

# *The TrueMotion SDK*

Copyright 1994-98, The Duck Corporation

Version 7.0 -04 November 1998

TrueMotion and Comprending are Patent Pending technologies of the Duck Corporation,  
all rights reserved

This document is provided according to the Duck Corporation's TrueMotion developer's license agreement and  
the laws of the State of  
New York.

Unauthorized reproduction, distribution, or disclosure of this document in any form, in part or in whole, to any  
third party are expressly  
prohibited by law and by the terms of the above mentioned license.

This document is intended for distribution and use only by TrueMotion licensed developers in accordance with  
the aforementioned  
license.

The information in this document is to be considered confidential and proprietary.  
The information contained herein is subject to change without notice.

TrueMotion and Comprending names are trade names of the Duck Corporation.  
The TrueMotion logo is a trademark of The Duck Corporation.

- 
- Published 04-Nov-1998 10:00 PM
- 
- © 1994-98 The Duck Corporation
- 
- The Duck Corporation
- 375 Greenwich St.
- New York, NY 10013 USA
- 
- For Technical Support regarding this document
- and the information contained herein, please contact Duck via
- 
- phone. 518.786.1696
- or fax 518.786.8140

# Table of Contents

<b>1. An Introduction to the TM2X Encoding</b>	<b>TMS-13</b>
<b>Introduction</b>	<b>TMS-13</b>
<b>Components in the TM2X Compression Tool Kit (CTK) for Microsoft VFW/AVI</b>	<b>TMS-14</b>
<b>Features</b>	<b>TMS-15</b>
TrueMotion 2X (TM2X) Video Encoding/Decoding Technology	TMS-15
TrueMotion DK3 and DK4 Sound Compression Technology	TMS-15
<b>Compression Speed</b>	<b>TMS-15</b>
<b>Data Rate</b>	<b>TMS-16</b>
Recommendations	TMS-16
<b>TM2X Encoder User Interface</b>	<b>TMS-17</b>
Profile	TMS-17
Noise Reduction (Blur)	TMS-17
Force All Keyframes	TMS-17
Motion Search	TMS-17
Scale Key Frame Data Rate	TMS-18
<b>Tricks and Tips (Getting the most out of your bits!)</b>	<b>TMS-18</b>

<b>2. The TrueMotion SDK .....</b>	<b>TMS-19</b>
<b>This Document .....</b>	<b>TMS-19</b>
<b>An Introduction to TrueMotion SDK application development .....</b>	<b>TMS-20</b>
TrueMotion Video Compression .....	TMS-20
IntRAframe (key-frame only) image compression .....	TMS-20
IntERframe (delta-frame) video compression .....	TMS-20
TrueMotion Applications .....	TMS-21
Linear and Branching Video .....	TMS-21
IntraFile Branching .....	TMS-21
InterFile Branching .....	TMS-21
<b>TrueMotion Files .....</b>	<b>TMS-22</b>
Files containing multiple streams of video .....	TMS-22
Files containing multiple streams of audio .....	TMS-22
<b>TrueMotion playback services .....</b>	<b>TMS-23</b>
TrueMotion Application Process/Flow .....	TMS-23
Hardware Specifics .....	TMS-24
<b>Library Overview .....</b>	<b>TMS-24</b>
Library Initialization .....	TMS-24
System Abstraction Layer (SAL) .....	TMS-25
Dreamcast .....	TMS-25
<b>The Decompression Libraries - (DXA/DXV) .....</b>	<b>TMS-26</b>
<b>Video Processing (Using Decompressors and Screen Buffers) .....</b>	<b>TMS-26</b>
XImages .....	TMS-27
VScreens .....	TMS-27

<b>3. Function and Data Reference</b>	<b>TMS-29</b>
<b>DXV_ Function and Data Reference</b>	<b>TMS-29</b>
Data Definitions	TMS-29
dxvBitDepth	DXV color depth specification. TMS-29
dxvBlitQuality	Vscreen Blit Mode Specifier. TMS-30
dxvError	DXV Error return Codes. TMS-30
dxvImageType	xImage frame type specification. TMS-31
dxvOffsetMode	Positioning mode specifier. TMS-31
<b>Functions</b>	<b>TMS-32</b>
DXL_AlterVScreen	Alter a virtual screen's object attributes. TMS-32
DXL_AlterVScreenView	Alters virtual screen viewport location and dimensions. TMS-33
DXL_AlterXImage	Alter xImage's base attributes. TMS-34
DXL_AlterXImageData	Place new data in xImage container object. TMS-35
DXL_CreateVScreen	Create a virtual screen destination object. TMS-36
DXL_CreateXImage	Create a compressed image source container object. TMS-37
DXL_DestroyVScreen	Release a destination vScreen object for reuse. TMS-38
DXL_DestroyXImage	Release compressed image container. TMS-38
DXL_dxImageToVScreen	Decompress an xImage to a vScreen. TMS-39
DXL_ExitVideo	Exit and de-initialize video decompression library. TMS-40
DXL_GetVScreenBlitQuality	Get the active blit quality for a specified vScreen. TMS-40
DXL_GetXImageColorDepth	Get color depth of an xImage. TMS-41
DXL_GetXImageFrameBuffer	Get a pointer to a xImage's internal frame buffer. TMS-42
DXL_GetXImageType	Get type of image contained in xImage. TMS-43
DXL_GetXImageXYWH	Get xImage offset and dimensions. TMS-43
DXL_InitVideo	Initializes video decompression services. TMS-44
DXL_IsXImageKeyFrame	Checks the keyframe status of an xImage. TMS-44
DXL_MoveXImage	Change xImage destination offset. TMS-45
DXL_SetVScreenBlitQuality	Set the active blit quality for a specified vScreen. TMS-45
DXL_GetVScreenView	Get viewport of vScreen. TMS-46
Extensions for DOS - VESA	TMS-47
<b>Audio Processing (Using Decompressors and Audio Buffers)</b>	<b>TMS-48</b>
XAudioSrc	TMS-48
AudioDst	TMS-48
Loading Audio Samples	TMS-49
Maintaining the Audio Hardware Buffer	TMS-50
<b>DXA_ Function and Structure Reference</b>	<b>TMS-50</b>
DXL_AlterAudioDst	TMS-50
DXL_AlterXAudioData	TMS-51
DXL_CreateAudioDst	Create audio destination object. TMS-52
DXL_CreateXAudioSrc	Create compressed audio object. TMS-53
DXL_DestroyAudioDst	Destroy an audio destination object. TMS-53
DXL_DestroyXAudioSrc	Destroy an audio source object. TMS-54
DXL_dxAudio	Decompress audio samples. TMS-54
DXL_ExitAudio	De-initialize audio decompression services. TMS-55
DXL_InitAudio	Initialize audio decompression service. TMS-55

<b>File management using HFB .....</b>	<b>TMS-56</b>
Opening a TrueMotion file and its Streams .....	TMS-56
HFB_Init – Starting the streaming process .....	TMS-57
Managing Streamed and/or unbuffered Data .....	TMS-57
Avoiding stalls during streaming .....	TMS-57
Sample Parse loop .....	TMS-59
 <b>HFB_ Function and Structure Reference .....</b>	 <b>TMS-60</b>
HFB_CloseFile.....Close a TrueMotion file. ....	TMS-60
HFB_CreateBuffer.....Create a High-speed File Buffer. ....	TMS-61
HFB_DestroyBuffer.....Destroy buffer for re-use by library/system. ....	TMS-62
HFB_Exit.....De-initialize/free HFB library and resources. ....	TMS-62
HFB_Init.....Initialize HFB library resources. ....	TMS-63
HFB_FillBuffer.....Fill/recycle empty space in buffer from disk. ....	TMS-64
HFB_GetAudioInfo.....Get general information describing an audio stream. ....	TMS-65
HFB_getBufferStatus.....Get status of a buffer. ....	TMS-65
HFB_getDataPosition.....Get starting position of data block (in samples). ....	TMS-66
HFB_GetDataSize .....	Get size of data block (in bytes). ....TMS-67
HFB_GetFileInfo .....	Get a pointer to file information. ....TMS-67
HFB_GetFrameRates .....	Get frame rate related information based on a combination of an audio and video stream. ....TMS-68
HFB_GetIndexFlags.....Get index flags for data block. ....	TMS-69
HFB_GetSamplesPerFrame.....Get number of audio samples per video frame. ....	TMS-70
HFB_GetStream.....Get a handle to a stream. ....	TMS-71
HFB_GetStreamInfo .....	Get pointer to stream info. ....TMS-72
HFB_GetStreamingData .....	Get buffered data block from stream. ....TMS-73
HFB_InitBuffer.....Initialize buffer and pre-load data from disk file. ....	TMS-74
HFB_LoadIndex.....Load a TrueMotion File's Index. ....	TMS-75
HFB_FindFile.....Initiate find and open of a TrueMotion File. ....	TMS-76
HFB_OpenFile.....Open a TrueMotion File. ....	TMS-77
HFB_ParseFile.....Parse a TrueMotion File's header and initiate index loading. ....	TMS-77
HFB_ReadData.....Non-buffered stream read. ....	TMS-78
HFB_ReleaseStream.....Release stream object for reuse. ....	TMS-79
HFB_ReleaseStreamingData .....	Release data block (chunk) for recycling. ....TMS-79
HFB_SetBufferMode.....Set buffer repeat (loop) mode. ....	TMS-80

# *1. An Introduction to the TM2X Encoding*

---

## **1.1 Introduction**

TrueMotion 2X combines playback efficiency with superior data rate control. TrueMotion 2X also greatly speeds and simplifies the compression process. TM2X is up to twice as fast to compress and decompress as TrueMotion 2.0!

TrueMotion 2X encoding is performed using a Microsoft Video for Windows compatible codec with applications such as Adobe Premiere, AVI Edit and others. A TM2X codec plug-in is also available for QuickTime 3.0 and compatible applications.

TrueMotion 2X was developed as an enhancement to TM2.0 to provide superior playback efficiency at lower data rates. TM2X puts high definition resolutions and quality into the range of the sub-\$1000 PC as well as high performance consoles and set-top boxes.

TM2X's native support for the YUV422 format allows it to exceed the quality of MPEG2 and DV (YUV 420) as seen in consumer and PC applications.

On 200Mhz systems, TM2X typically exceeds 640x480 at 30fps.

## **1.2 Components in the TM2X Compression Tool Kit (CTK) for Microsoft VFW/AVI**

- **TM2X VFW Codec**

This is the main component of the TM2X compression toolkit. Once installed, it is used with applications like Adobe Premiere to prepare AVI files w/ TM2X compressed video. Unless a valid TM2X key string is specified, the codec will function in "Trial"-mode, creating videos with a TrueMotion Logo Watermark in one of the corners.

- **TM2X DirectShow Filter**

This is a re-distributable DirectShow decompression filter for TrueMotion. You'll find that in addition to TM2X, it will decompress most other TrueMotion formats (TM1, TM2.0 and TMRT)

- **DK3 and DK4 ACM Codecs**

Duck's audio DK3 and DK4 compression technology is included with the Compression Tool Kit.



