Battle Animation File Format

File format discovered/decoded by me, L. Spiro.
This file was written by me, L. Spiro, as can be seen by the proper grammar and perfect spelling.

Part I: Structures

1. `FF7FrameHeader`
2. `FF7FrameMiniHeader`
3. `FF7ShortVec`
4. `FF7FrameBuffer`

Part II: Functions and Format

1. `GetBitsFixed()`

A. Format

2. `GetDynamicFrameOffsetBits()`
3. `GetEncryptedRotationBits()`

Part III: Putting it All Together

1. `LoadFrames()`
2. A Sample Loop

Part IV: Qhimm's Input

Part I: Structures

There are 4 basic structures we will use in decoding the file format.
The header of animation data has been considered to be composed of 3 DWORD's, 3 WORD's, and one BYTE, however this is not how the header is really intended to be, despite being aligned correctly.

Battle animation files start with a DWORD which tells us how many animations are in the file. This number includes the special animations which are not actually animations at all. Although I have not yet decoded them, I suspect these are sets of keys for actual animation sets; keys that call scripted actions or tell the engine to print the damage numbers.

Cloud's battle animation file (rtda) has 94 (0x 5E) animations in it.
After this number begins each animation.
Each animation begins with a 12-byte header (3 DWORD's) we will call "`FF7FrameHeader`".

To get from one animation to the next, start at offset 0x04 in the animation file and begin reading these headers. For each header, skip "`FF7FrameHeader.dwChunkSize`" bytes until you get to the index of the file you want to load. When skipping, remember to skip starting at the end of the "`FF7FrameHeader`" header.
I mentioned a type of special animation data set that is in the header file.
These data sets, when filled with the "`FF7FrameHeader`" header, will have a "`dwChunkSize`" less than eleven, we skip them by jumping over the next 8 bytes that follow.

First, the two main headers in the animation file.

1.
```
typedef struct FF7FrameHeader {
      DWORD       dwBones;          // Bones in the model + 1.    0x00
      DWORD       dwFrames;         // Frames in the animation.   0x04
      DWORD       dwChunkSize;      // Size of the animation set. 0x08
} * PFF7FrameHeader;                // Size = 12 bytes.
```

2.
```
typedef struct FF7FrameMiniHeader {
      SHORT       sBones;     // Bones in the animation.          0x00
      SHORT       sSize;      // Size of the animation data.      0x02
      BYTE        bKey;       // A key flag used for decoding.    0x04
} * PFF7FrameMiniHeader;      // Size = 5 bytes.
```

Most of these members are straight-forward, however there is a very special and VERY important member in the "`FF7FrameMiniHeader`" structure called "`bKey`".
This is used for every rotation-decoding scheme (but one). It determines, essentially, the precision of the rotations and the deltas that follow in successive frames.
The value of "`bKey`" can only be 0, 2, or 4; the equation (`12 - bKey`) is used to determine the length of each raw (uncompressed) rotation.
After decompression, every rotation must be 12 bits, giving it a range from 0 to 4095.
But if "`bKey`" is 4, for example, then that means uncompressed rotations are stored as 8 bits, which gives them a range from 0 to 255. How is this fixed? After the 8 bits are read, they are then shifted left (up) by "`bKey`". This will place them at 12 bits, but with decreased accuracy.
This loss in accuracy is acceptable since rotations work as deltas and usually only change by a small amount.
Most large rotation deltas are things that are spinning, such as the blades on Aero Combatant. These cases are always a nice round number that can be handled with lower precision (in the case of Aero Combatant, it is 90 degrees even).

Now the code to skip to any animation, by index, where "`iTarget`" is the index. This code assumes you have already opened the animation file (`hFile`) and you have skipped pasted the first 4 bytes.

```
FF7FrameHeader    fhHeader;
DWORD             dwBytesRead;
for ( int I = 0; I < iTarget; I++ ) {
      if ( !ReadFile( hFile,
            &fhHeader, sizeof( fhHeader ),
            &dwBytesRead, NULL ) ) {

            CloseHandle( hFile );
            return false;
      }
      if ( fhHeader.dwChunkSize < 11 ) {  // If this is a special
                                          //    chunk, skip it (it
                                          //    is counted as part
                                          //    of the total in the
                                          //    file).
            if ( SetFilePointer( hFile, 8, NULL, FILE_CURRENT ) ==
                                            INVALID_SET_FILE_POINTER ) {
                  CloseHandle( hFile );
                  return false;
            }
            continue;                     // Go on to the next
                                          //    animation.
      }
      // Skip this animation set, whose size is determined by
      //    fhHeader.dwChunkSize.
      if ( SetFilePointer( hFile, fhHeader.dwChunkSize, NULL,
            FILE_CURRENT ) == INVALID_SET_FILE_POINTER ) {
            CloseHandle( hFile );
            return false;
      }
}
// Once we come to this point, we are at the very first byte of the
//    animation we want to load.  Let's store it into a BYTE array.
if ( !ReadFile( hFile,
      &fhHeader, sizeof( fhHeader ),
      &dwBytesRead, NULL ) ) {

      CloseHandle( hFile );
      return false;
}
BYTE * pbBuffer = new BYTE[fhHeader.dwChunkSize];
// Now pbBuffer holds the actual animation data, including the 5-byte
//    "FF7FrameMiniHeader" header.
```

We now have the animation we want loaded into a BYTE array (remember to `delete` it later).

