IN 98

Interactive Media Systems

CD-I

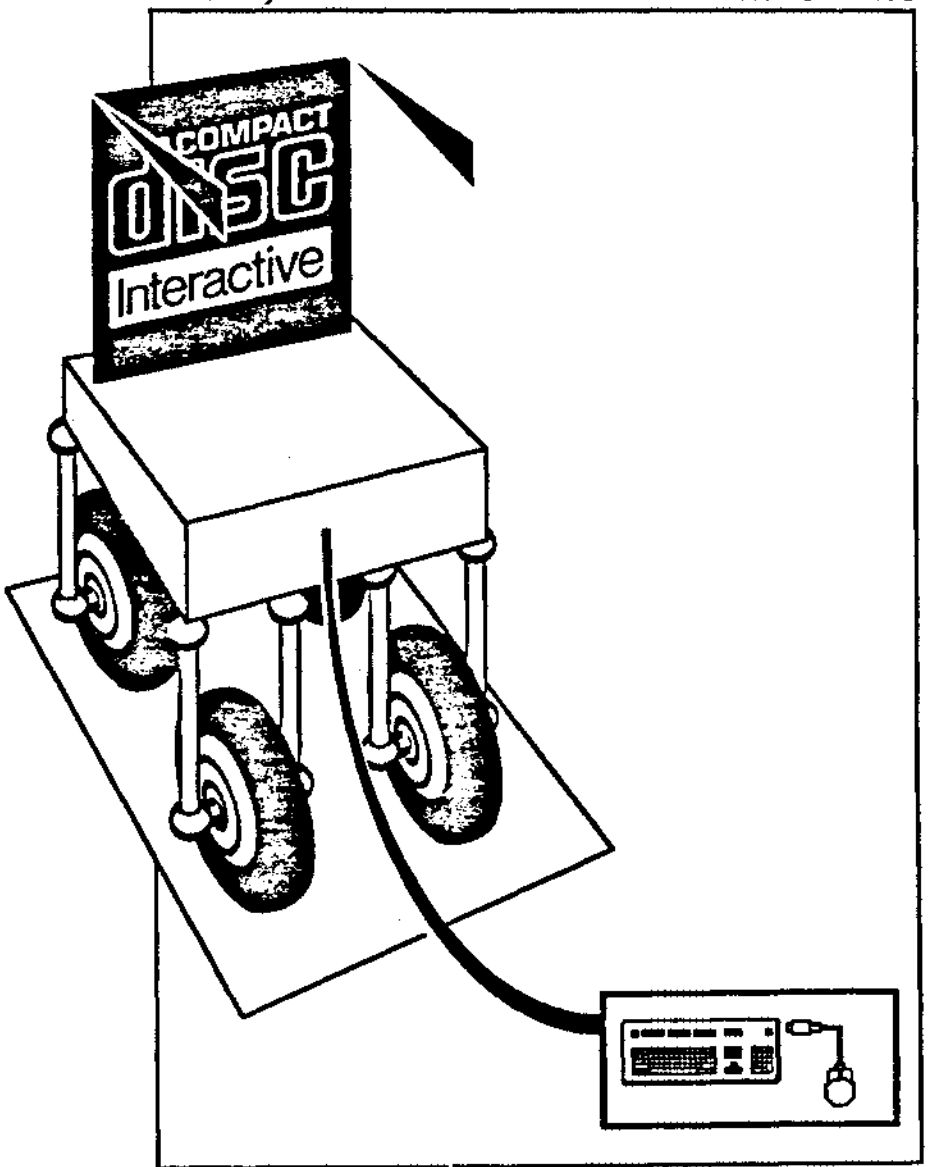



TSA APPLICATION NOTES NR. TSA-008 May 05 1993

Intro to programming the FMV System

This paper reveals the basic knowledge required to successfully implement simple FMV features such as play, pause, slow motion and scan. The build-time tools, run-time algorithms and data structures required to support these features are covered. Code examples are used to illustrate the techniques described.

Written by *Ken Ellinwood*



nr. of pages: 26

PHILIPS

Introduction

The purpose of this document is to help the application programmer get started in understanding the process of playing FMV sequences in CD-I. Some basic CD-I knowledge on the part of the reader is assumed, especially in the area of real-time plays with CDFM and OS9 signal handling.

This document is broken into two parts. In the first part, the relevant tools that must be used to create FMV sequences within a disc image will be discussed. In the second section, the focus turns to the implementation of simple linear play features in run-time application code.