## 1.3 Features

### 1.3.1 TrueMotion 2X (TM2X) Video Encoding/Decoding Technology

- full screen, 24-bit color (full YUV422 color resolution)
- 15,24,30,60 fps
- 640x480 full screen playback (or higher)
- supports high resolution textures
- 50-60% CPU utilization for 640x480x30fps decompression\*
- typically 24:1 compression ratio
- 1-8Mbps, depending on frame rate, resolution and content

### 1.3.2 TrueMotion DK3 and DK4 Sound Compression Technology

- 16:3 and 16:4 compression ratios
- DK3 compresses Stereo sound only
- DK4 compresses Mono and Stereo formats
- 11.025, 22.050 and 44.100 KHz sample rates supported
- less than 1 percent CPU utilization per stereo pair\*

\*CPU utilization indicates the estimated impact of compression only on a 200MHz processor (e.g. Pentium w/MMX, PowerPC, Hitachi SH-4), disk speed, video card performance etc. may cause actual performance to vary.

## 1.4 Compression Speed

The largest contributing factor to compression speed is the motion search radius. If only key frames only are used, the compression speed is typically in the range of frames per second, rather than in seconds per frame. If you are using TM2X for archival purposes, try disabling data rate limitations and making every frame a keyframe, this will result in the fastest compression, but also, the lowest (worst) compression ratio.

## 1.5 Data Rate

TrueMotion 1.0 and 2.0 users should be especially pleased by TM2X's data rate control. You'll find that the TM2X compressor reliably produces the specified data rate in as little as half the time!

### 1.5.1 Recommendations

DataRate requirements are actually related more to the information content of an image, rather than strictly by the resolution. For example, high resolution clips with subtle motion may have a lower data rate than lower resolution clips with saturated color and/or fast motion.

The following table is only a recommendation for data rate. The material being compressed will determine actual data requirements.

Frame Size	Frame Rate	Data Rate Range	CD Speed	Typical Data Rate
<b>320x240</b>	15fps	150-300KB/Sec	(1-2x)	200K/Sec
<b>320x240</b>	30fps	300-600KB/Sec	(2-4x)	450K/Sec
<b>640x480</b>	15fps	450-900KB/Sec	(3-6x)	600K/Sec
<b>640x480</b>	30fps	600-1200KB/Sec	(4-8x)	900K/Sec

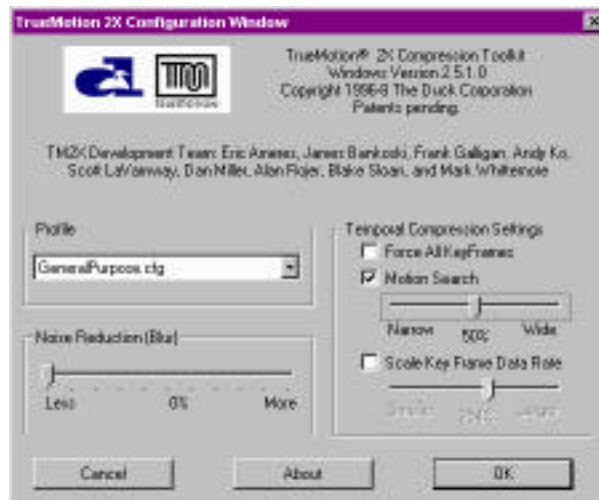
greater than 640x480 @30fps 1200-1800KB/Sec (8-12x)

## 1.6 TM2X Encoder User Interface

The TM2X configuration window shown is used in the Video for Windows plug-in codecs.

### Profile

The 2X profiles provide the overall personality of the encoder. Profiles are provided to adjust the encoder to such conditions as film grain, video noise, rendering style, etc. We recommend you try more than 1 profile with your materials since it is hard to predict the correct profile for all materials.



### Noise Reduction (Blur)

The 2X codec includes a Gaussian blur noise reduction kernel. The lowest setting disables the kernel entirely. Adding blur will slow down the compression process slightly.

### Force All Keyframes

This setting forces all frames to be keyframes, regardless of the compression applications settings. This allows the decompressor to work without the need for a frame buffer. This increases datarate (or reduces quality), but also increases performance and reduces memory requirements. Files compressed with this option enabled should generally not be merged with files compressed without this option.

### Motion Search

Using (non-zero) motion estimation can improve encoding quality substantially, but at a playback performance cost of 10-20%. Real motion estimation also requires the decompressor to use 2 frame buffers. Motion compensation also has the unfortunate side effect of drastically increasing the time required to compress each frame. Using a wider motion radius will require more compression time.

The maximum motion radius corresponds to 1/40 of the image's resolution and 1/30 of the vertical. For example, for 640x480 resolution, 100% is +/- 32 pixels horizontally and vertically.

---

**Note:** Disabling motion estimation still allows the compressor to take advantage of interframe redundancy, but only where the motion vector is zero (ie the pixels are still).

---

### Scale Key Frame Data Rate

The 2X compressor can automatically adjust the size of key-frames as requested by the compression application. For example, Adobe Premiere 5.0 requests that keyframes be twice as large as the average data rate. Setting this parameter to 50% will scale the key frame requests back down to the average data rate of the file.

When motion estimation is used, or the material being compressed is fairly static, this value should be 100% or greater. When motion estimation is disabled, but interframing is still in use (i.e. not all keyframes) this setting should be between 50 and 100% (when using Adobe Premiere).

## 1.7 Tricks and Tips (Getting the most out of your bits!)

On systems that provide hardware support for YUV scaling, try compressing your materials at 320x480 (scaled to \* in the horizontal dimension). Use the hardware scaling to scale to the 640x480 resolution. Vertical scaling is typically far more noticeable (usually as excessive blockiness). Horizontal scaling is almost always far more subtle. This technique can drastically reduce data rate requirements without impacting quality as noticeably as dropping to 320x240 and scaling in both dimensions on playback.

---

**Note:** On many platforms, TM2X playback provides fast horizontal scaling for platforms w/o hardware support.

---

## *2. The TrueMotion SDK*

---

### **2.1 This Document**

It is the intent of this document to provide a general understanding of the TrueMotion Development APIs on all supported platforms.

Platform specific issues will be addressed in the appendices.

It is our intent at Duck to minimize the differences between developing using TrueMotion on different platforms while still providing a framework that allows each system to shine in it's strongest areas (i.e. file i/o, supported video modes, color depths, resolutions, etc...).

It is also our intention to provide a single, consistent interface for development, regardless of the type(s) of TrueMotion compression employed by your application (e.g. TrueMotion 1.0 vs. TrueMotion 2.0 vs. TrueMotion 2X and DK3 vs. DK4 vs. uncompressed audio).

Note that all TrueMotion algorithms may not support all playback formats. Applications should check return codes and act accordingly.

Your comments regarding this document are always welcome. Please address them to [support@duck.com](mailto:support@duck.com) or contact Product Support at the Duck Corporation.

## 2.2 An Introduction to TrueMotion SDK application development

The TrueMotion application development tools form a suite of software tools for the compression, decompression and real-time management of audio and video information. It is from these libraries and tools that the TrueMotion codecs for other video service provider packages have been developed.

Programming using the TrueMotion SDK provides developers with:

- Operating system independence (operating system/ vendor specific video service providers are not required). TrueMotion is available for DOS, Win32, Macintosh, Sega Dreamcast, and many Unix platforms.
- Greater control and flexibility over video playback, the application can completely control memory management, file handling, user input, etc.
- easy customization
- complete integration with developer's application. The TrueMotion SDK provides both static and dynamic versions of most libraries.
- higher performance and efficiency
- The same level of facilities used in developing our own filters and codecs for other video service provider packages.

TrueMotion compression supports a variety of video formats as well as 16 bit mono and stereo audio.

The hallmark application of TrueMotion has been as a means for providing High-Quality Full Motion Video (FMV) playback without dedicated, special purpose hardware.

We will begin by discussing the underlying technologies of TrueMotion and looking at some applications.

### 2.2.1 TrueMotion Video Compression

#### **IntraFrame (key-frame only) image compression**

TrueMotion is at its core, an IntraFrame (still image) compression algorithm. Simply put, its strength lies in its ability to compress independent frames. When applied to video streams, this type of compression is often referred to as "Key frame only" compression.

IntraFrame compression is most useful in applications that require either still-frame or non-linear (random) access to images. Since each frame is stored in its entirety it can be processed independently.

Another important consideration is that since delta-frames are not used, very little working memory is needed to decompress an IntraFrame image. This can be important in applications that are severely limited on memory usage.

During playback of an IntraFrame compressed video stream, it is possible to skip decompression of any particular frame (or sequence of frames) without effecting subsequent images.

#### **InterFrame (delta-frame) video compression**

To provide higher compression ratios, TrueMotion can also employ interFrame video compression techniques. Interframe compression uses delta frames to store only the differences in a progression of frames. Key-frames are also used to provide points for seeking, edits and cuts.

During playback of interframe video streams it is important to not skip decompressing frames (at least not entirely) since subsequent frames require that the interframe buffer (i.e. XImage workRAM) be in the proper state.

## 2.2.2 TrueMotion Applications

### Linear and Branching Video

By intelligently managing file buffering and video playback, branching video becomes an effective tool in the development of interactive entertainment. The TrueMotion development libraries provide a framework and capability for multi-tiered buffering, multi-stream video, fast transition between streams and/or files and other real-time, interactive “editing” techniques.

Branching can be accomplished in one of two ways. Intrafile branching and Interfile branching. Interfile branching is branching from one file to another and requires the use of multiple buffers, files and streams. Intrafile branching is branching within a single file to a new position (or stream) within the same (currently playing) file.

### IntraFile Branching

IntraFile branching allows branching without additional buffering requirements beyond those of normal playback. Since the file’s index is already loaded and more resources can be devoted to a single buffer, branching in short jumps or between streams within a file can be completely seamless.

Short branching is performed by skipping over (i.e. getting and releasing without decompressing) video and audio records in the buffer until the destination is reached. This is only efficient where the jump is less than or equal to the size of the allocated i/o buffer (plus the size of any hardware buffering).

Longer branching is accomplished by stopping playback from a buffer, re-initializing the buffer at the new frame location and reading from the new location. A delay is introduced while waiting for the CD head to seek to the new location in the file and while waiting for the initial data at the branch point to be loaded. To minimize this latency, use DuckTool.exe to remove audio skew from the file and pay close attention to the data rate of the file during compression.

Branching between streams - Files can contain multiple video and audio streams that can be switched between on the fly. Having alternative streams can provide for instant branching between multiple scenarios. The difficulty with this method results from the limited bandwidth of the CD-ROM.

### InterFile Branching

InterFile Branching adds flexibility by allowing the application to store each segment in its own file and choose between them as necessary.

To branch between files, a single buffer and file can be used but would introduce latency since the branch operation would have to wait for one segment to end before the next could be opened and prepared for playback. Typically branches of this sort would introduce a latency of <1sec.

By using two or more buffers and files, branching can occur more seamlessly by allowing the application to pre-seek, index and load new files while the old one continues to play. While playing through the “current” segment, `HFB_getBufferStatus` is used to determine when the “current” segment has been completely buffered. Once this occurs, the application is free to interleave the “new” segment’s file loading, indexing, etc. in with the playback of the “current” segment as it plays out. See “Loading a file in the Background” below for details.

## 2.3 TrueMotion Files

The TrueMotion libraries support the AVI (Audio Video Interleave, Microsoft Video for Windows) file standard.

Like all AVI files, TrueMotion files are composed of distinct streams of information that can be used independently or in conjunction with other streams. HFB limits control to up to 16 streams within a single file but has no restrictions on stream length or content.