Now let's look at the other structures we will use.

3.

```
typedef struct FF7ShortVec {
      SHORT sX, sY, sZ;       // Signed short versions.     0x00
      INT   iX, iY, iZ;       // Integer representation.    0x06
```

```
      FLOAT fX, fY, fZ;       // Float version after math.  0x12
} * PFF7ShortRot;             // Size = 30 bytes.
```

Each rotation goes through 3 forms. Firstly, everything is stored as 2-byte SHORT's. These SHORT's are stored from 0 to 4096, where 0 = 0 degrees and 4096 = 360 degrees. This is the equation to convert one of these SHORT's into degrees: (SHORT / 4096 * 360). Each frame is based off the previous frame, using the SHORT value as its basis. Each SHORT is converted to an INT, which is the exact same as the SHORT version, except always positive.
Finally, the FLOAT gets filled with the final value, using the INT version as its base.
So, the sequence is:
First frame…

Read X bits and store as a signed SHORT.
Convert the SHORT to the INT field, adding 0x1000 if negative.
Convert to FLOAT using (INT / 4096 * 360). Apply this FLOAT to your model.

Next frame…

Read X bits, and add them to the SHORT value from last frame.
Convert the SHORT to the INT field, adding 0x1000 if negative.
Convert to FLOAT using (INT / 4096 * 360). Apply this FLOAT to your model.

Repeat…

This structure is for one bone rotation.
To load an entire frame's work of bones, we need this structure:

4.

```
typedef struct FF7FrameBuffer {
      DWORD             dwBones;
      FF7ShortVec       svPosOffset;
      FF7ShortVec *     psvRots;

      FF7FrameBuffer() {
            dwBones = 0;
            psvRots = NULL;
      }
      ~FF7FrameBuffer() {
            dwBones = 0;
            delete [] psvRots;
            psvRots = NULL;
      }

      VOID  SetBones( DWORD dwTotal ) {
            // Delete the old.
            dwBones = 0;
            delete [] psvRots;

            // Create the new.
            psvRots = new FF7ShortVec[dwTotal];
            if ( psvRots != NULL ) { dwBones = dwTotal; }
      }
} * PFF7FrameBuffer;
```

This structure will allocate enough memory for one frame of rotations. Simply call "`FF7FrameBuffer.SetBones`" with the number of bones in your animation.

Part II: Functions and Format

First, we need a way to read bits from the BYTE array we have stored.
This is a basic bit-reading function. It reads "`dwTotalBits`" from "`pbBuffer`" starting at the "`dwStartBit`"'th bit.

1.

```
INT GetBitsFixed( BYTE * pbBuffer, DWORD &dwStartBit,
            DWORD dwTotalBits ) {
      INT iReturn = 0;

      for ( DWORD I = 0; I < dwTotalBits; I++ ) {
            iReturn <<= 1;

            __asm mov eax, dwStartBit
            __asm mov eax, [eax]
            __asm cdq
            __asm and edx, 7
            __asm add eax, edx
            __asm sar eax, 3
            __asm mov ecx, pbBuffer
            __asm xor edx, edx
            __asm mov dl, byte ptr ds:[ecx+eax]
            __asm mov eax, dwStartBit
            __asm mov ecx, [eax]
            __asm and ecx, 7
            __asm mov eax, 7
            __asm sub eax, ecx
            __asm mov esi, 1
            __asm mov ecx, eax
            __asm shl esi, cl
            __asm and edx, esi
            __asm test edx, edx
            __asm je INCBIT
            iReturn++;
INCBIT:     dwStartBit++;
      }

      // Force the sign bit to extend across the 32-bit boundary.
      iReturn <<= (0x20 - dwTotalBits);
      iReturn >>= (0x20 - dwTotalBits);
      return iReturn;
}
```

Now that we can read the bits in the buffer we have made, it's time to know what we're doing!

A.

The animation data begins with one full frame that is uncompressed, but stored in one of 3 ways. Every frame after that is compressed, but compressed in one of three ways; one way can be decoded using the same method as on the first frame, which is why sometimes the second, third, and even fourth frames can be decoded using the same method as was used on the first frame.