Part I - Tools for FMV Development

In general, the process of preparing assets for full-motion playback in CD-I requires the following steps:

1. Grabbing
2. Encoding
3. Multiplexing
4. RTF generation / Disc building.

The grabbing and encoding steps are not applicable to this discussion and are therefore skipped.

Multiplexing

Audio and video are typically encoded by different encoders and therefore are delivered as independent assets. Multiplexing is the step where the elementary encoded audio and video streams are merged into a file containing two interleaved "system" streams, one for audio and one for video. The resulting streams each contain the ISO 11172 system layer - information that is necessary to support the synchronized playback of audio and video.

Multiplexing is performed with Pink, the multiplexer. Pink takes only one argument, the name of a script file that directs its operation. Here is an example of a simple script:

```
CDI  "mystream.mpg" pattern 0xFFFF "mystream.dlt" from
       video "../video/mystream.mpv" stream 0 start 0
       audio "../audio/mystream1.mpa" stream 0 start 0
       audio "../audio/mystream2.mpa" stream 1 start 0
```

This script will multiplex the elementary video stream and two elementary audio streams into a file named mystream.mpg. The pattern construct is used here to indicate that all available sectors may be used. Later on, the outputfile mystream.mpg will be used as input during creation of the real-time file for the disc image. The file mystream.dlt is called a delta file - it is created by the multiplexer and contains information used by the disc builder to allocate sectors for the multiplexed streams. The stream numbers and starting presentation time stamps (synchronization information) for each elementary stream are also given in the script.

If encoding was performed by the Philips encoder, the following modification can be made to the script to support the propagation of entry point information through the encoding and disc building process:

```
CDI  "mystream.mpg" pattern 0xFFFF "mystream.dlt"
       entrypoints "mystream.pel" from

       video "../video/mystream.mpv" stream 0 start 0
             entrypoints "../video/mystream.eel"
       audio "../audio/mystream.mpa" stream 0 start 0
```

This script directs Pink to read "mystream.eel" (generated by the encoder) and write the file "mystream.pel" (to be read by Master, the RTF/disc builder).

Entry Points and Scan Tables

Let's take a moment to discuss entry point data and its use by a CD-I application. Entry point data is used by the application at run-time to start playback at places other than the beginning of the video sequence. Due to the nature of the MPEG encoding and decoding algorithms, access to the sequence cannot begin from arbitrary locations - decoding must begin at an entry point. The location of entry points must be determined prior to encoding so that the encoder can be directed to encode the entry points at the proper locations. If the Philips encoding system is being used (Philips encoder, Pink, and Master), then the encoding system produces information that the application can use at run-time to access the entry points (assuming that it is told to, of course). Otherwise, the required information can be determined by parsing the output of the multiplexer.

The entry point information needed by the application at run-time corresponds to the location of entry points within the stream. In the case of video, the stream starts with a sequence header and may be repeated at various location throughout the stream. Each entry point corresponds to a sequence header within the video stream. The entry point information for audio simply results in a sector number due to the frequency at which audio frames occur within sectors.

In the general case, the information needed to access an entry point consists of the following information:

- Channel number and sector within the RTF.
- For each stream in the multiplex the following information is also needed:

    i. stream type (audio or video)
    ii. stream number
    iii. byte offsetto the entry point in the multiplexed stream of this type (audio or video).

Each entry point may also have a name associated with it for symbolic access. Other information such as the width and height of the encoded video may be determined at run-time through CD-RTOS calls.

This is quite a lot of information. In practice, many of these parameters may be assumed. For example, let's assume that there is only one video and one audio stream to play back and that the channel number and stream numbers are always zero. It can also be assumed that the audio and video offsets are zero simply because the MPEG drivers will perform correctly in almost all circumstances when the zero is passed as the video and audio offsets. In this case, all that is required is a sector number for each entry point.

A scan table is required to support playback in scan mode. A scan table contains data in the same format as an entry point table - in fact, it is simply a super set of data that also contains entry point information. Once the play has been started at an entry point, the application can enter scan mode and seek to other addressable units within the stream, such as a group of pictures (GOP). In this case, the GOP is not required to have been preceded by a sequence header. Because the application can also seek to an entry point during scan mode, the two tables may be one and the same.

RTF Generation

Master can be used to generate MPEG within real-time files. There are three options when generating MPEG within real-time files.