Normally, streams are interleaved within a TrueMotion file to provide high performance transfer from disk. The compression tools used to produce the AVI file determine how the streams are interleaved. It is recommended that video and audio be interleaved on a single frame (1:1) basis.

---

**Note:** TrueMotion files require some memory and load time overhead for index management. Approximately 8 bytes are allocated per frame (block) of data. This memory is considered to be part of, and is taken directly from, the space allocated to an HFB buffer.

---

### 2.3.1 Files containing multiple streams of video

For example, files can be created with a single stream of audio and a separate stream of video for each of a number of view angles for a given scene. During playback, all streams are parsed (incurring very little overhead) but only the audio stream and a single stream of video is realized.

---

**Note:** Multiple streams (especially video) can considerably effect the data rate of a file. There is however, little additional CPU load incurred by parsing any unused compressed/un-realized streams.

---

### 2.3.2 Files containing multiple streams of audio

Multiple streams of audio can be used to provide either multi-channel sound (each stream can be mono or stereo) or alternative sound tracks, sound effects, commentary, etc...



## 2.4 TrueMotion playback services

### 2.4.1 TrueMotion Application Process/Flow

Before going into the details of TrueMotion, we will look at as simple a play loop as possible to introduce some of the underlying concepts in the TrueMotion developer's kit.

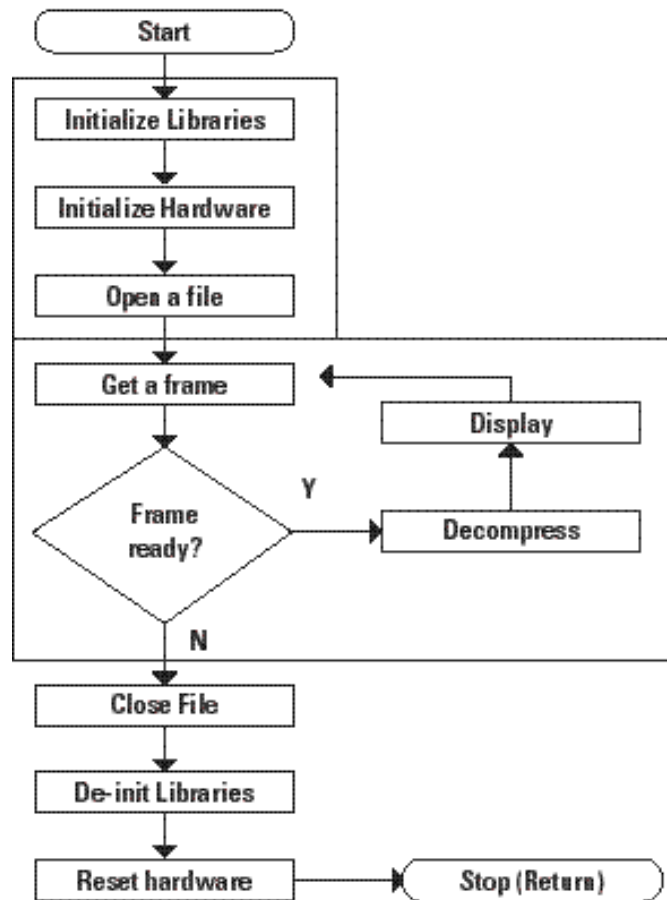


Figure 2.1 Initialization of Hardware and Software

Figure 1.1 represents perhaps the simplest play loop one might use for playing a video. The loop performs the minimum library and hardware initialization, steps through a file frame by frame, displaying them as it goes, and resets when it runs out of frames. The player does not address audio, synchronization/timing or user input among other things.

No matter how complex a player implementation becomes, it is important to see this underlying loop's functionality.

The program consists of three main parts

- 1) Initialization of hardware and software
- 2) Retrieval/Decompression/display of data
- 3) De-Initialization of hardware and software

## Hardware Specifics

Hardware initialization consists of those steps necessary to provide for the display of decompressed information (whether that be video or audio). The specific hardware initialization steps necessary are left up to the application developer. Please feel free to use the sample code as a starting point. Hardware initialization will not be discussed in detail in this document.

## 2.5 Library Overview

The TrueMotion API combines functions from a set of libraries. Each library provides an independent aspect of functionality. The libraries will each be discussed in their own section(s).

In order to provide the most flexibility, the API layer has been designed to be as hardware non-specific as possible. Complete source for a sample player implementation is provided.

The TrueMotion libraries have been developed using an “object oriented” model to simplify development and improve maintainability. Because C and assembly languages were used, certain liberties have been taken with the “object oriented” model.

---

**Note:** The TrueMotion libraries have been implemented in such a way as to limit the amount of dynamic memory allocation during operation. Much of the memory allocation is consolidated into larger blocks which are allocated at the initialization of each library. During normal operation, the libraries pre-allocate an application specified number of objects during initialization. However, by specifying dynamic allocation during initialization, it is possible to bypass this “pooling” of resources and allow the libraries to dynamically create and destroy objects as needed.

---

### Library Initialization

Before using any of the lower level TrueMotion library functions, the libraries should be initialized. It is at this stage that most memory management and allocations are performed. During initialization, the application specifies the number of audio/video/file/stream and buffer resources that will be required by the application.

Each library contains one or more initialization and exit functions that are suffixed “Init” and “Exit”.

Init functions take as arguments the number of objects to allocate within that library’s pool of resources. These numbers represent the maximum number of simultaneously held objects. Objects may be created, used and destroyed any number of times throughout the life of an application.

Exiting a library will invalidate any open objects. It is for this reason that all objects should be destroyed before exiting a library.

*Code Sample 1 Basic Initialization and Exit*

```
HFB_Init(1,2,1); /* initialize HFB library w/1 file, 2 streams, 1 buffer */

DXL_InitVideo(1,1); /* initialize video library for 1 xImage and 1 vScreen */

XL_InitAudio(1,1); /* initialize audio library for 1 source and 1 dest */

MoviePlay("movie.avi");

DXL_ExitAudio();

DXL_ExitVideo();

HFB_Exit();
```

## 2.5.1 System Abstraction Layer (SAL)

The System Abstraction Layer or SAL is actually an internal API that the TrueMotion libraries use to communicate with a platform's operating system. By avoiding directly accessing the operating system, the TrueMotion libraries afford developers the flexibility to employ their own file handling, memory management, etc...

Default SAL modules and/or libraries are provided along with the TrueMotion libraries and sample applications. These modules can be used directly or modified as needed. This means that if you do not need to customize system access to your needs, a default method is automatically and invisibly provided.

The SAL is comprised of relatively few functions which should be familiar to any programmer.

For file management, `duck_open`, `duck_close`, `duck_seek` and `duck_read` are called in place of direct accesses to `open`, `close`, `seek` and `read` respectively.

For memory management, `duck_malloc`, `duck_calloc`, and `duck_free` are used in place of direct calls to `malloc`, `calloc` and `free`. One important difference in the `duck_malloc` and `duck_calloc` functions is the addition of a parameter which the libraries use to specify the intended use of the memory allocated. This parameter allows for effective use in non-unified memory architectures. (i.e seldom used objects can be relegated to less precious resources). Calls to `duck_calloc` must clear the are of memory that they allocate, unpredictable behavior will otherwise occur.

For string manipulation, `duck_memcpy`, `duck_memset` and `duck_strcmp` are called in place of `memcpy`, `memset` and `strcmp`.

### **Dreamcast**

On game consoles, additional functionality is often needed to translate and/or actually implement the functionality required by the library. For instance, most game consoles use sector level access to CD ROM files which must be translated to byte level access for the library. In console game development, it is also important to be able to manipulate memory management performed by the libraries directly.

## 2.6 The Decompression Libraries - (DXA/DXV)

The Decompression Library (hereafter DXA and DXV or collectively DXL) provides decompression services for audio and video. DXL functions are prefixed with DXL\_.

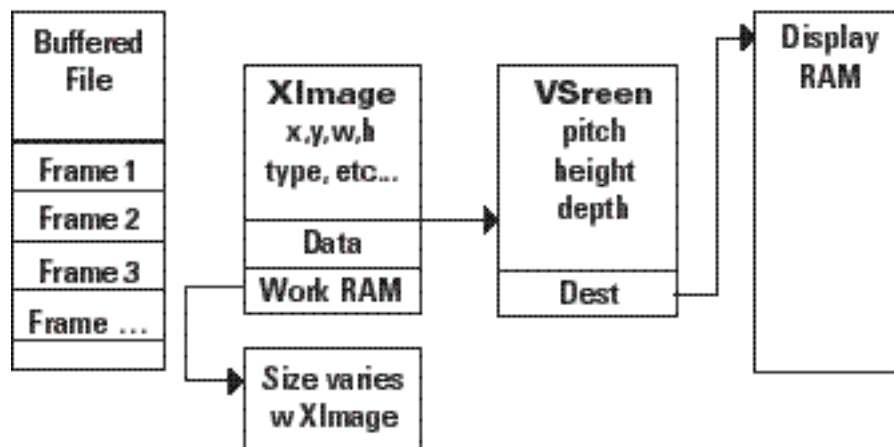
Throughout the decompression library, the concept of container objects is used as a way of describing containers for video and audio data in compressed and uncompressed states.

To manipulate (decompress) compressed video data, the data must first be associated with an `XImage` (Compressed Image) container object. Audio data must be associated similarly with an `XAudioSrc` (Compressed Audio Source). The format of compressed data is determined at the time of compression and cannot be changed by an application. Container objects can be queried to gather information about the data they contain.

Before compressed information can be decompressed, a virtual destination object must be created to describe the characteristics and location for the uncompressed data. Video data is decompressed into a `VScreen` (Virtual Screen) and audio data is decompressed into a `AudioDst` (Audio Destination). Unlike compressed source containers, the format of the virtual destination is determined entirely by the application at run time. The library will map and/or convert data as it is decompressed to match the format set by the application.

The term “virtua” is used to describe destinations since there is no strict physical connection between a destination and hardware. Virtual destinations can just as easily be locations in CPU work RAM video display memory.

## 2.7 Video Processing (Using Decompressors and Screen Buffers)



**Figure 2.2** Video and image decompression in its simplest form involves a single `XImage/VScreen` (decompressor/blitter) pair.

### 2.7.1 XImages

An XImage is a container object or decompressor, which holds a reference to a compressed image record and the resources necessary for decompressing it. An XImage is generally created according to a sample frame (generally the initial frame) in a stream, and may not work properly for data from other streams (whether they be audio or video).

The attributes of an XImage object are available via DXL\_Get functions; these attributes are fixed during compression and can not generally be altered by an application. Attribute information describing an image is available immediately after creating the XImage and is updated whenever the DXL\_AlterXImageData function is used to associate new data with the XImage.

Generally, the XImage dimensions are less than or equal to the pitch and height of the Vscreen.

Before beginning decompression, the program creates an XImage and associates the first frame/block of compressed video data with it. This makes the dimensions of the compressed image available to the application via the DXL\_GetXImageXYWH call:

*Code Sample 2 Creating an xImage*

```
bg.index = HFB_GetStreamingData(xVideoStream,
(void **)&data,&length, DUK_FORWARD, 1);

xImage = DXL_CreateXImage(data);

DXL_GetXImageXYWH(xImage, &xoff, &yoff, &width, &height);
```

If necessary, the application can use this information to set up display overlays, polygon structures, etc., using the hardware's display engine.