First Frame:

Remember that we stored our animation buffer with a 5-byte "`FF7FrameMiniHeader`" at the beginning of it? We need this header now!

```
PFF7FrameMiniHeader pfmhMiniHeader = (PFF7FrameMiniHeader)pbBuffer;
```

After this cast, "`pfmhMiniHeader->bKey`" will contain a number, either 0, 2, or 4.
Each rotation is stored in (12 - "`pfmhMiniHeader->bKey`") bits. This mean either 12, 10, or 8, respectively.
This is important to know.
But first, there is offset data. Each offset is 16 bits (a signed SHORT).
In the first frame of Cloud's first animation (rtda), these bytes are 00 00 FE 2E 00 00.
16 bits × 3 = 48 bits, or 6 bytes.
To get these bits, we first need to make a pointer point to the correct location.
"`pbBuffer`" points 5 bytes before this data, so let's make a pointer that points to this data directly.

```
BYTE * pbAnimBuffer = &pbBuffer[5];
```

When we use "`GetBitsFixed()`" to get the bits.

```
DWORD dwBitStart = 0;   // The bits at which to begin reading in the
                        //    stream.
SHORT sX = GetBitsFixed( pbAnimBuffer, dwBitStart, 16 );
SHORT sY = GetBitsFixed( pbAnimBuffer, dwBitStart, 16 );
SHORT sZ = GetBitsFixed( pbAnimBuffer, dwBitStart, 16 );
```

After doing this, we have each of the three offsets, 0, -466, and 0.
The Y (-466) is always stored as its inverse, but for now we don't worry about that.

The first frame is uncompressed, but it could be 12, 10, or 8 bits per rotation.
How do we know? "`pfmhMiniHeader->bKey`"!

For each bone, there are 3 rotations. So, for each bone, we do this:

```
SHORT sRotX = GetBitsFixed( pbAnimBuffer, dwBitStart,
      12 - pfmhMiniHeader->bKey );
SHORT sRotY = GetBitsFixed( pbAnimBuffer, dwBitStart,
      12 - pfmhMiniHeader->bKey );
SHORT sRotZ = GetBitsFixed( pbAnimBuffer, dwBitStart,
      12 - pfmhMiniHeader->bKey );
// We have each rotation, but for the equation to work, the range
//    must always be from 0 to 4095.  If we got 8 bytes, for example,
//    the range would only be from 0 to 255, so here we need to fix
```

```
//    this.
sRotX <<= pfmhMiniHeader->bKey;
sRotY <<= pfmhMiniHeader->bKey;
sRotZ <<= pfmhMiniHeader->bKey;
```

The first rotation is always 0, 0, 0. This is the root rotation and is not actually counted as part of the bone network of the character.

The first frame is easy.
Remember that all frames after are stored as relative offsets from the frame before it.
The offsets are relative to the SHORT values of the previous frame rather than the FLOAT or INT values.

Each frame begins with the three position offset values, but in the second frame and after, they can be either 7 or 16 bits.
To determine if which they are, we first get one bit. If that bit is 0, then the following 7 bits are the actual value of the offset (signed).
If it is 1, then the next 16 bits are the value of the offset. In total, the offsets will be either 8 or 17 bits.

Now the code to perform this operation.

2.

```
SHORT GetDynamicFrameOffsetBits( BYTE * pBuffer, DWORD &dwBitStart ) {
      DWORD dwFirstByte, dwConsumedBits, dwBitsRemainingToNextByte,
      dwTemp;
      SHORT sReturn;
      __asm {
            mov eax, dwBitStart
            mov eax, [eax]
            cdq
            and edx, 7
            add eax, edx
            sar eax, 3
            mov dwFirstByte, eax
            mov ecx, dwBitStart
            mov edx, [ecx]
            and edx, 7
            mov dwConsumedBits, edx
            mov eax, 7
            sub eax, dwConsumedBits
            mov dwBitsRemainingToNextByte, eax
            mov ecx, pBuffer
            add ecx, dwFirstByte          // Go to the first byte that
                                          //    has the bit where we
                                          //    want to begin.
            xor edx, edx
            mov dl, byte ptr ds:[ecx]
            shl edx, 8
            mov eax, pBuffer
            add eax, dwFirstByte
```

```
            xor ecx, ecx
            mov cl, byte ptr ds:[eax+1]
            or edx, ecx
            mov dwTemp, edx
            mov ecx, dwBitsRemainingToNextByte
            add ecx, 8
            mov edx, 1
            shl edx, cl
            mov eax, dwTemp
            and eax, edx
            test eax, eax
            jnz SeventeenBits

EightBits :
            mov ecx, dwConsumedBits
            add ecx, 1
            mov edx, dwTemp
            shl edx, cl
            movsx eax, dx
            sar eax, 9
            mov sReturn, ax
            mov ecx, dwBitStart     //
            mov edx, [ecx]          //
            add edx, 8              //
            mov eax, dwBitStart     //
            mov [eax], edx          // Increase dwBitStart by 0x8 (8).
            jmp End

SeventeenBits :
            mov ecx, dwTemp
            shl ecx, 8
            mov edx, pBuffer
            add edx, dwFirstByte
            xor eax, eax
            mov al, byte ptr ds:[edx+2]
            or ecx, eax
            mov dwTemp, ecx
            mov ecx, dwConsumedBits
            add ecx, 1
            mov edx, dwTemp
            shl edx, cl
            shr edx, 8
            mov sReturn, dx
            mov eax, dwBitStart     //
            mov ecx, [eax]          //
            add ecx, 0x11           //
            mov edx, dwBitStart     //
            mov [edx], ecx          // Increase dwBitStart by 0x11
                                    //    (17).

End :
      }
      return sReturn;
}
```

