Stradis™

Professional 4:2:2 MPEG-2 Decoder

SDM050, SDM250 and SDM250

Application Program Interface

for Windows® 98 and Windows NT®

Introduction/Overview

The Stradis Professional MPEG-2 Decoder Board and API provide everything needed to decode all standard MPEG streams at full 4:2:2 profile or MP@ML (4:2:0) up to 50 Mbits per second. MPEG-1 Elementary and System Streams and MPEG-2 Elementary, Packetized Elementary (PES), Transport and Program Streams are supported.

The Transport, Program and System streams are processed by the host processor in the API. PES packets (or MPEG-1 System Packets) are extracted from the stream and placed into a PC-based rate buffer before being block transferred to the decoder hardware. The de-multiplexing processing takes only small amount of the host processor's time. The decoding of the PES packets (or MPEG-1 System Packets) is performed by hardware on the decoder card. The hardware is also capable of decoding a stereo MPEG-2 audio PES stream in parallel with an MPEG video PES stream.

Audio/Video synchronization of the streams can be performed with the stream level PCR, video PTS or audio PTS as the master. Streams can also be played with no synchronization. The audio and video clocks are always phase-locked together to maintain a fixed synchronization between the audio and video. In other words, no video frames need be skipped or repeated once synchronization is established if the original stream was synchronized. The hardware also contains a VCXO (Voltage Controlled Crystal Oscillator) so that real-time clock recovery can be performed and the decoder can be synchronized to a real-time video feed.

This API is structured as a hierarchy that allows the programmer to interface software to the Stradis Professional MPEG-2 Decoder Card at any level from playing a user specified file to setting low level registers on the decoder card. The use of high level functions like playback of an MPEG encoded disk file does not require any knowledge of the lower levels of the API. The lower levels are provided for implementing functions that are not contained in the higher level functionality.

In addition to the hierarchy, the API contains two methods of accessing its functionality. The first method is a C++ interface. This requires the use of the MSVC C++ compiler. The other method is a "C" style interface that uses stdcall calling convention. This is the same calling convention used to call WIN API functions. This interface is appropriate when using a program language other than MSVC C++.

The API contains three levels of interface for decoding streams. The three interfaces (in order of increasing level) are known as the *RateBuffer*, *Demux* and *File* interface and each higher level builds on the lower level. The lowest level is the *RateBuffer* interface. The *RateBuffer* interface contains the PC-based video and audio rate buffers. The rate buffers can contain elementary stream, packetized elementary stream (PES) or system packet data. If elementary stream data is used, no synchronization is performed. The PES or packet data must be used in order for hardware synchronization to take place. The next higher level is the *Demux* interface. The *Demux* interface de-multiplexes a single stream and places the result into the rate buffers. The *Demux* interface can handle Transport, Program, System, PES and Elementary Streams. The highest level is the File interface, which provides all of the mechanisms to open and play files from a disk, CD-ROM, DVD or network file. With regard to the DVD, there is no mechanism for de-encrypting encrypted DVD files. The File interface also provides for seamless back-to-back playback of multiple files.

The *RateBuffer* and *Demux* interfaces can be used in either a real-time or playback mode. In the real-time mode, there is a mechanism to adjust the master clock to match the transmitted clock. The playback mode is used to decode files from a disk drive. The default mode is the playback mode.

In addition to the interfaces described above, a synchronization interface (CStradisSynchronize class) can be used to synchronize multiple boards. There can be multiple instances of the synchronization class to control mutually exclusive groups of decoders. The synchronization

class does not need to be used if multiple cards are not being used or there is no need to synchronize them.

The API also provides interfaces for the OSD (On Screen Display) and VBI (Vertical Blanking Interval) as well as interfaces for the various subsystem of the decoder.

Note that the API supports up to eight Stradis decoder cards in one system.

The "C++" Interface Method

The C++ interface method is comprised of four major classes. The *CStradisScan*, *CStradisDecoder*, *CStradisDecoderSettings* and the *CStradisSynchronize* class. One other class is provided to create buffers that can be directly DMAed to the decoder for use with the Still Picture interface. This class is the *CSpciDmaMem* class. Unless, the application is using the Still Picture interface, this class is not needed.