- No entry point data generation.
- Read and write entry point data from/to external files.
- Read entry point data and place the output in data sectors prior to the MPEG sectors in the real-time file.

Here is a simple script that creates a real-time file without processing entry point data:

```
real_time mpeg in channel 0                    ! Channel 0
      from "../pink/mystream.mpg"              ! Multiplexed input
      delta_file "../pink/mystream.dlt"        ! Delta file input
      ep_list off                              ! No EP input/output
      at 0                                     ! Sector 0
```

The MPEG streams will be placed in the real-time file such that the video is in video sectors and the audio is in audio sectors. FMV coding types are defined for

MPEG audio and video in the Green Book extension, Chapter IX of the CD-I Full Functional Specification.

For more information regarding Master's syntax and the format of the entry point output file, refer to chapter 2 of the Disc Building document available from Optimage.

For those developers that are dependent on DBL and/or CDL-BD, there is an alternative method for creating MPEG real-time files that is compatible with these tools. Contact Developer Services at PIMA for more information.

The process of building a disc has not changed with the advent of FMV. For this reason, disc building is left as an exercise for the reader.

Part II - Writing Code for FMV Playback

In this section many code examples are given. Most of the #define values that appear in upper case in the code examples are not present in mv.h and ma.h, the header files delivered for the FMV system. The file fmv.h, where these symbols are defined, has been attached to the end of this document for reference.

One final note, the examples that follow are for instructional purposes only. The author has derived these segments from actual working code but does not guarantee that they are free from mistakes or omissions.

Devices and Descriptors

The MPEG decoders are accessed as devices through CD-RTOS. Before opening each device to obtain a path number, the application must first determine the name of the device from the CSD in a manner similar to that used for the base case video device. The device numbers used for the MPEG video and audio decoders are 90 and 91, respectively. It may be desirable to update your local copy of csd.h to include the following two lines:

```
#define DT_MPEGV90      /* Device number of MPEG Video decoder */
#define DT_MPEGA91      /* Device number of MPEG Audio decoder */
```

Once the path numbers for the audio and video decoder have been obtained, the application must create audio and video descriptors. Each of the descriptors come in two types, CD and HOST. These types determine if the play is to take place from data delivered by the CD or from pre-loaded memory. For video, the descriptor can also be created to play MPEG Still Pictures. In all cases, the application must make the decision regarding the type of the descriptor before the descriptor is created. To create a descriptor, use mv_create() and ma_create(). The examples given in this document will all assume that the play is from the CD:

```
mv_desc = mv_create( mv_path, MV_TYPE_CD);
ma_desc = ma_create( ma_path, MA_TYPE_CD);
```

Signals Generated by the MPEG Decoders

The application may request that the decoders signal the application on the occurrence of certain events during the playback of MPEG. The mv_trigger() and ma_trigger() calls are used to inform the decoders of the events that the application is interested in. Both functions take 16 bit parameters - the upper 5 bits act as the signal base and the lower 11 bits act as a mask by which the application can select events. For example, here is the format of the parameter to mv_trigger():

Signal Base	•	NIS	BUF	EOS	ECI	CNP	LPD	SOS	GOP	PIC	DER
Bit 15.....11	10	9	8	7	6	5	4	3	2	1	0

Refer to the Chapter IX of the CD-I Full Functional Specification for more details regarding the actual meaning of the bits listed here.

The implication is that the decoders do not generate unique signals based on each event. If two selected events occur simultaneously, then the decoder generates a signal whose value is that of the signal base combined with the bits corresponding to the two events. This may require a modification to the application's existing signal allocation and handing logic. If the application previously intended to receive unique signal numbers, it may continue to do so as long as the non-MPEG signals have values less than 2048. In this case the application's signal handler can check the upper bits of the signal for the MPEG signal bases to determine how it should be handled. Other applications that already allocate signal classes (a la Balboa) should switch to classes that are multiples of 2048. Balboa users must switch to version 1.3.2 (or higher).