After the first frame, we know that the positional offsets immediately follow.

So to get the positional deltas for the next frame, we would do this:

```
SHORT sDeltaX = GetDynamicFrameOffsetBits( pbAnimBuffer, dwBitStart );
SHORT sDeltaY = GetDynamicFrameOffsetBits( pbAnimBuffer, dwBitStart );
SHORT sDeltaZ = GetDynamicFrameOffsetBits( pbAnimBuffer, dwBitStart );
```

Now we have the change from the previous frame. In our "`FF7ShortVec`" structure, these are the SHORT values. To get the position of this frame, we add these offsets to the last frame's position.
If "`I`" is this frame and "`I-1`" is the last frame, we could do something like this:

```
FF7FrameBuffer[I].svPosOffset.sX = FF7FrameBuffer[I-1].svPosOffset.sX +
sX;
FF7FrameBuffer[I].svPosOffset.sY = FF7FrameBuffer[I-1].svPosOffset.sY +
sY;
FF7FrameBuffer[I].svPosOffset.sZ = FF7FrameBuffer[I-1].svPosOffset.sZ +
sZ;
```

Now all that is left is to decode the rotations.
Rotations change size in multiple ways.
There is no single simple way to express them.

They are, however, always at least one bit long.
The first bit is a flag. If 0, the rotational change is 0, and that is the end of that rotation.
If it is not 0, then we must get the next 3 bits.
The next 3 bits can tell us to do one of three things.
If the resulting 3-bit signed value is 0, then the rotation delta is (`-1 << pfmhMiniHeader->bKey`). This is the smallest possible decrement for the given precision (remember that precision is based off "`pfmhMiniHeader->bKey`".
If the 3-bit value is 7, then we treat the rotation the same way as we do in the first frame, where we read (`12-pfmhMiniHeader->bKey`) bits, then shift left by "`pfmhMiniHeader->bKey`".

The complicated cases are 1 through 6.
If the 3-bit value is from 1 to 6, then this indicates the number of bits in the rotation delta.
For our example, let's assume the 3-bit value was 4.
This means we need to read the next 4 bits from the stream. These 4 bits will be the animation delta, but we actually have to handle them before we can call it final.
The first bit of this new data is a sign bit which determines if the value is below 0.
If it is below zero, we must subtract from that number (1 << ([Number of Bits] – 1)).
So, if the 3-bit value was 4, and we read 4 bits from the stream, and the resulting value was negative, we would subtract from that value (1 << 3), or 8.
If the 4-bit value is positive, we **add** (1 << ([Number of Bits] – 1)) to it.
After we handle the positive and negative cases, we have to adjust for our precision again.
So, we shift left the resulting value by "`pfmhMiniHeader->bKey`".
This is all shown in the code below.