Note that Ximages do not contain copies of the compressed data, they merely contain a reference to it. It is important, therefore, to insure the persistence of that data during the time between the association of the data (DXL\_AlterXImageData) and the actual decompression (DXL\_dxImageToVScreen).

### 2.7.2 VScreens

A VScreen describes the destination location and format for blitting (rendering/realizing) an XImage. Unlike XImages, VScreens are controlled entirely by the application. During decompression, the TrueMotion libraries convert to the format specified by the application.

The primary characteristics of a VScreen are destination address, bit/color depth, pitch and height.



## *3. Function and Data Reference*

---

### **3.1 DXV\_ Function and Data Reference**

#### **3.1.1 Data Definitions**

##### **dxvBitDepth**

DXV color depth specification.

```
typedef enum BITDEPTH {  
  
    DXRGBNULL = 0,  
    DXRGB8 = 1,  
    DXRGB16_555 = 2,  
    DXRGB24 = 3,  
    DXRGB_UNUSED = 4,  
    DXRGB16VESA_555 = 5,  
    DXRGB8VESA = 6,  
    DXRGB16_565 = 7,  
    DXYUY2 = 8,  
    DXYVU9 = 9,  
    DXYV12 = 10,  
    DXUYVY = 11,  
    DXRGB32 = 12,  
    DXRGB16VESA_565 = 13,  
    DXMAX  
  
} dxvBitDepth ;
```

##### **Description**

Color depth of an XImage or VScreen

## dxvBlitQuality

Vscreen Blit Mode Specifier.

```
typedef enum BLITQUALITY {  
  
    DXBLIT_SAME = 0, /* Blit without scale or stretch */  
    DXBLIT_RESERVED1,  
    DXBLIT_RESERVED2,  
    DXBLIT_STRETCH, /* stretch, Double Pixels; leave skipped lines */  
    DXBLIT_RESERVED3,  
    DXBLIT_STRETCH_BRIGHT, /* stretch, double pixels, interpolate lines */  
    DXBLITMAX  
  
} dxvBlitQuality ;
```

### Description

Specifier for scaling to be performed by vScreen (blitter)

## dxvError

DXV Error return Codes.

```
typedef enum DXL_ERR{  
  
    DXL_LOW_ERR = -32000,  
    DXL_INVALID_REQUEST = -8,  
    DXL_VERSION_CONFLICT = -7,  
    DXL_INVALID_DATA = -7,  
    DXL_INVALID_BLIT = -6,  
    DXL_BAD_DATA = -5,  
    DXL_ALLOC_FAILED = -4,  
    DXL_NULL_FRAME = -3,  
    DXL_NULLSOURCE = -2,  
    DXL_NOTINUSE = -1,  
    DXL_OK = 0,  
    DXL_HOLD_FRAME = 1  
  
} dxvError ;
```

### Description

Provided for error handling purposes.



## dxvImageType

xImage frame type specification.

```
typedef enum IMAGETYPE {  
  
    DXL_INTRAFRAME = 0,  
    DXL_INTERFRAME  
  
} dxvImageType;
```

### Description

Type of information held by an xImage

## dxvOffsetMode

Positioning mode specifier.

```
typedef enum OFFSETXY {  
  
    DXL_ABSOLUTE = 0,  
    DXL_RELATIVE  
  
} dxvOffsetMode;
```

### Description

Type of information held by an xImage

## 3.2 Functions

### DXL\_AlterVScreen

Alter a virtual screen's object attributes.

```
#include <duck_dxl.h>
```

```
int DXL_AlterVScreen( DXL_VSCREEN_HANDLE dst, unsigned char *addr, enum BITDEPTH bd, int p, int h  
)
```

#### Arguments

**dst** - handle to a virtual screen destination.

**addr** - new destination address or NULL for no change.

**bd** - new destination bit depth or -1 for no change.

**p** - new destination pitch in bytes or 0 for no change.

**h** - new destination height in scanlines (pixels) or 0 for no change.

#### Return Value

**0** - success or negative error code.

#### See Also

DXL\_CreateVScreen

#### Description

Alter a virtual screen's object attributes.

## DXL\_AlterVScreenView

Alters virtual screen viewport location and dimensions.

```
#include <duck_dxl.h>
```

```
int DXL_AlterVScreenView( DXL_VSCREEN_HANDLE dst, int x, int y, int w, int h )
```

### Arguments

**dst** - handle to a virtual screen object.

**x,y,w,h** - new viewport rectangle (in pixels).

### Return Value

None.

### See Also

DXL\_ AlterVScreen

### Description

Alter virtual screen viewport location and dimensions

### Notes

width/height clipping is currently supported by TrueMotion 1.0 Only

## DXL\_AlterXImage

Alter xImage's base attributes.

```
#include <duck_dxl.h>
```

```
DXL_XIMAGE_HANDLE DXL_AlterXImage( DXL_XIMAGE_HANDLE xImage, unsigned char *data, enum  
IMAGETYPE type, enum BITDEPTH bitDepth, int width, int height )
```

### Arguments

**xImage** - handle to a compressed image source (xImage).

**data** - pointer to compressed video data

**type** - new xImage type (DXL\_INTRAFRAME, DXL\_INTERFRAME)

**bitDepth** - new bit depth (DXRGB16 or DXRGB24)

**width** - new xImage width

**height** - new xImage height

### Return Value

handle to xImage on success, NULL on error

### See Also

### Description

Alter xImage's attributes.

### Notes

The use of this function may affect the state of the xImage's frame buffer. During interframe compression, this can result in corruption of the decompressed image. Make sure to use this function only prior to decompressing a key frame.

## DXL\_AlterXImageData

Place new data in xImage container object.

```
#include <duck_dxl.h>
```

```
int DXL_AlterXImageData( DXL_XIMAGE_HANDLE src, unsigned char *data )
```

### Arguments

**src** - handle to a compressed image source (xImage).

**data** - pointer to compressed video data.

### Return Value

0 on success or negative error code.

-3 indicates that the data pointer was passed in as NULL, some compression applications (such as Adobe Premiere) use this to indicate that the new frame is identical to the previous frame and the previous frame is to be held.

### See Also

DXL\_CreateXImage

### Description

Provides a compressed source with new data to decompress. New xImage attributes can be queried any time after changing the address of the compressed data with this function.

## DXL\_CreateVScreen

Create a virtual screen destination object.

```
#include <duck_dxl.h>
```

```
DXL_VSCREEN_HANDLE DXL_CreateVScreen( unsigned char *addr, enum BITDEPTH bd, short p, short h )
```

### Arguments

**addr** - pointer to destination frame buffer, memory, h/w sprite, etc.

**bd** - bit depth of the virtual screen destination.

```
(DXRGB8, DXRGB16, DXRGB24  
(DXRGB24 = DXRGB24CHAR, DXRGB24BITMAP)
```

**p** - destination area pitch in bytes.

**h** - destination area height in scanlines (pixels).

### Return Value

A 32 bit handle to a virtual screen.

NULL - no vScreen is available

### See Also

DXL\_DestroyVScreen, DXL\_AlterVScreen, DXL\_AlterVScreenVESAMode

### Description

Creates a virtual screen destination for video decompression. A destination address, bitdepth (DXRGB8, DXRGB16, DXRGB24, etc...), pitch and height must be provided.

### Notes

DXRGB24 maps to DXRGB24CHAR (VDP2 character mode) which is currently the only supported 24bit mode for the Sega Saturn.

DXRGB24BITMAP should be supported in the near future.

Use DXRGB8VESA or DXRGB16VESA when using paged VESA modes (DOS only).

## DXL\_CreateXImage

Create a compressed image source container object.

```
#include <duck_dxl.h>
```

```
DXL_XIMAGE_HANDLE DXL_CreateXImage( unsigned char *data )
```

### Arguments

**data** - pointer to compressed image data.

### Return Value

a handle to a compressed image source (xImage).

NULL - no xImages available.

### See Also

HFB\_GetStreamingData, HFB\_ReadData, DXL\_DestroyXImage

### Description

Allocates buffers and initializes structures needed to decompress a source-compressed image (xImage).

### Notes

This is one of the few non-lib initialization functions that allocate memory.

The amount of memory allocated will vary according to the size and type of compressed source image (interFrame images require a complete frame buffer, intraFrame and sprites require only a small RAM buffer, usually less than 1K).

## DXL\_DestroyVScreen

Release a destination vScreen object for reuse.

```
#include <duck_dxl.h>
```

```
void DXL_DestroyVScreen( DXL_VSCREEN_HANDLE dst );
```

### Arguments

**dst** - handle to a virtual screen destination.

### Return Value

None.

### See Also

DXL\_CreateVScreen

### Description

Release a destination vScreen object for reuse.

## DXL\_DestroyXImage

Release compressed image container.

```
#include <duck_dxl.h>
```

```
void DXL_DestroyXImage( DXL_XIMAGE_HANDLE src )
```

### Arguments

**src** - handle to a compressed image source (xImage).

### Return Value

None.

### See Also

DXL\_CreateXImage

### Description

Releases a compressed image source from use. Frees allocated memory for re-use.



## DXL\_dxImageToVScreen

Decompress an xImage to a vScreen.

```
#include <duck_dx1.h>
```

```
int DXL_dxImageToVScreen( DXL_XIMAGE_HANDLE src, DXL_VSCREEN_HANDLE dst )
```

### Arguments

**src** - handle to a compressed image source (xImage).

**dst** - handle to a virtual screen (vScreen) (may be NULL to perform internal decompression only).

### Return Value

**0** - success

**1** - place-holder frame (hold previous frame, no frame decompressed)

**<0** error code.

### Description

Decompress the compressed image source to the destination virtual screen according to current source and destination objects' attributes.

### Notes

Primarily for use with interframe streams, DXL\_dxImageToVScreen with a NULL destination can be used if an application deems it necessary to “skip” an interframe frame in order to gain back some CPU time to maintain proper synchronization and/or performance.

This NULL destination feature can also be used in conjunction with DXL\_BlitXImageToVScreen to perform the decompression and transfer (“blit”) as two distinct passes (useful in multi-processor or multi-tasking architectures).

*Code Sample 4 Decompressing and/or Skipping Frames*

```
if (frameCount > lastFrame){

    if (frameCount > lastFrame + 3){
        /* behind by 3, use faster decompression to catch up */
        DXL_dxImageToVScreen(src, NULL);
    }else{
        DXL_dxImageToVScreen(src, dst); /* normal decompression */
    }
    lastFrame++;
}
```

## DXL\_ExitVideo

Exit and de-initialize video decompression library.

```
#include <duck_dxl.h>
```

```
void DXL_ExitVideo( void )
```

### Arguments

none

### Return Value

none

### See Also

DXL\_InitVideo

### Description

Exit and de-initialize the video decompression library. Release any allocated data structures.

### Notes

Always destroy xImages **BEFORE** calling DXL\_ExitVideo; otherwise memory allocated by xImage buffers is not released.

## DXL\_GetVScreenBlitQuality

Get the active blit quality for a specified vScreen.

```
#include <duck_dxl.h>
```

```
enum BLITQUALITY DXL_GetVScreenBlitQuality( DXL_VSCREEN_HANDLE vScreen )
```

### Arguments

**vScreen** - handle to a vScreen

### Return Value

blit quality setting for vScreen

### See Also

SetVScreenBlitQuality

### Description

GetVScreenBlitQuality queries a vScreen to find out the currently active BlitQuality. BlitQuality determines if any stretching is performed on images, as they are blitted.