No matter what API level is being used, the *CStradisDecoder* class must be created. The *CStradisDecoder* class contains most of the functionality of the API. In order to create the *CStradisDecoder* class, a card descriptor must be obtained. The card descriptor is obtained from the *CStradisScan* class. Once the *CStradisDecoder* class is created, the functionality of the API is accessed through the *CStradisDecoder* class and the *CStradisScan* class is no longer needed.

The CStradisDecoderSettings class provides a mechanism for maintaining a set of parameters that specify how the decoder will function. This includes things like whether the card is in NTSC or PAL mode, what video line the decoded output will start on, etc.

The *CStradisSynchronize* class works with the *File* interface and is used to synchronously start and stop multiple cards in a frame accurate manner.

The *CStradisScan* Class

This class is used to get the card descriptor for a specific Stradis Decoder card. After the class is constructed, the function *GetCardsFound()* can be used to get the number of Stradis cards found.

CStradisDecoderSettings Class

The *CStradisDecoderSettings* class is used to provide the various initialization routines with information about how to initialize the decoder. Some of the member variables of the class are set by the *File_Open()* function. The default values after construction of the class are shown next to the member variable name. Not all of the member variables of the *CStradisDecoderSettings* class are used by every initialization functions. Some variables are only used during the *Decoder_Init()* function, some during the *Demux_Init()* function and some during the *RateBuffer_Init()* function and so on. Table 1 indicates the variables used by each initialization function.

Initializing the *CStradisDecoder* Class

The *CStradisDecoder* class is used to gain access to the Stradis Professional MPEG-2 Decoder. Before the card can be used, the *CStradisDecoder* class must be created and the *Decoder_Init()* function called. The *Decoder_Init()* Function must be called before any other function in the *CStradisDeocder* class can be called. The constructor for the *CStradisDecoder* class takes a single argument. This argument is a descriptor for the card. The descriptor must be obtained from the *CStradisScan* class.

Error Handling

Anytime that an error occurs during a call to one of the *CStradisDecoder* class functions, an error variable is set inside the class. Functions are used to retrieve and interpret these errors. During the debugging phase, a flag can be set that causes all errors to display an *AfxMessageBox* with the associated text message of the error. The default setting is not to display these error messages.

File Interface

The File interface functions provide a mechanism to cue and play back audio, video and combined streams. Seamless back-to-back playback of multiple MPEG files playback is provided with the *File_Next()* and *File_Switch()* functions. Use of the File interface does not require any knowledge of the *Demux* or *RateBuffer* interfaces. In order to understand how to play two files seamlessly back-to-back, first read how to open and play a single file. Of the three interfaces, the *File* interface is the simplest to use for the functionality provided. In its simplest form, a typical program would simply call the *File_Open()* functions, then the *File_First()* function followed by the *File_Command(CStradisDecoder::FILE_PLAY)* function to start the playback.

The *Demux* Interface

The *Demux* interface is used when the application already has the stream data in memory in a transport stream, program stream or system stream format. The *Demux* interface is used to separate the audio and video PES or Packet data from the stream and place the data into the video and audio rate buffers. For convenience, the *Demux* interface will also process a single stream of audio or video PES or Elementary Stream data. If separate audio and video PES or Elementary Streams are in memory, the appropriate interface to use is the *RateBuffer* interface.

Before using the *Demux* interface, the *Decoder_Init()* and then the *Demux_Init()* functions must be called in that order. Once the initialization has taken place, the rate buffers can be pre-loaded by calling the *Demux_Stream()* function. Once sufficient data has been loaded into the rate buffers, the *Demux_StartPlay()* function can be called to start playing the stream. The *RateBuffer* interface can be used to determine how much data is in the rate buffers. When the end of the stream is reached (no more data to send to the *Demux_Stream()*), the *Demux_SetEndOfStream()* function must be called. The *Demux_SetEndOfStream()* function

cleans up the de-multiplexing process and assures that the decoder hardware receives all of the de-multiplexed data.

The *Demux* interface has no buffers associated with it. It takes data from the buffer passed to the *Demux_Stream()* function, separates the audio and video data and places that data directly in the rate buffers through *RateBuffer* interface.