3.

```
SHORT GetEncryptedRotationBits( BYTE * pBuffer, DWORD &dwBitStart,
            INT iKeyBits ) {
      DWORD dwNumBits, dwType;
      INT iTemp;
      SHORT sReturn;
      // Check the first bit.
      INT iBits = GetBitsFixed( pBuffer, dwBitStart, 1 );
      __asm mov eax, iBits    // If the first bit is 0, return 0
                              //    and continue. It is not necessary
                              //    to mov iBits into EAX, but I do it
                              //    anyway.
      __asm test eax, eax
      __asm jnz SecondTest
      __asm jmp ReturnZero    // Return 0

SecondTest :
      // Otherwise continue by getting the next 3 bits.
      iBits = GetBitsFixed( pBuffer, dwBitStart, 3 );
      __asm mov eax, iBits
      __asm and eax, 7
      __asm mov dwNumBits, eax
      __asm mov ecx, dwNumBits
      __asm mov dwType, ecx   // dwType = ecx = dwNumBits = eax =
                              //    (iBits & 7).
                              //    When we get to the case, all of
                              //    these values are the same.
      __asm cmp dwType, 7
      __asm ja ReturnZero     // Is dwType above 7?  If so, return 0.
                              //    This can never actually happen.

      // Otherwise, use it in a switch case.
      switch ( dwType ) {
            case 0 : {
                  __asm or eax, 0xFFFFFFFF      // After this, EAX will
                                                //    always be -1.
                  __asm mov ecx, iKeyBits
                  __asm shl eax, cl             // Shift left by
                                                //    precision.
                                                //    (-1 << iKeyBits)
                  __asm mov sReturn, ax         // Return that number.
                  __asm jmp End
            }
            case 1 : {}
            case 2 : {}
            case 3 : {}
            case 4 : {}
            case 5 : {}
            case 6 : {
                  // Get a number of bits equal to the case switch (1,
                  //    2, 3, 4, 5, or 6).
                  iTemp = GetBitsFixed( pBuffer, dwBitStart,
                        dwNumBits );
                  __asm mov eax, iTemp
                  __asm cmp iTemp, 0
```

```
                  __asm jl IfLessThanZero
                  // If greater than or equal to 0…
                  __asm mov ecx, dwNumBits      // dwNumBits = (iBits &
                                                //    7) from before.
                  __asm sub ecx, 1              // dwNumBits - 1.
                  __asm mov eax, 1
                  __asm shl eax, cl             // (1 << (dwNumBits -
                                                //    1)).
                  __asm mov ecx, iTemp
                  __asm add ecx, eax            // iTemp += (1 <<
                                                //    (dwNumBits - 1)).
                  __asm mov iTemp, ecx
                  __asm jmp AfterTests
                  // If less than 0…
IfLessThanZero :
                  __asm mov ecx, dwNumBits      // dwNumBits = (iBits &
                                                //    7) from before.
                  __asm sub ecx, 1              // Decrease it by 1.
                  __asm mov edx, 1
                  __asm shl edx, cl             // Shift "1" left by
                                                //    (dwNumBits - 1).
                  __asm mov eax, iTemp          // iTemp still has the
                                                //    bits we read
                                                // from before.
                  __asm sub eax, edx            // iTemp - (1 <<
                                                //    (dwNumBits - 1))
                  __asm mov iTemp, eax

                  // Now, whatever we set on iTemp, we need to shift it
                  //    up by the precision value.
AfterTests :
                  __asm mov eax, iTemp
                  __asm mov ecx, iKeyBits
                  __asm shl eax, cl             // iTemp <<= iKeyBits
                  __asm mov sReturn, ax
                  __asm jmp End
            }

            case 7 : {
                  // Uncompressed bits.  Use standard decoding.
                  iTemp = GetBitsFixed( pBuffer, dwBitStart,
                        12 - iKeyBits );
                  __asm mov ecx, iKeyBits
                  __asm shl eax, cl             // iTemp <<= iKeyBits.
                  __asm mov sReturn, ax
                  __asm jmp End
            }

      }


ReturnZero :
      __asm xor ax, ax
      __asm mov sReturn, ax

End :
```

```
	return sReturn;
}
```

Part III: Putting it All Together

To make life easy, let's use one function to load an entire frame at a time.
This function will load an entire frame into a "`FF7FrameBuffer`" structure.
The function will return the bit position where the next frame will begin.
After the function returns, we must translate the rotational INT values to their FLOAT forms (although the function can be modified to do this part itself).
This function will be called in a loop for every frame in the rotation.