BLIT\_SAME is a direct transfer,

BLIT\_STRETCH performs pixel doubling and line skipping to stretch an image to twice its original dimensions (four times its size).

BLIT\_STRETCH\_BRIGHT performs pixel doubling and vertical interpolation to stretch an image to twice its original dimensions (four times its original size)

## DXL\_GetXImageColorDepth

Get color depth of an xImage.

```
#include <duck_dxl.h>
```

```
int DXL_ GetXImageColorDepth( DXL_XIMAGE_HANDLE src )
```

### Arguments

**src** - handle to a compressed image source (xImage).

### Return Value

DXRGB16 - 16bit compressed image.

DXRGB24 - 24bit compressed image.

### Description

Get the color depth of the compressed data currently held in an xImage container object. Since this is determined at compression time, this attribute cannot be set by the playback application.

## DXL\_GetXImageFrameBuffer

Get a pointer to a xImage's internal frame buffer.

```
#include <duck_dxl.h>
```

```
unsigned short *DXL_GetXImageFrameBuffer( DXL_XIMAGE_HANDLE src )
```

### Arguments

**src** - handle to a compressed image source (xImage)

### Return Value

A pointer to the xImage's internal Frame buffer

NULL if none exists or is not accessible (i.e. 24bit, 16bit intraframe and sprite frames)

### See Also

DXL\_dxImageToVScreen

### Description

This function allows applications to monitor and use a xImage's frame buffer directly (e.g. for blitting to a screen).

### Notes

16bit TrueMotion 1 and TrueMotion 2X IntERframe only

DXL\_GetXImageFrameBuffer is provided as a method for accessing the internal frame buffer of an xImage. Operations that require custom blitting and/or processing of an image before or in conjunction with their final display should use this to achieve the least amount of overhead.

To decompress an image without blitting to a vScreen use DXL\_dxImageToVscreen with the vScreen handle specifed as NULL. This performs only the decompression to the internal Frame buffer of the xImage.

IntRAframe images and Sprites do not use an internal frame buffer so this call does not apply to them. The frame buffer of a 24 bit xImage is kept in a semi-compressed form until it is actually blitted to the screen. Access to this frame buffer is disallowed for 24bit xImages.

## DXL\_GetXImageType

Get type of image contained in xImage.

```
#include <duck_dxl.h>
```

```
enum IMAGETYPE DXL_GetXImageType( DXL_XIMAGE_HANDLE src )
```

### Arguments

**src** - handle to a compressed image source (xImage).

### Return Value

Image type contained within current compressed data.

### Description

Returns IntRAframe, IntERframe or Sprite, see data structures and definitions above for a description of types.

## DXL\_GetXImageXYWH

Get xImage offset and dimensions.

```
#include <duck_dxl.h>
```

```
int DXL_GetXImageXYWH( DXL_XIMAGE_HANDLE src, int *x, int *y, int *w, int *h )
```

### Arguments

**src** - handle to a compressed image source (xImage).

**x,y,w,h** - addresses for compressed image offset and dimensions

### Return Value

0 on success or negative error code.

### Description

Retrieve the current offset and dimensions for a compressed source image. (i.e. used for figuring destination H/W sprite dimensions). All units are pixels.

## DXL\_InitVideo

Initializes video decompression services.

```
#include <duck_dx1.h>
```

```
void DXL_InitVideo( int maxVScreens, int maxXImages )
```

### Arguments

**maxVScreens** - maximum number of open virtual screens

**maxXImages** - maximum number of open compressed image sources

### Return Value

**0** - if successful

**-1** - unable to initialize library, insufficient memory available

### See Also

DXL\_ExitVideo, DXL\_CreateVScreen, DXL\_CreateXImage

### Description

Initializes video decompression services.

### Notes

At initialization, a maximum number of concurrent source container objects (xImages or compressed images) and destination objects (vScreens or virtual screens) must be determined. For simple player applications, one xImage (source) and one vScreen (destination) should be sufficient.

The numbers specified become the maximum numbers of simultaneously held objects. Objects may be created and destroyed any number of times so long as the total concurrently held number is never exceeded.

## DXL\_IsXImageKeyFrame

Checks the keyframe status of an xImage.

```
#include <duck_dx1.h>
```

```
int DXL_IsXImageKeyFrame( DXL_XIMAGE_HANDLE src )
```

### Arguments

**src** - handle to xImage.

### Return Value

1 if the xImage is a keyframe, 0 if not.

### Description

Checks the keyframe status of an xImage.

## DXL\_MoveXImage

Change xImage destination offset.

```
#include <duck_dxl.h>
```

```
int DXL_MoveXImage( DXL_XIMAGE_HANDLE src, enum OFFSETXY mode, int x, int y )
```

### Arguments

**src** - handle to a compressed image source (xImage).

**mode** - movement mode, relative or absolute.

**x,y** - new location for src compressed image when decompressed to vScreen

### Return Value

**0** - success, or negative error code.

### Description

Changes the destination offset for decompressing an xImage to a vScreen. The offset only effects the position of the xImage with the destination vScreen.

## DXL\_SetVScreenBlitQuality

Set the active blit quality for a specified vScreen.

```
#include <duck_dxl.h>
```

```
enum BLITQUALITY DXL_SetVScreenBlitQuality( DXL_VSCREEN_HANDLE vScreen, enum BLITQUALITY bq )
```

### Arguments

**vScreen** - a handle to a vScreen

**bq** - the desired blitQuality

BLIT\_SAME - no translation

BLIT\_STRETCH - lineskip and stretch to twice dimensions (4x size)

### Return Value

previous blit Quality

### See Also

DXL\_GetVScreenBlitQuality

### Description

On platforms capable of displaying a resolution 640x400 or better, the blitQuality of a vScreen can be set to better fill the screen.

## DXL\_GetVScreenView

Get viewport of vScreen.

```
#include <duck_dxl.h>
```

```
int DXL_GetVScreenView( DXL_VSCREEN_HANDLE dst, int *x, int *y, int *w, int *h )
```

### Arguments

**dst** - handle to vScreen.

**x,y,w,h** - addresses for viewport rectangle location and dimensions.

### Return Value

0 on success negative on error.

### See Also

DXL\_AlterVScreenView

### Description

Retrieve the current offset and dimensions for a vScreen's viewport rectangle. All units are pixels.





## 3.3 Audio Processing (Using Decompressors and Audio Buffers)

### 3.3.1 XAudioSrc

An XAudioSrc object contains information describing a compressed audio source. The characteristics of such an object are defined at compression time and are available via read only query functions at run time.

Data is associated with an XAudioSrc through the use of the `DXL_CreateXAudioSrc` and `DXL_AlterXAudioSrcData` functions. Unlike creating XImages however, creating an XAudioSrc also requires a pointer to a format structure that describes the format in which the data is compressed. For audio data found within a TrueMotion file this is available through the use of the `HFB_GetAudioInfo` function.

### 3.3.2 AudioDst

An AudioDst describes the format, size, location and state of an output audio buffer. AudioDsts are created by the application and maintained by the DXA library.

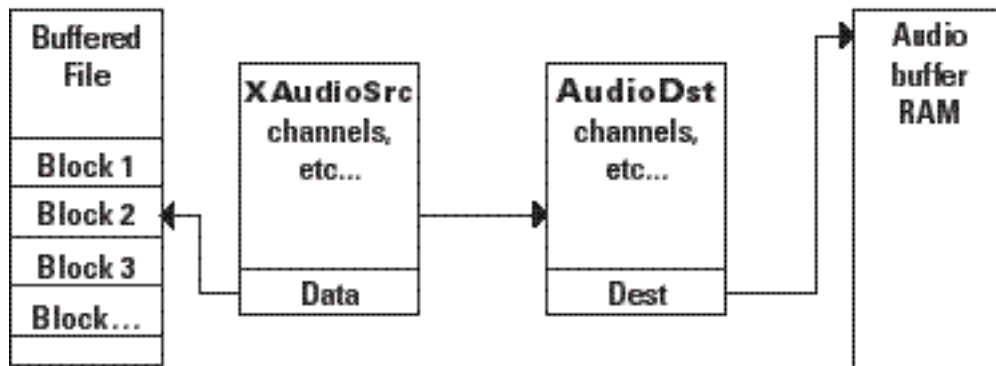


Figure 3.1 *Decompressing and Realizing Audio*

Audio decompression is identical (in process) to image decompression. Compressed audio is associated with a XaudioSrc object (Compressed Audio Source) and is resolved to a destination buffer described by an AudioDst (Audio Destination).

### 3.3.3 Loading Audio Samples

Because file buffering and data decompression are completely independent services, the application must assume the responsibility for coordinating data transfer between the independent libraries. Since audio playback is not as “granular” as video playback, the control for decompression is more complex. A typical implementation would look something like this:

*Code Sample 5 Decompressing audio*

```
/* First, during initialization get the first block of audio data and feed it to
the audio decompressor. */

aIndex = HFB_GetStreamingData(aStream, (void **)&aData, &aLength, DUK_FORWARD, 1);

DXL_AlterXAudioData(audioSrc, (unsigned char *) aData, aLength);


/* Now, during playback, tell the decompressor to decompress up to "limit" sample. This
loop will repeat until the requested number of samples is decompressed. */

while((gotten = DXL_dxAudio(audioSrc, audDst, limit)) != limit){
/* if gotten != limit there must not have been enough
samples in the chunk so get another chunk and try again */

/* reduce request by number of samples already decompressed */
limit -= gotten;

/* release the used up record and get a new record of data from HFB */

if (hMovie->aIndex > -1)

HFB_ReleaseStreamingData(xBuffer, aIndex);

hMovie->aIndex = HFB_GetStreamingData(aStream, (void **)&aData,

&aLength, DUK_FORWARD, 1);

/* now, prime the decompressor with the new data and start the next iteration */

/* so, if we successfully got some audio data, use it */

if (aIndex > -1)

DXL_AlterXAudioData(audioSrc, (unsigned char *) aData, aLength);

} // end while
```

### 3.3.4 Maintaining the Audio Hardware Buffer

Because audio buffering can be closely tied to a hardware platform, the DXA library relies on the application to keep track of “emptying” the buffer. DXA tracks only the position into which to write new data. By timing/spacing the use of `DXL_dxAudio`, the application avoids buffer starvation as well as overwriting data in the buffer.

## 3.4 DXA\_ Function and Structure Reference

### DXL\_AlterAudioDst

```
#include <duck_dxl.h>
```

```
void DXL_AlterAudioDst( DXL_AUDIODST_HANDLE dst, void *addrL, void *addrR, int length, int  
bitDepth, int numChannels, int sampleRate )
```

#### Arguments

**dst** - handle to an audio destination.

**\*addrL** - pointer to left channel buffer

**\*addrR** - pointer to right channel buffer

**length** - length (in samples) of each buffer.

**bitDepth** - bit depth of destination (8 or 16)

**numChannels** - number of audio channels (1 or 2)

**sampleRate** - in samples per second

#### Return Value

none.

#### Description

Change attributes of an audio destination. Also resets internal location pointers.

Specify 0 or NULL values for no change.

(i.e. `DXL_AlterAudioDst(dest, NULL, NULL, 0, 0, 0, 44100)` ; will change only the sample rate of the destination.)

