (u8*)malloc(getBytesForPcmBuffer(samplesToDecode));

if (pcm)
{
        // ok.. lets decode it!
        decode(source, pcm, adpcminfo, samplesToDecode);

        ... store decoded PCM buffer to file

        // don't need the PCM buffer anymore
        free(pcm);
}

... continue
```