RateBuffer Interface

The *RateBuffer* interface is the lowest level interface and provides direct access to the PC-based rate buffers. This interface can be used if the application has a separate audio and video stream in the Elementary, PES or Packet format. If the data is in the PES or Packet format, the hardware audio/video synchronization modes can be used. To use the *RateBuffer* interface, the application must first call the *Decoder_Init()* function followed by a call to the *RateBuffer_Init()* function. Once the interface is initialized, the rate buffers can be pre-loaded by calling the *RateBuffer_AddToVideoBuffer()* and *RateBuffer_AddToAudioBuffer()* functions. The *RateBuffer_StartPlay()* function can be used to start the decoder playing the data in the rate buffers.

Master Clock System

The decoder's master clock is a 27 MHz VCXO that can be controlled one of two ways. The VCXO can be set to follow a reference video signal (GenLock) or it can be controlled locally by the software with a pulse width modulator (PWM). In the GenLock mode, an input composite video signal is connected the decoder card. The decoder card watches the sync edges of the signal and locks the output vertical and horizontal sync edges to the reference signal. The relationship of the horizontal sync edges of the reference and output signals can be shifted in 37ns increments.

If the GenLock mode is not set, the VCXO is controlled by software with a Pulse Width Modulator (PWM). Larger values of the PWM value increase the VCXO frequency. The VCXO can be pulled by at least 100 PPM high or low of the nominal 27 MHz frequency. This allows the board to recover the clock of the original encoded signal in a transmission environment.

In the PWM mode, the API can provide a software mechanism to aid in the implementation of a clock recovery system. Clock recovery is required for MPEG transmission systems that require low delay and continuous feeds. A clock recovery system attempts to reproduce the sample clock used to sample the original video and audio. Unless clock recovery is implemented, the difference between the original clock and the decoder's clock will cause the decoder's rate-buffers to eventually underflow or overflow depending on whether the original clock is slower or faster than the decoder's clock.

Clock recovery can be ignored for relatively short transmissions. In the case of short transmissions, the application would set up a large enough rate buffer and allow it to fill half-way before starting the decoding process. If the decoder's clock was faster than the original clock, the rate-buffer would have enough data buffered so that by the end of the transmission the rate-buffer would not underflow. Likewise, if the decoder's clock was slower than the original clock, the rate-buffer could accept enough data that it would not overflow before the transmission completed. The size of the rate-buffer required to handle such a transmission depends on the length of transmission as well as the difference in the original and decoder clocks.

To implement clock recovery, the *m_bClockRecover* member variable of the *CStradisDecoderSettings* class must set when calling the function *Decoder_Init()* or *File_First()*, The API will call the virtual function *Demux_ClockRecover()* every time the *Demux_Stream()* function runs across a PCR in the stream. The *Demux_ClockRecovery()* function is called with two *double* parameters. The first parameter is the PCR value found in the stream. The second parameter is the current system time clock (STC). Both values are measured in seconds. The STC is a hardware clock reference in the decoder that is used to determine when frames of video and audio are presented to display and audio D/A system. When the STC is equal to the PTS of the frame, then the frame is displayed.

The base *Demux_ClockRecover()* function is a null function. The application is expected to overload the base function with its own function for clock recovery. The application's *Demux_ClockRecover()* function can then look at the difference between the PCR and STC and use some algorithm to vary the VCXO frequency to track the original clock. The API does implement an algorithm to do this. The algorithm used to recover the clock will depend on the transmission system used. The specification and implementation of such an algorithm is therefore beyond the scope of this API and its implementation is left to the application. However, the API provides all of the functions to implement such a clock recovery algorithm.

On Screen Display (OSD) Interface

The OSD function displays one or more bitmaps on top of a decoded video image. The size of a bitmap can be from 8x2 to 720x480 NTSC or 8x2 to 720x576 PAL and is not affected by the video bitstream parameters. The bitmaps are contained in regions that contain a header that specifies the coordinates of the region, size of the region, address of the next region, color resolutions, color table and other controls. The OSD function has the following features:

- Multiple rectangular regions of bitmaps linked by addresses
- Each line pair can be a region and up to one region per horizontal line pair
- 16 or 4 colors per pixel or pixel pair by region
- Each region has its own color table
- 16 levels of blending between video and OSD
- 16 levels of shading of video
- Easy removal of OSD region from screen
- Animation Support

The API provides two methods for building regions in the decoder's OSD memory. The first method provides

functions for building OSD regions from data resident in the host's memory space. This includes functions for creating YCrCb color tables from RGB values and setting the various OSD modes.