1.

```
DWORD LoadFrames( PFF7FrameBuffer pfbFrameBuffer,
		INT iBones,
		INT iBitStart,
		BYTE * pbAnimBuffer ) {
	// Get backups of the information we need.
	DWORD		dwThisBitStart	= iBitStart;
	INT		iThisBones	= iBones;
	BYTE *		pbThisBuffer	= pbAnimBuffer;

	PFF7FrameMiniHeader pfmhMiniHeader =
		(PFF7FrameMiniHeader)pbAnimBuffer;
	SHORT sSize = pfmhMiniHeader->sSize;
	BYTE  bKeyBits = pfmhMiniHeader->bKey;

	// Skip the first 5 bytes because they are part of the frame
header.
	pbThisBuffer += sizeof( FF7FrameMiniHeader );

	if ( iBitStart == 0 ) { // First frame?
		// The first frame is uncompressed and each value is the
		//    actual rotation.
		pfbFrameBuffer->svPosOffset.sX = GetBitsFixed(
pbThisBuffer, dwThisBitStart, 16 ); // Always 16 bits here.
		pfbFrameBuffer->svPosOffset.sY = GetBitsFixed(
pbThisBuffer, dwThisBitStart, 16 );
		pfbFrameBuffer->svPosOffset.sZ = GetBitsFixed(
pbThisBuffer, dwThisBitStart, 16 );

		// This function will set the FLOAT values for the
		//    positions.
		// Any scaling that needs to be done would be done here.
		pfbFrameBuffer->svPosOffset.fX = (FLOAT)pfbFrameBuffer-
>svPosOffset.sX;
		pfbFrameBuffer->svPosOffset.fY = (FLOAT)pfbFrameBuffer-
>svPosOffset.sY;
		pfbFrameBuffer->svPosOffset.fZ = (FLOAT)pfbFrameBuffer-
>svPosOffset.sZ;
```

```
            for ( int I = 0; I < iThisBones; I++ ) {
                  // Now get each bone rotation (the first bone is
                  //    actually the root, not part of the skeleton).
                  // During the first frame, the rotations are always
                  //    (12 - bKeyBits).
                  // We shift by bKeyBits to align it to 12 bits.
                  pfbFrameBuffer->psvRots[I].sX = (GetBitsFixed(
pbThisBuffer, dwThisBitStart, 12 - bKeyBits ) << bKeyBits);
                  pfbFrameBuffer->psvRots[I].sY = (GetBitsFixed(
pbThisBuffer, dwThisBitStart, 12 - bKeyBits ) << bKeyBits);
                  pfbFrameBuffer->psvRots[I].sZ = (GetBitsFixed(
pbThisBuffer, dwThisBitStart, 12 - bKeyBits ) << bKeyBits);

                  // Store the INT version as the absolute value
                  //    of the SHORT version.
                  pfbFrameBuffer->psvRots[I].iX = (pfbFrameBuffer-
>psvRots[I].sX < 0) ? pfbFrameBuffer->psvRots[I].sX + 0x1000 :
pfbFrameBuffer->psvRots[I].sX;
                  pfbFrameBuffer->psvRots[I].iY = (pfbFrameBuffer-
>psvRots[I].sY < 0) ? pfbFrameBuffer->psvRots[I].sY + 0x1000 :
pfbFrameBuffer->psvRots[I].sY;
                  pfbFrameBuffer->psvRots[I].iZ = (pfbFrameBuffer-
>psvRots[I].sZ < 0) ? pfbFrameBuffer->psvRots[I].sZ + 0x1000 :
pfbFrameBuffer->psvRots[I].sZ;
            }
      }
      else {                              // All other frames.
            SHORT sX, sY, sZ;             // Get the positional
                                          //    offsets.
            sX = GetDynamicFrameOffsetBits( pbThisBuffer,
dwThisBitStart );
            sY = GetDynamicFrameOffsetBits( pbThisBuffer,
dwThisBitStart );
            sZ = GetDynamicFrameOffsetBits( pbThisBuffer,
dwThisBitStart );

            // When we come to this area of the function,
            //    pfbFrameBuffer will have the previous frame
            //    still stored in it.  Just add the offsets.
            pfbFrameBuffer->svPosOffset.sX += sX;
            pfbFrameBuffer->svPosOffset.sY += sY;
            pfbFrameBuffer->svPosOffset.sZ += sZ;

            pfbFrameBuffer->svPosOffset.fX = (FLOAT)pfbFrameBuffer-
>svPosOffset.sX;
            pfbFrameBuffer->svPosOffset.fY = (FLOAT)pfbFrameBuffer-
>svPosOffset.sY;
            pfbFrameBuffer->svPosOffset.fZ = (FLOAT)pfbFrameBuffer-
>svPosOffset.sZ;
            for ( int I = 0; I < iThisBones; I++ ) {
                  // The same applies here.  Add the offsets
                  //    and convert to INT form, adding 0x1000
                  //    if it is less than 0.
                  // When Final Fantasy® VII loads these animations,
                  //    it is possible for the value to sneak up above
                  //    the 4095 boundary through a series of positive
```

```
                  //    offsets.
                  sX = GetEncryptedRotationBits( pbThisBuffer,
dwThisBitStart, bKeyBits );
                  sY = GetEncryptedRotationBits( pbThisBuffer,
dwThisBitStart, bKeyBits );
                  sZ = GetEncryptedRotationBits( pbThisBuffer,
dwThisBitStart, bKeyBits );

                  pfbFrameBuffer->psvRots[I].sX += sX;
                  pfbFrameBuffer->psvRots[I].sY += sY;
                  pfbFrameBuffer->psvRots[I].sZ += sZ;

                  pfbFrameBuffer->psvRots[I].iX = (pfbFrameBuffer-
>psvRots[I].sX < 0) ? pfbFrameBuffer->psvRots[I].sX + 0x1000 :
pfbFrameBuffer->psvRots[I].sX;
                  pfbFrameBuffer->psvRots[I].iY = (pfbFrameBuffer-
>psvRots[I].sY < 0) ? pfbFrameBuffer->psvRots[I].sY + 0x1000 :
pfbFrameBuffer->psvRots[I].sY;
                  pfbFrameBuffer->psvRots[I].iZ = (pfbFrameBuffer-
>psvRots[I].sZ < 0) ? pfbFrameBuffer->psvRots[I].sZ + 0x1000 :
pfbFrameBuffer->psvRots[I].sZ;
            }
      }

      // If we did not read as many bits as there are in the frame,
      //    return the location where the bits should start for the
      //    next frame.
      if ( (SHORT)(dwThisBitStart / 8) < sSize ) {
            return dwThisBitStart;
      }
      // Otherwise, return 0.
      return 0;
}
```

2.