## DXL\_AlterXAudioData

```
#include <duck_dxl.h>
```

```
void DXL_AlterXAudioData( DXL_XAUDIO_SRC_HANDLE xSource, unsigned char *addr, int length )
```

### Arguments

**xSource** - handle to a xAudio source object.

**\*addr** - pointer to new audio data

**length** - length of new audio data (bytes)

### Return Value

none.

### See Also

DXL\_CreateXAudioSrc

### Description

Change the compressed source data currently associated with an xAudio source. The supplied data must be of the same type as that used when creating the xAudioSrc; this function cannot be used to change the type of an xAudioSrc.

Setting the address of audio data to NULL causes the xAudio object to become a NULL object which is interpreted as containing an infinite number of “zero” value samples.

## DXL\_CreateAudioDst

Create audio destination object.

```
#include <duck_dxl.h>
```

```
DXL_AUDIODST_HANDLE DXL_CreateAudioDst( void *addrL, void *addrR, int length, int  
bitDepth, int numChannels, int sampleRate )
```

### Arguments

**\*addrL** - pointer to left channel buffer

**\*addrR** - pointer to right channel buffer

**length** - length (in samples) of each buffer.

**bitDepth** - bit depth of destination (8 or 16)

**numChannels** - number of audio channels

**sampleRate** - in samples per second

### Return Value

a handle to a `AudioDst` object or NULL if none available.

### Description

Create an `AudioDst` object with the provided attributes.

### Notes

When `numChannels` is equal to 2 but `addrR` is NULL, it is assumed that multi-channel samples should be interleaved (i.e. LRLRLRLR) within the single buffer pointed to by `addrL`.

## DXL\_CreateXAudioSrc

Create compressed audio object.

```
#include <duck_dxl.h>
```

```
DXL_XAUDIO_SRC_HANDLE DXL_CreateXAudioSrc( DKWAVEFORM *wv, unsigned char *addr, int length )
```

### Arguments

**wv** - pointer to a structure of type DKWAVEFORM describing the input audio type.

**addr** - pointer to audio data

**length** - length of the initial block of audio information

### Return Value

NULL unable to create audio source (max sources exceeded) or a handle to a compressed audio source.

### See Also

DXL\_AlterXAudioData

### Description

Creates a compressed audio object. Normally a xAudio source is created using information stored in a TrueMotion A/V file. Data can be from other sources.

## DXL\_DestroyAudioDst

Destroy an audio destination object.

```
#include <duck_dxl.h>
```

```
void DXL_DestroyAudioDst( DXL_AUDIODST_HANDLE dst )
```

### Arguments

**dst** - handle to an audio destination.

### Return Value

None.

### Description

Clears an audio destination object and returns it to the object pool.

## DXL\_DestroyXAudioSrc

Destroy an audio source object.

```
#include <duck_dxl.h>
```

```
void DXL_DestroyXAudioSrc( DXL_XAUDIO_SRC_HANDLE xSource )
```

### Arguments

**xSource** - handle to a compressed audio source object.

### Return Value

none.

### See Also

DXL\_CreateXAudioSrc

### Description

destroys an xAudio source and returns all of its internal structures to the DXL\_Audio pool.

## DXL\_dxAudio

Decompress audio samples.

```
#include <duck_dxl.h>
```

```
int DXL_dxAudio( DXL_XAUDIO_SRC_HANDLE src, DXL_AUDIO_DST_HANDLE dst, int limit )
```

### Arguments

**src** - handle to a compressed audio source (xAudioSrc).

**dst** - handle to a audio destination (AudioDst).

**limit** - maximum number of samples to transfer from source to dest.

### Return Value

Number of samples actually transferred or a negative error code.

### Description

Decompress and transfer up to a given number of samples from an xAudioSrc to an AudioDst.

### Notes

The number of samples transferred is controlled two factors, one is the limit parameter, the other is the number of remaining samples in the xAudioSrc. If the function returns a result less than the desired number of samples, get another audio record (via HFB\_GetStreamingData( )) for source data and try again.)



## DXL\_ExitAudio

De-initialize audio decompression services.

```
#include <duck_dxl.h>
```

```
void DXL_ExitAudio( void )
```

### Arguments

None.

### Return Value

None.

### See Also

DXL\_InitAudio

### Description

De-initialize library and free allocated objects

## DXL\_InitAudio

Initialize audio decompression service.

```
#include <duck_dxl.h>
```

```
int DXL_InitAudio( int maxSrcs, int maxDsts )
```

### Arguments

**maxSrcs** - maximum number of simultaneous sources

**maxDsts** - maximum number of simultaneous destinations

### Return Value

**0** - success or negative error code.

### See Also

DXL\_ExitAudio()

### Description

Allocates and initializes objects for audio libraries.

This function allocates memory.

## 3.5 File management using HFB

The High-speed File Buffering Library (hereafter HFB) provides high performance file buffering and TrueMotion A/V parsing. HFB functions are prefixed by `HFB_`.

A TrueMotion file object is a container object that holds information not only about where information can be found on a disk, but also about what types of information the file contains and how the file is organized. Files can be buffered or un-buffered, but for practical reasons some buffering (e.g., index management) is almost always necessary.

Buffers are used to provide efficient access to data stored in TrueMotion files.

Streams are logical collections of data spread out according to a timeline. Streams are generally interleaved with other streams of varying types to provide for efficient usage of physical disk, file and/or network services.

Data records are one of the smallest atoms of data within a TrueMotion file. A data record will generally consist of a single compressed video frame or a chunk of compressed audio data. (Audio data records may consist of multiple audio chunks or samples). Data records can either be read directly from disk (unbuffered) or streamed within a file.

---

**Note:** When reading data from a stream of from disk it is important to note that records (frames) begin at (index) 0.

---

### Opening a TrueMotion file and its Streams

Before opening a TrueMotion file, a buffer must be created for holding at least the file's index that provides the libraries with look up information for the file. Once a buffer is created a file can be opened and associated with it. At this point, no data has yet been read from the file. `HFB_InitBuffer` is called to perform the initial data fill.

The following example demonstrates opening a simple TrueMotion file containing a single audio stream and a single video stream.

*Code Sample 6 Creating HFB Objects*

```
{

/* get a file and buffer */

movie->xBuffer = HFB_CreateBuffer(movie->HFB_bufferSize, 0);

movie->xFile = HFB_OpenFile(movie->fileName, movie->xBuffer);

/* get first audio and first video stream handles */

movie->vStream = HFB_GetStream(movie->xFile, NULL, 1, DUK_VIDSTREAM);

movie->aStream = HFB_GetStream(movie->xFile, NULL, 1, DUK_AUDSTREAM);

}
```

After opening the file, `HFB_GetStream` is used to get reference handles to the streams within the file.

When finished using a stream, the application should use `HFB_ReleaseStream` to release the stream reference object. The stream itself is unaffected by this call and new references to it may be subsequently obtained.

When finished using a file and all its streams, the application must use `HFB_CloseFile` to close the physical disk file and de-allocate work ram. Any further access to the file will be invalid unless the file is re-opened.

Applications should use `HFB_DestroyBuffer` to release all structures and work ram associated with a buffer. After calling `HFB_DestroyBuffer`, the buffer cannot be reused; destruction is an irrevocable process.

### **HFB\_Init – Starting the streaming process**

`HFB_Init` moves the stream pointers to a specified frame number and begins data streaming. The amount of preloading necessary will be determined by the average and peak data rates of the file being played. This pre-loading is largest contributing factor to the amount of time between opening a file (or repositioning the file location) and playback. It is recommended that at least 1 sec worth of data (according to the average data rate of the file) be pre-loaded to cover pre-loaded audio and the initial frames of the file. More pre-buffering will help smooth out any data-rate peaks in the file but will slow down load/start.

See the description of `HFB_InitBuffer` for specific information.

### **Managing Streamed and/or unbuffered Data**

After opening a reference to a stream with `HFB_GetStream`, data can be read from that stream using one of two methods.

#### *Unbuffered access*

Unbuffered reading is considerably slower than buffered reading and is recommended only in instances where random access to a file is required (e.g. random scanning through a file). Un-buffered is read as follows:

```
errFlag = HFB_ReadData(xVideoStream,data,&length, DUK_FORWARD, 1);
```

#### *Buffered access (Data Streaming)*

Buffered or “streaming” data is read from disk automatically by `HFB_InitBuffer` and/or `HFB_FillBuffer`. Indices to data blocks are retrieved as follows:

```
index = HFB_GetStreamingData(xVideoStream,&data,&length, DUK_FORWARD, 1);
```

Because the reading and usage of buffered data are asynchronous, a means for marking blocks of data used must be provided so that memory occupied by them can be recycled for buffering new data.

Both of these examples used `DUK_FORWARD, 1` to retrieve the next block from the stream. See the [HFB Function Reference](#) for more information about stream pointer movement modes.

When no longer needed (as soon as possible), Data Blocks should be marked for release:

```
HFB_ReleaseStreamingData(xBuffer,index);
```

If data blocks are not released, they will stagnate in the buffer and prevent new data being read. Normally, every call to `HFB_GetStreamingData` must be balanced by a call to `HFB_ReleaseStreamingData`.

### **Avoiding stalls during streaming**

In addition to pre-loading disk information during initialization (see `HFB_InitBuffer`) it is necessary to provide points during playback at which resources can be recycled for loading with new data. This is normally accomplished by calling `HFB_FillBuffer`.

During the creation and initialization of a buffer, internal information is calculated using a combination of frame rate, CD data rate, file size, hardware buffer size and other parameters that allow the HFB\_library to read data from disk files in a very efficient manner. If HFB\_FillBuffer is called at approximately the dominant frame rate (frame rate of the first video stream) of the file it should always return with minimal (if any) latency. For example:

Given a file with a dominant frame rate of 20 frames per second, HFB\_Fillbuffer will read roughly 15K per call to match the CD transfer rate of approximately 300K/sec. Internally however, HFB\_FillBuffer maintains tables of effective transfer rate that can be affected by changes in the size of frames, size of sectors, hardware buffering, empty buffer space, etc... so the actual amount read may vary. Additionally, HFB\_FillBuffer always reads in full sector multiples to maximize performance.

During streaming, data from a file is read directly into an HFB\_Buffer (via HFB\_InitBuffer and HFB\_FillBuffer). There the data is ordered in a circular fashion which allows mapping the buffer directly to the physical layout of the file. This mapping greatly enhances flexibility and performance.

When blocks of data are needed for decompression/playback, HFB\_GetStreamingData retrieves a pointer to (via a passed (void \*\*) parameter) a block within the HFB\_Buffer along with the block's length and its index reference. The function's return value is this index reference or "handle".

No movement (copying) of data results from HFB\_GetStreamingData. DXL functions are performed on data "in place" within the HFB\_Buffer. By eliminating data movement, further efficiency is gained.

In order for HFB to reclaim space within the HFB\_Buffer, blocks of data must be released using: HFB\_ReleaseStreamingData. This function marks the block in the index as used/invalidated. This allows HFB\_FillBuffer to recycle the space occupied by the record.