PCL Handling by the MPEG Drivers

The mechanism by which data gets from the disc to the decoder during MPEG playback is through PCL transfer buffers. For each decoder, the application is required to provide a chain of one sector PCL buffers to which CDFM will deliver data and from which the MPEG driver will receive it. For simple plays that do not involve slow-motion mode, two sets of two circularly linked PCLs will suffice. With one exception, the interface to CDFM remains the same as it always has - the application sets a bit in the PCB_Chan mask of the PCB as well as the corresponding PCL pointers in the audio and video channel index lists in order to enable data reception in the channel containing the MPEG sequence. The exception is that the MPEG drivers will maintain the PCLs during the play. The drivers reset the PCL_Cnt and PCL_Ctrl fields in the PCL after it empties each buffer. As expected, if the PCL_Sig field is set, CDFM will signal the application when the buffer is filled. The MPEG driver will also signal the application with the same signal when thebuffer is emptied. Under simple playback circumstances, however, the application does not need to set the PCL_Sig fields in the PCLs used for MPEG

Starting the Playback of FMV

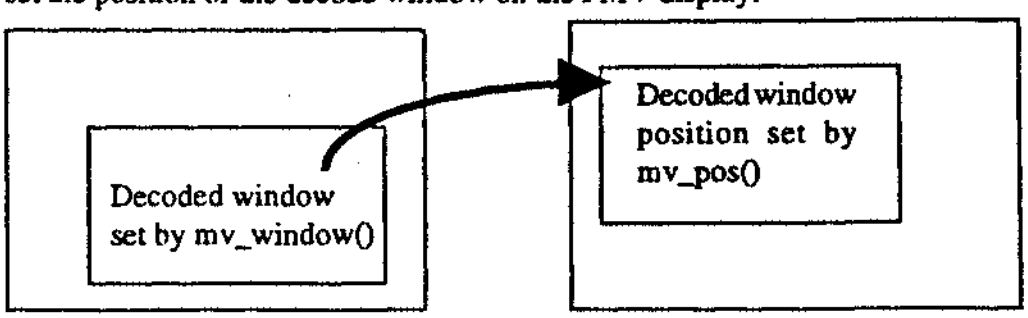
Now that the finer points of signaling and PCL handling have been covered, it is time to move on to starting an actual play of an MPEG sequence. In an attempt to keep the examples simple, many parameters of the playback will be assumed. Let's assume for the time being that the sequence to be played back is encoded at 30 frames per second, is 352x240 pixels wide, that the video and audio are in

stream number zero, and that it is to be displayed in NTSC. Also, keep in mind that the functions being called return proper error values when an error occurs. Error handling logic has been omitted from these examples for the sake of readability. First, let's set up the FMV window for display:

```
/* Set the display origin for FMV */
mv_org( mv_path, 0, 0, 0);

/* Preset the encoded picture size and rate */
mv_imgsize( mv_path, mv_desc, 704, 480, 30);

/* Set the size and position of the decoded window relative to the encoded pictures. */
mv_window( mv_path, mv_desc, 0, 0, 704, 480, 0);

/* Center the decoded window in the display */
mv_pos( mv_path, mv_desc, 32, 0, 0);
```

If the dimensions of the encoded pictures are not known by the application prior to starting the play, it can wait until the first picture signal arrives and query the motion video descriptor with mv_info() in order to determine the size. Under these circumstances mv_imgsize() will not need to be called since the decoder will have established the picture size from the video sequence header. The application simply needs to set the window size and position with mv_pos() and mv_window().

There are several important things to note from the example above. Prior to decoding a video sequence, the decoder does not assume the width and height of the encoded pictures. Because mv_window() clips the display window to the encoded picture size, if the encoded picture size has not been preset to something other than zero, the call to mv_window() will result in a decode window size of zero. It is also important to know the difference between mv_window() and mv_pos(). The size of the encoded pictures is determined when the video is grabbed and encoded. The function mv_window() sets the size and position of a decode window on the real-estate of the encoded image. The function mv_pos() set the position of the decode window on the FMV display.



The application must also set the audio and video streams to be decoded:

```
/* Set the audio attenuation and stream number */
ma_cntrl( ma_path, ma_desc, MA_ATTEN_NORM, 0);

/* Set the video stream number and width, height, pic rate */
mv_selstrm( mv_path, mv_desc, 0, 704, 480, 30;
```

The function ma_cntrl() is used to set the audio attenuation in a manner similar to sc_atten().

Typically, the application should wait until the first picture is displayed to enable the FMV display. To do this the PIC signal should be enabled:

```
/* Set upper 5 bits to sig base determined by application */
mv_trigs = mv_signal_base;

/* Add the PIC bit and call mv_trigger() to enable */
 mv_trigs |= MV_TRIG_PIC;
mv_trigger( mv_path, mv_trigs);
```

It is left to the reader to initialize a PCB and a set of PCL buffer chains (one for audio and one for video). Refer back to the section titled "PCL Handling by the MPEG Drivers" for more information.