This is an example loop that could be used to load a full animation.

```
      FF7FrameHeader fhHeader;
      ReadFile( hFile, &fhHeader, sizeof( fhHeader ), &ulBytesRead,
            NULL );
      BYTE * baData = new BYTE[fhHeader.dwChunkSize];
      ReadFile( hFile, &baData, fhHeader.dwChunkSize, &ulBytesRead,
            NULL );

      // This will be our buffer to hold one frame.
      //    We will only buffer one frame at a time, so to
      //    to fully load the animations, you would need to
      //    write your own routine to store the data in
      //    fbFrameBuffer after each loaded frame.
      FF7FrameBuffer fbFrameBuffer;
      fbFrameBuffer.SetBones( fhHeader.dwBones );
      INT iBits = 0;
      for ( DWORD J = 0; J < fhHeader.dwFrames; J++ ) {
            // We pass a pointer to fbFrameBuffer.  The first frame
            //    will load diretly into it.
            // Every frame after that will actually use it with the
```

```
            //    offsets loaded to determine the final result of that
            //    frame.
            iBits = LoadFrames( &fbFrameBuffer, fhHeader.dwBones,
                  iBits, baData );
            // Reverse the Y offset (required).
            fbFrameBuffer.svPosOffset.fY = 0.0f -
                  fbFrameBuffer.svPosOffset.fY;

            // The first rotation set is skipped.  It is not part of
            //    the skeleton.  Skipping is optional, but
            //    Final Fantasy® VII skips it; it is always 0, 0, 0.
            // I believe the actual use for the "root" rotation is
            //    to dynamically make the model point at its target
            //    or face different directions during battle.
            for ( DWORD I = 0; I < fhHeader.dwBones - 1; I++ ) {
                  fbFrameBuffer.psvRots[I+1].fX =
(FLOAT)fbFrameBuffer.psvRots[I+1].iX / 4096.0f * 360.0f;
                  fbFrameBuffer.psvRots[I+1].fY =
(FLOAT)fbFrameBuffer.psvRots[I+1].iY / 4096.0f * 360.0f;
                  fbFrameBuffer.psvRots[I+1].fZ =
(FLOAT)fbFrameBuffer.psvRots[I+1].iZ / 4096.0f * 360.0f;
            }
            // Store the data for this frame here (in your own
            //    routine).

      }

      delete [] baData;
```

Part IV: Qhimm's Input

Qhimm has taken the time to rewrite two of these functions used in decoding, so it is easier to understand for people who know C++ better than they know assembly (despite my comments being in the assembly code).
He has also written a more in-depth look at the logistics behind the rotation compression format and explains its limitations

"`GetValueFromStream`" is the C/C++ version of my "`GetDynamicFrameOffsetBits`" and his "`GetCompressedDeltaFromStream`" is the C++ version of my "`GetEncryptedRotationBits`".

```
short GetValueFromStream( BYTE *pStreamBytes,
      DWORD *pdwStreamBitOffset )
{
      // The return value;
    short sValue;
      // The number of whole bytes already consumed in the stream.
      DWORD dwStreamByteOffset = *pdwStreamBitOffset / 8;
      // The number of bits already consumed in the current stream byte.
      DWORD dwCurrentBitsEaten = *pdwStreamBitOffset % 8;
      // The distance from dwNextStreamBytes' LSB to the 'type' bit.
      DWORD dwTypeBitShift = 7 - dwCurrentBitsEaten;
```

```
        // A copy of the next two bytes in the stream (from big-endian).
        DWORD dwNextStreamBytes = pStreamBytes[dwStreamByteOffset] << 8 |
pStreamBytes[dwStreamByteOffset + 1];

        // Test the first bit (the 'type' bit) to determine the size of the value.
        if (dwNextStreamBytes & (1 << (dwTypeBitShift + 8)))
        {       // Sixteen-bit value:
                // Collect one more byte from the stream.
                dwNextStreamBytes = dwNextStreamBytes << 8 |
                        pStreamBytes[dwStreamByteOffset + 2];
                // Shift the delta value into place.
                sValue = (dwNextStreamBytes << (dwCurrentBitsEaten + 1)) >> 8;
                // Update the stream offset.
                *pdwStreamBitOffset += 17;
        }
        else
        {       // Seven-bit value
                // Shift the delta value into place (taking care to preserve the sign).
                sValue = ((short)(dwNextStreamBytes << (dwCurrentBitsEaten + 1))) >> 9;
                // Update the stream offset.
                *pdwStreamBitOffset += 8;
        }

        // Return the value.
        return sValue;
}


short GetCompressedDeltaFromStream( BYTE *pStreamBytes, DWORD *pdwStreamBitOffset,
        int nLoweredPrecisionBits )
{
        unsigned int nBits;
        int iFirstBit = GetBitsFromStream( pStreamBytes, pdwStreamBitOffset, 1 );
        if (iFirstBit)
        {
                unsigned int uType = GetBitsFromStream( pStreamBytes,
                        pdwStreamBitOffset, 3 ) & 7;
                switch (uType)
                {
                case 0:
                        // Return the smallest possible decrement delta (at given
                        //      precision).
                        return (-1 << nLoweredPrecisionBits);

                case 1: case 2: case 3: case 4: case 5: case 6:
                        // Read a corresponding number of bits from the stream.
                        int iTemp = GetBitsFromStream( pStreamBytes,
                                pdwStreamBitOffset, nBits );
                        // Transform the value into the full seven-bit value, using the bit
                        //      length
                        // as part of the encoding scheme (see notes).
                        if (iTemp < 0) iTemp -= 1 << (nBits - 1);
                        else                    iTemp += 1 << (nBits - 1);
                        // Adapt to the requested precision and return.
                        return (iTemp << nLoweredPrecisionBits);
                case 7:
                        // Read an uncompressed value from the stream (at requested
                        //      precision), and return.
                        iTemp = GetBitsFromStream( pStreamBytes,
                                pdwStreamBitOffset, 12 - nLoweredPrecisionBits );
                        return (iTemp << nLoweredPrecisionBits);
                default:
                }
        }
        // Default/error: return zero.
        return 0;
}


/*
        Notes for rotational delta compression scheme
```