**Sample Parse loop***Code Sample 7 Exercising HFB*

```
{  
  
/* get a file and buffer */  
  
xBuffer = HFB_CreateBuffer(512*1024, 0); /* create a 512K buffer */  
xFile = HFB_OpenFile("sample.avi", xBuffer); /* open file w/buffering */  
  
/* get audio and first video stream handles */  
vStream = HFB_GetStream(xFile, NULL, 1, DUK_VIDSTREAM);  
aStream = HFB_GetStream(xFile, NULL, 1, DUK_AUDSTREAM);  
  
/* get StreamInfo */  
DKWAVEFORM *wavPtr = &(HFB_GetStreamInfoPtr(aStream)->a;  
DKBITMAP *bmpPtr = &(HFB_GetStreamInfoPtr(vStream)->a;  
  
HFB_InitBuffer(xBuffer, xFile, 0, -1); /* preload * of the buffer */  
do{ /* loop through all the data in the file */  
HFB_FillBuffer(xBuffer,HFB_USE_DEFAULT,HFB_IGNORE_COUNT);  
  
if((vidIndex =  
HFB_GetStreamingData(vStream,&data,&length, DUK_FORWARD, 1)) > -1){  
/* process video record (frame) */  
HFB_ReleaseStreamingData(xBuffer,vidIndex);  
}  
  
if ((audIndex =  
HFB_GetStreamingData(aStream,&data,&length, DUK_FORWARD, 1)) > -1){  
/* process audio record here */  
HFB_ReleaseStreamingData(xBuffer,audIndex);  
}  
  
}while(vidIndex != -1); /* continue until no video records left */  
HFB_ReleaseStream(vStream); /* release video Stream */  
HFB_ReleaseStream(aStream); /* release audio Stream */  
HFB_CloseFile(xFile) /* close the open file */  
HFB_DestroyBuffer(xBuffer); /* destroy the buffer object */  
}
```

## 3.6 HFB\_ Function and Structure Reference

### HFB\_CloseFile

Close a TrueMotion file.

```
#include <duck_hfb.h>

void HFB_CloseFile( HFB_FILE_HANDLE FileHandle )
```

#### Arguments

**FileHandle** - a handle to an HFB file object.

#### Return Value

None.

#### See Also

HFB\_OpenFile

#### Description

Close a TrueMotion file and release file structures.

## HFB\_CreateBuffer

Create a High-speed File Buffer.

```
#include <duck_hfb.h>
```

```
HFB_BUFFER_HANDLE HFB_CreateBuffer( long bSize, long reserved )
```

### Arguments

**bSize** - size (in bytes) of buffer to allocate.

**reserved** - must be 0.

### Return Value

A handle to a HFB buffer object.

**NULL** - no buffer objects available.

### See Also

`HFB_InitBuffer`, `HFB_DestroyBuffer`

### Description

This function is used to create a buffer object

### Notes

A good rule for the size of a buffer is enough space for 1 sec of A/V + 32-64K for indexing. (i.e. for a 300K per second file 364K should be more than adequate in most cases)

The largest contributing factor to buffer size requirements is the A/V file's peak data rate. If a file maintains a low data rate (i.e. always less than the capacity of the drive) less buffer space is necessary for playback (i.e. 128-256K should suffice). The best way to determine the optimum buffer size for a given file is by experimentation, start with a fairly large buffer and cut back until performance is noticeably affected (i.e. audio synchronization, hanging frames).

To figure the approximate space used for indexing, 8 bytes are used for each index entry (16 bytes per frame including audio).

The second parameter is reserved for future use and should be specified as 0.

## HFB\_DestroyBuffer

Destroy buffer for re-use by library/system.

```
#include <duck_hfb.h>
```

```
void HFB_DestroyBuffer( HFB_BUFFER_HANDLE Buffer )
```

### Arguments

**Buffer** - handle to a created buffer

### Return Value

none

### See Also

HFB\_CreateBuffer

### Description

Frees memory used by a buffer and releases buffer object.

## HFB\_Exit

De-initialize/free HFB library and resources.

```
#include <duck_hfb.h>
```

```
void HFB_Exit( void )
```

### Arguments

None.

### Return Value

None.

### See Also

HFB\_Init

### Description

free any allocated structures, close all files, etc...



## HFB\_Init

## Initialize HFB library resources.

```
#include <duck_hfb.h>
```

```
int HFB_Init( int MaxOpenFiles, int MaxOpenStreams, int MaxOpenBuffers )
```

### Arguments

**MaxOpenFiles** - maximum number of concurrently open file objects.

**MaxOpenStreams** - maximum number of concurrently open streams.

**MaxOpenBuffers** - maximum number of concurrent buffer objects.

### Return Value

**0** - success or negative return code.

### See Also

HFB\_Exit

### Description

Allocate and initialize required data structures. If all three parameters are -1, HFB will dynamically allocate objects as needed

This function allocates memory.

## HFB\_FillBuffer

Fill/recycle empty space in buffer from disk.

```
#include <duck_hfb.h>
```

```
int HFB_FillBuffer( HFB_BUFFER_HANDLE bfHnd, long MaxToRead, long elapsedFrames )
```

### Arguments

**bfHnd** - handle to a buffer object.

**MaxToRead** - maximum number of bytes to read during this call.

**HFB\_USE\_DEFAULT** to use internal calculated values.

**elapsedFrames** - number of frames elapsed since start of play,

**HFB\_RESET\_COUNT** to reset internal counter.

**HFB\_IGNORE\_COUNT** to ignore internal counter.

### Return Value

Number of bytes actually read from the disk file into the buffer or a negative error code.

### See Also

HFB\_CreateBuffer

### Description

The HFB\_FillBuffer() function reads additional data from a file into space invalidated by HFB\_ReleaseStreamingData() calls or any empty buffer space available.

For best results, call this function once per frame with the elapsedFrames parameter set to DUK\_IGNORE\_COUNT.

### Notes

Using an elapsed frame counter: In order to avoid having to call this routine at exactly the frame rate do the following:

- 1) reset the internal frame counter by calling HFB\_FillBuffer with the elapsedFrames parameter set to DUK\_RESET\_COUNT
- 2) call HFB\_FillBuffer as often as possible, passing it an accurate elapsed frame count.

The function will use the elapsedFrames parameter in conjunction with internally computed values to determine the amount to read from the file in order to avoid waiting for data to become available.

## HFB\_GetAudioInfo

Get general information describing an audio stream.

```
#include <duck_hfb.h>
```

```
const DKWAVEFORM *HFB_GetAudioInfo( HFB_STREAM_HANDLE aStream, int *lNumChannels, int  
*lSamplesPerSec, int *lBytesPerSec, int *lwFormat )
```

### Arguments

**aStream** - a handle to an HFB (audio) stream object.

**lNumChannels** - pointer to an integer in which to place the number of channels in the specified audio stream.

**lSamplesPerSec** - pointer to an integer in which to place the number of samples per second in the specified audio stream.

**lBytesPerSec** - pointer to an integer in which to place the number of actual bytes per second in the specified audio stream.

**lwFormat** - pointer to an integer in which to place the type of audio in the specified audio stream. 1 = PCM, otherwise DK3/4 bit.

### Return Value

Pointer to a DKWAVEFORM structure describing the audio information contained in the specified stream.

NULL if the stream is of unknown or non-audio type.

### See Also

HFB\_GetSamplesPerFrame, HFB\_GetFrameRates

### Description

Gets general information describing an audio stream.

## HFB\_getBufferStatus

Get status of a buffer.

```
#include <duck_hfb.h>
```

```
int HFB_getBufferStatus( HFB_BUFFER_HANDLE bfHnd )
```

### Arguments

**bfHnd** - handle to a buffer object

### Return Value

**0** - buffer OK

**1** - buffer has reached end of file.

### See Also

HFB\_CreateBuffer

### Description

HFB\_getBufferStatus is used to determine if a buffer has read the end of a file.

## HFB\_getDataPosition

Get starting position of data block (in samples).

```
long HFB_getDataPosition( HFB_STREAM_HANDLE StreamHandle, HFB_DATA_HANDLE DataIndex )
```

### Arguments

**StreamHandle** - handle to a stream object.

**DataIndex** - Index to a data record within a stream use -1 to find position of first available record in the buffered stream

### Return Value

Longword starting position of the data record within the stream, expressed in audio samples for audio streams and frames for video streams.

### See Also

HFB\_GetStreamingData

### Description

Once a record is found within a stream, its exact starting location can be found using this function. This is useful for resolving differences between streams when starting from a position other than the beginning of the file.

*Code Sample 8 Using HFB\_getDataPosition to resolve startup audio synchronization*

```
if (frame > 0){

int skipAudio = 0;

/* first figure frame starting position in audio samples */
skipAudio = (frame*SamplesPerSec)/GlobalFrameRate;

/* subtract the position of the available audio record */
skipAudio -= HFB_getDataPosition(xAudioStream,-1);

if (skipAudio > 0){

/* if samples to skip then skip by decompressing */

loadSamples(skipAudio);

/* reset the output buffer to start (write over skipped)*/

DXL_AlterAudioDst(audioDst,NULL,NULL,0,0,0,0);

}

}
```

## HFB\_GetDataSize

Get size of data block (in bytes).

```
#include <duck_hfb.h>
```

```
int HFB_GetDataSize( HFB_STREAM_HANDLE StreamHandle, enum FTYPE format, int frameNum )
```

### Arguments

**StreamHandle** - handle to a stream object

**format** - type of search to perform (frameNum index)

**frameNum** - frame number or handle (index) of desired record

### ReturnValue

HFB\_GetDataSize returns the size of the data record within the stream, expressed as bytes

### Description

get size of data block (in bytes)

### See Also

HFB\_GetTotalDataSize

## HFB\_GetFileInfo

Get a pointer to file information.

```
#include <duck_hfb.h>
```

```
const HFB_FILE_INFO *HFB_GetFileInfo( HFB_FILE_HANDLE FileHandle )
```

### Arguments

**FileHandle** - a HFB file object handle.

### Return Value

a pointer to a HFB\_FILE\_INFO structure describing the indicated file.

### See Also

HFB\_OpenFile

### Description

Used to read information about an opened TrueMotion File.

## **HFB\_GetFrameRates**    Get frame rate related information based on a combination of an audio and video stream.

```
#include <duck_hfb.h>
```

```
void HFB_GetFrameRates( HFB_STREAM_HANDLE vStream, HFB_STREAM_HANDLE aStream, int  
*lGlobalFrameRate, int *prebuffer )
```

### **Arguments**

**vStream** - handle to a video stream.

**aStream** - handle to an audio stream

**lGlobalFrameRate** - pointer to int to receive frame rate of dominant stream (video)

**prebuffer** - number of audio samples appearing before first video frame in file

### **Return Value**

none

### **See Also**

HFB\_GetAudioInfo, HFB\_GetSamplesPerFrame

## HFB\_GetIndexFlags

Get index flags for data block.

```
int HFB_GetIndexFlags( HFB_BUFFER_HANDLE StreamHandle, enum FFORMAT format, int frameNum );
```

### Arguments

**StreamHandle** - handle to a stream object.

**format** - describes the format the next parameter is to take

HFB\_FRAMENUM - frameNum specifies a frame number in a stream

HFB\_INDEXNUM - frameNum specifies an index number returned by

HFB\_GetStreamingData(...)