The other method of building regions in the OSD memory reads a .bmp file from disk and places the image in OSD memory as a region. The .bmp file must be a 2, 4 or 16 color file and should have a width that is devisable by 8 (16 color) or 16 (2 and 4 color) and height that is devisable by 2. If the width or height is not divisible by the appropriate amount, the image will be cropped on the right and/or top/bottom (depending on the .bmp file) to fit into the appropriately divisible size.

The API also supports the animation feature of the OSD.

Closed Caption

The API supports Closed Caption as two separate interfaces. One interface captures Closed Caption from the MPEG stream in real-time and the output produces the VBI waveform from Closed Caption data sent to the output interface. The Closed Caption interface can be set up to direct the captured Closed Caption information extracted from the MPEG stream to the output interface.

The Closed Caption capture interface automatically recognizes ATSC and several proprietary formats. However, there are two ways to incorporate Closed Caption data in to the MPEG stream. One way is to attach each pair of Closed Caption bytes to the frame from which they were originally decoded. This causes the Closed Caption bytes to arrive at the decoder out of sequence (same as video frames). In this mode, the decoder must re-order the bytes to get them back in time sequence.

The other way of encoding the bytes is to place them in the stream sequentially without regard to the actual frame with which they were originally associated. In this mode, the decoder simply passes the Closed Caption data as it is received and no re-ordering is necessary.

Still Picture Interface

The Still Picture Interface provides a means to place bitmaps on the video output. This interface is separate from the MPEG decoding interface in that the decoder must be reset (by calling *Decoder_Init()*) between Still Picture mode and MPEG playback mode. The call to *Decoder_Init()* is automatically performed by the *Decoder_PrepareVideoMemory()* (see below) and *File_First()* (see Section 6 File Interface) functions. There are two levels to the Still Picture Interface. At the highest level, the application can make one function call to display one of three predefined patterns or a bitmap file from disk. The predefined patterns are 100% color bars, 75% color bars and grayscale ramp.

The Still Picture Interface can also be used to place bitmaps in the host's memory into the decoder's video memory for display on the video output. The video memory is first prepared for the bitmap by calling the *Decoder_PrepareVideoMemory()* function. After the *Decoder_PrepareVideoMemory()* returns, the video display memory is initialized to black. In order to select the decoder's video output, the *DvBus_Set()* function must be called with the parameter equal to *DECODER*. The address of the luminance display memory can be obtained from the *IBM_GetLuminanceAddress()* function (See Miscellaneous Functions). The address of the chroma display memory can be obtained from the *IBM_GetChrominanceAddress()* function. The luminance data can be built in the host memory organized as Y0, Y1, Y2, Y3, Y4, Y5 … and the chrominance data is organized in the hosts memory as Cb_0, Cr_0, Cb_1, Cr_1 … . The *IBM_DramWrite()* function can then be used to transfer the bitmap image into the video memory.

Subsystem Settings Functions

These functions are used to setup and interrogate the various subsystems including GenLock, analog audio and the Digital Video Encoder.

Configuration ROM

The decoder contains a ROM that contains configuration information. Functions are provided to access this information.

Miscellaneous Functions

CSpciDmaMem Class

The purpose of the *CSpciDmaMem* class is to provide an easy-to-use mechanism to allocate and manage memory needed for DMA transfers. The class will allocate physically locked memory, and provide a logical address for access.

The Synchronization Interface (*CStradisSynchronize*)

The synchronization interface (*CStradisSynchornize* class) is used to synchronize one or more groups of decoders for use with the *File* interface. The *CStradisSynchronize* class provides for synchronous starting and stopping of each decoder in the group. If there is more than one group, each group must contain a mutually exclusive set of decoders with respect to the other groups. After the synchronization class is created, it must be initialize (by calling the *Init()* function) before any other functions within the class are called.

Note that in order for the synchronization class to work on a frame-accurate basis, all cards in a group must be GenLocked together. This may be done by feeding all cards from a common external sync generator, or by using one card to provide sync to the other cards in the system. In the first case, all decoders should have their GenLock enabled. In the second case, the master decoder should *not* have GenLock enabled and the slave decoder(s) should.

Stradis™
3170 Sugarplum Rd., NE
Atlanta, GA 30345-1553 USA
1-770-621-0625
fax 1-770-625-0310
www.stradis.com