==========

The delta values are stored in compressed form in a bit stream. Consecutive values share no bit correlation or encoding dependencies, rather they are encoded separately using a scheme designed to optimize small-scale rotations.

Rotations are traditionally given in normalized PSX 4.12 fixed-point compatible values, where a full rotation is the integer value 4096. Values above 4095 simply map down into the [0,4095] range, as expected from rotational arithmetics. When encoded, only the required 12 bits of precision are ever considered.

In some animations, the author can choose to forcibly lower the precision of rotational delta values below 12 bits. Though these animations are naturally not as precise, they encode far more efficiently, both because of the smaller size of 'raw' values, and also because of the increased relative span of the 'close-range' encodings available for small deltas. The smallest 128 [-64,63] sizes of deltas can be stored in compressed form instead of as raw values. This method is efficient
since a majority of the rotational deltas involved in skeletal character animation will be small, and thus doubly effective if used with reduced precision. Precision can be reduced by either 2 or 4 bits (down to 10 or 8 bits).

The encoding scheme is capable of encoding any 12-bit value as follows:

First, a single bit tells us if the delta is non-zero. If this bit is zero, there is no delta value (0) and the decoding is done. Otherwise, a 3 bit integer follows, detailing how the delta value is encoded. This has 8 different meanings, as follows:

Type	Meaning
0	The delta is the smallest possible decrement (under the current precision)
1-6	The delta is encoded using this many bytes
7	The delta is stored in raw form (in the current precision)

The encoding of small deltas works as follows: The encoded delta can be stored using 1-6 bits, giving us a total of 2+4+8+16+32+64 = 126 possible different values, which during this explanation will be explained as simple integers (the lowest bits of the delta, in current precision). The values 0 (no change) and -1 (minimal decrement) are already covered, leaving the other 126 values to neatly fill out the entire 7 bit range. We do this by encoding each value like follows:

- The magnitude of the delta is defined as the value of its most significant value bit (in two's complement, so the highest bit not equal to the sign bit). For example, the values '1' and '-2' have magnitude 1, while the value '30' will have a magnitude of 32. For simplicity, we also define the 'signed magnitude' as the magnitude multiplied by the sign of the value (so '-2' has a signed magnitude of -1).
- When encoding a value, we subtract its signed magnitude; essentially pushing everything down one notch towards zero, setting the most significant value bit to equal the sign bit and thus ensuring that none of the transformed values require more than six bits to accurately represent in two's complement.
- The transformed value is then stored starting from its magnitude bit (normally, you would have to start one bit higher to include the sign bit and prevent signed integer overflow). Small values will be stored using fewer bits, while larger values use more bits. The two smallest values, 0 and -1, are not encodeable but are instead handled using the previously mentioned scheme.

When decoding, you only need to know the number of bits of the encoded value, use the value of its most significant bit (not the most significant value bit!) as the magnitude, multiply it by the sign of the encoded value to get the signed magnitude, and then add that to the encoded value to get the actual delta value.

Some examples of encodings:

Delta value *	Encoded
0	0
-1	1 000
-5	1 011 111

*) in current precision, as integer

```
        15                  1 100 0111
        128                 1 111 xxxx10000000  (length depends on precision)

        (Note: The reduced precision is treated as rounding towards negative infinity)

*/
```