**frameNum** - see above format

### Return Value

**>= 0** data block flags for data block specified

**-1** - frameNum is out of index range

**-2** - frameNum is out of frame range

### See Also

HFB\_ReadData, HFB\_GetStreamingData

### Description

get index flags for data block

*flags used:*

```
#define HFB_DATA_READ 0x01
```

```
#define HFB_DATA_RELEASED 0x02
```

```
#define HFB_DATA_KEYFRAME 0x08
```

*Code Sample 9 Unbuffered read and decompression of keyframes*

```
int flags,length;
```

```
char data[128*1024];
```

```
do{
```

```
while(!((flags = HFB_GetIndexFlags(xStream, HFB_FRAMENUM, frameNum)) &  
HFB_DATA_KEYFRAME))
```

```
frameNum++;
```

```
if (flags < 0) break;
```

```
length = sizeof(data);
```

```
HFB_ReadData(xStream,data,&length,DUK_ABSOLUTE,frameNum);
```

```
DXL_AlterXImageData(xImage,data);
```

```
DXL_dXImageToVScreen(xImage,vScreen);
```

```
}while(1);
```

## HFB\_GetSamplesPerFrame

Get number of audio samples per video frame.

```
#include <duck_hfb.h>
```

```
int HFB_GetSamplesPerFrame( HFB_STREAM_HANDLE vStream, HFB_STREAM_HANDLE aStream )
```

### Arguments

**vStream** - a handle to an HFB (video) stream object.

**aStream** - a handle to an HFB (audio) stream object.

### Return Value

The number of samples from the audio stream occurring within a single frame interval of the video stream.

### See Also

HFB\_GetAudioInfo, HFB\_GetFrameRates

### Description

Given a video stream, vStream and an audio stream, aStream, this function returns the number of audio samples that should be played within the duration of a single video frame.



## HFB\_GetStream

Get a handle to a stream.

```
#include <duck_hfb.h>
```

```
HFB_STREAM_HANDLE HFB_GetStream( HFB_FILE_HANDLE FileHandle, char *StreamName, int  
StreamNum, HFB_StreamType Stype )
```

### Arguments

**FileHandle** - a handle to an HFB file object.

**StreamName** - a zero terminated string containing the name of a stream within the specified file. Specify NULL to ignore this parameter.

**StreamNum** - an absolute stream number or the nth occurring stream of the specified type.

**Stype** - the type of stream to be opened.

### Return Value

a handle to stream container object

NULL - pool of stream container objects has been exhausted.

### See Also

HFB\_ReleaseStream, HFB\_OpenFile

### Description

get a handle to a stream within a TrueMotion/S A/V file by name, number or type.

### Notes

For example, to get a handle to the 2nd video file the application would use:

```
hnd = HFB_GetStream(fHnd, NULL, 2, DUK_VIDSTREAM);
```

to get the second stream from the file, regardless of type:

```
hnd = HFB_GetStream(fHnd, NULL, 2, DUK_UNDEFINED);
```

A single stream may be opened more than once.

## HFB\_GetStreamInfo

Get pointer to stream info.

```
#include <duck_hfb.h>
```

```
const HFB_STREAM_INFO *HFB_GetStreamInfo( HFB_STREAM_HANDLE StreamHandle )
```

### Arguments

**StreamHandle** - a handle to an HFB stream object.

### Return Value

Pointer to a HFB\_STREAM\_INFO structure describing the stream associated with the specified handle.

### See Also

HFB\_GetStream

### Description

Get information about a stream.

## HFB\_GetStreamingData

Get buffered data block from stream.

```
HFB_DATA_HANDLE HFB_GetStreamingData( HFB_STREAM_HANDLE StreamHandle, void **DataPointer,
long *Length, enum dukDirect Direction, int Count )
```

### Arguments

**StreamHandle** - handle to a stream object.

**DataPointer** - pointer to a pointer in which to place the address of the compressed data record (position within the HFB\_Buffer)

**Length** - pointer to a long variable in which to place the length of the compressed data

**Direction** - direction in which to read records

**Count** - number of the requested record.

### Note

absolute references begin at 0.

### Return Value

**>= 0** - a handle to a compressed data block.

**-1** - request out of range/no additional records available.

**-2** - block exists in file but is not available in buffer.

Usually caused by an unreleased block or buffer starvation.

**-4** - block has been released from use (presumably by HFB\_Release...)

### See Also

HFB\_ReadData, HFB\_ReleaseStreamingData

### Description

Get a handle to a compressed record in a buffered file.

### Notes

If a file does not completely fit within a buffer, each call to HFB\_GetStreamingData( ) must be balanced by a corresponding call to HFB\_ReleaseStreamingData( ) in order to allow the system to reuse the space occupied by no-longer needed records.

No data is moved as a result of HFB\_GetStreamingData( ). The value filled into \*\*DataPointer is a location within the HFB streaming buffer. Since all DXL functions are non-destructive, data can be decompressed without moving it from the HFB streaming buffer.

## HFB\_InitBuffer

Initialize buffer and pre-load data from disk file.

```
#include <duck_hfb.h>
```

```
void HFB_InitBuffer( HFB_BUFFER_HANDLE Buffer, HFB_FILE_HANDLE File, int startFrame, long  
preload )
```

### Arguments

**Buffer** - a handle to an HFB buffer object.

**File** - a handle to an HFB file object.

**startFrame** - frame at which to begin playback. Normally set to 0.

**preload** - amount of buffer to preload with data (specified in bytes)

-1 - fill buffer three quarters full

-2 - fill entire buffer

### Return Value

the number of samples from the audio stream occurring within a single frame interval of the video stream.

### See Also

HFB\_CreateBuffer, HFB\_FillBuffer

### Description

After creating a buffer and opening a file, an application must initialize the buffer with data.

### Notes

The preloadAmount can greatly affect the time required between starting the player and the start of playback. Remember that 2x CD readers generally have a max data rate of 300KBS and a seek time on the order of 300ms. By keeping your average data rate under 300K, less buffering and preloading will be necessary and files will load and start faster.

## HFB\_LoadIndex

Load a TrueMotion File's Index.

```
#include <duck_hfb.h>
```

```
void HFB_LoadIndex( HFB_FILE_HANDLE fHnd, HFB_BUFFER_HANDLE bfHnd )
```

### Arguments

**fHnd** - handle to an opened file.

**bfHnd** - handle to a created buffer (see HFB\_CreateBuffer()).

### Return Value

none

### See Also

HFB\_OpenFile, HFB\_FindFile, HFB\_ParseFile

### Description

Load a TrueMotion file's index into the specified buffer object.

Must be used in this order:

```
HFB_FindFile, 2) HFB_ParseFile, 3) HFB_LoadIndex
```

This function allocate memory for temporary use (i.e. for use within this function only)

## HFB\_FindFile

Initiate find and open of a TrueMotion File.

```
#include <duck_hfb.h>
```

```
HFB_FILE_HANDLE HFB_FindFile( char *FileName )
```

### Arguments

**FileName** - name of file to open.

### Return Value

Handle to a TrueMotion file object.

NULL - the pool of file container objects has been exhausted.

### See Also

HFB\_OpenFile, HFB\_ParseFile, HFB\_LoadIndex

### Description

This function implements a portion of the functionality performed by HFB\_OpenFile. The basic actions performed by HFB\_FindFile are to allocate a HFB\_File object and send the file and/or CD-subsystem a command to open a file via the duck\_open function.

HFB\_OpenFile parses the file's header immediately after opening which may block while the I/O-system finds and opens the file.

In contrast, HFB\_FindFile returns immediately after requesting the file be opened and buffered. No data is read until HFB\_ParseFile is called.

Must be used in this order:

1) HFB\_FindFile, 2) HFB\_ParseFile, 3) HFB\_LoadIndex

## HFB\_OpenFile

Open a TrueMotion File.

```
#include <duck_hfb.h>
```

```
HFB_FILE_HANDLE HFB_OpenFile( char *FileName, HFB_BUFFER_HANDLE bfHnd )
```

### Arguments

**FileName** - name of file to open.

**bfHnd** - handle to a created buffer (see HFB\_CreateBuffer()).

### Return Value

Handle to a TrueMotion file object.

NULL - pool of file container objects has been exhausted.

### See Also

HFB\_CloseFile

### Description

This function opens a TrueMotion compressed file and readies it for playback. The buffer object provided is used for buffering and streaming the file.

HFB\_ functions cannot be used on file opened without specifying a buffer. Only one file may be associated with a given buffer at one particular time.

## HFB\_ParseFile

Parse a TrueMotion File's header and initiate index loading.

```
#include <duck_hfb.h>
```

```
void HFB_ParseFile( HFB_FILE_HANDLE fHnd, HFB_BUFFER_HANDLE bfHnd )
```

### Arguments

**fHnd** - handle to a "found" file (see HFB\_FindFile()).

**bfHnd** - handle to a created buffer (see HFB\_CreateBuffer()).

### Return Value

none.

### See Also

HFB\_OpenFile, HFB\_FindFile, HFB\_LoadIndex

### Description

After a file has been found and at least single sector is buffered, it's header can be parsed for the information necessary to describe what the file contains.

The combination of file loading functions must follow this order:

1) HFB\_FindFile, 2) HFB\_ParseFile, 3) HFB\_LoadIndex

## HFB\_ReadData

Non-buffered stream read.

```
#include <duck_hfb.h>
```

```
int HFB_ReadData( HFB_STREAM_HANDLE StreamHandle, void *data, long *maxLength, enum  
dukDirect direction, int count )
```

### Arguments

**StreamHandle** - a handle to an HFB stream object.

**data** - a pointer to a buffer in which to place the compressed data.

**maxLength** - a pointer to a long describing byte length of the specified buffer. Also used to return the actual number of bytes read.

**direction** - direction in which to seek data

**count** - number of data records to move before reading.

### Note

absolute references begin at 0.

### Return Value

0 on success or negative error code..

### See Also

HFB\_GetStream, HFB\_GetStreamingData

### Description

Perform an unbuffered read from the specified stream. Any record at any position within the file can be retrieved using this function. Because it is an unbuffered read, it may take considerably longer to get data than with a buffered read.



## HFB\_ReleaseStream

Release stream object for reuse.

```
#include <duck_hfb.h>
```

```
void HFB_ReleaseStream( HFB_STREAM_HANDLE StreamHandle )
```

### Arguments

**StreamHandle** - a handle to an HFB stream object

### Return Value

None

### See Also

HFB\_GetStream

### Description

HFB\_ReleaseStream releases a stream description object into the object pool.

HFB\_ReleaseStream does not affect the stream itself or the data within.

## HFB\_ReleaseStreamingData

Release data block (chunk) for recycling.

```
#include <duck_hfb.h>
```

```
void HFB_ReleaseStreamingData( HFB_BUFFER_HANDLE bfHnd, HFB_DATA_HANDLE DataHandle )
```

### Arguments

**bfHnd** - handle to a buffer object.

**DataHandle** - handle of data record to be released

### See Also

HFB\_GetStreamingData

### Description

HFB\_ReleaseStreamingData( ) invalidates data records in a buffered file/stream. Data records cannot be re-used after being released. Data records must be released in order to make room for new records to be read from the I/O system

When a file is entirely pre-loaded and/or random or backward play is required, do not use

HFB\_ReleaseStreamingData( ) which invalidates the records

## HFB\_SetBufferMode

Set buffer repeat(loop) mode.

```
#include <duck_hfb.h>
```

```
HFB_Modes HFB_SetBufferMode( HFB_BUFFER_HANDLE Buffer, HFB_Modes newMode )
```

### Arguments

**Buffer** - handle to a created buffer

**newMode** - new buffering mode to assign to file.

### Return Value

Previously set or default buffering mode.

### See Also

HFB\_CreateBuffer

### Description

Sets the mode for the specified buffer. Unless specifically set using this function, buffer mode defaults to HFBMODE\_NORMAL. See data types and definitions above for supported buffer modes.