Once the PCB and PCL-buffers are set, it is time to start the play:

```
mv_cdplay(  mv_path,              /* MPEG Video path */
            mv_desc,              /* MPEG Video descriptor ID */
            MV_SPEED_NORMAL,      /* Playback at normal speed */
            0,                    /* Video entry point offset */
            mv_pcls,              /* PCL list for MPEG Video */
            &mv_stat,             /* Status block for video play*/
            MX_WAIT_SYNC, /* Wait to sync with audio play */
             0);                  /* Synchronization offset */

ma_cdplay(  ma_path,              /* MPEG Audio path */
            ma_desc,              /* MPEG Audio descriptor ID */
            0,                    /* Audio entry point offset */
            ma_pcls,              /* PCL list for MPEG Audio */
            &ma_stat,             /* Status block for audio play*/
            mv_path,              /* Sync playback to video */
            0);                   /* Synchronization offset */
```

Note that the video path is passed as the synchronization mode parameter to the ma_cdplay() call. This will enable synchronized playback of the audio and video based on the system clock reference and presentation time stamps found in the system layer of the two streams. The synchronization offset is zero because it is assumed that the beginning of the audio and the first picture in the video are to be presented together.

The drivers are now ready to receive data. The only thing that remains is to start the disc play:

```
/*    Seek to the beginning of the real-time file that contains the MPEG streams and start
      the disc play. */

lseek( rt_file_path, 0, 0);
ss_play( rt_file_path, &fmv_pcb);
```

Now that the play is started, the application will quickly be signaled as to the receipt of the first picture. At this time the application's signal handler should enable the display of the FMV plane. In the following example, a simple implementation of how to handle the MPEG signal format is also shown:

```
sig_handler( signal)
 int signal;
{

        if ((signal & SIGNAL_BASE_MASK) == mv_signal_base)
        {
                /* Handle MPEG video signals */

                /* Is the PIC bit set? */
                if (signal & MV_TRIG_PIC)
                {
                        /*      Enable the FMV plane and set the base case video planes to
                                transparent. */
                        mv_show( mv_path);
                        dc_wrfi( vpath, fct_a, FCT_TCI,
                                cp_tci( MIX_OFF, TR_ON, TR_ON));

                        /* Disable further reception of the PIC signal */
                        mv_trig &= -MV_TRIG_PIC;
                        mv_trigger( mv_path, mv_trig);
                }
        } else if ((signal & SIGNAL_BASE_MASK) == ma_signal_base)
        {
                /* Handle MPEG Audio signals */

                ...
        } else
        {
                /* Handle other signals */

                ...
        }
}
```

Pause and Continue

The FMV play may be easily paused or continued at any time:

```
/* Pause the play */
mv_pause( mv_path);
ss_pause( rt_file_path);


/* Continue the play */
mv_continue( mv_path);
ss_cont( rt_file_path);
```

Because the playback of audio and video are synchronized, the call to mv_pause() will also pause the playback of audio. When mv_continue() is called, the playback of audio will resume as well.

Handling the End of Play

The FMV playback may be aborted at any time. Here is an example of the calls that must be made:

```
mv_abort( mv_path);
ma_abort( ma_path);
ss_abort( rt_file_path);
```

Both mv_abort() and ma_abort() must be called even if the playback is synchronized because if one is aborted without the other, the non-aborted decoder will continue to decode in non-synchronized mode.

Both decoders must be aborted before another play may be started, even if the play finishes normally. A normal end of play condition can be determined when the EOI event occurs just after the Last Picture Displayed (LPD) event is received.

Slow Motion Playback

In slow motion mode, the drivers decode and display pictures at a slower rate with the audio muted. The drivers support seven slow motion speeds - 1/2 through 1/8 normal speed. This mode requires a higher level of interaction between the MPEG drivers and CDFM, but because there is no communication between these

two, it is left to the application to act as an intermediary. While in slow motion mode, the MPEG drivers empty the PCL buffers at a slower rate than normal. To ensure that the PCL-buffers do not overflow or under flow in this mode, the application must pause and continue the disc play at the correct times to slow down the rate of data delivery from CDFM. However, because it may take up to a second for data delivery to resume after an ss_cont() call, there must be at least one second's worth of data to decode in the PCL buffers when the call is made. This means that to support this mode the application must use a larger PCL-buffer chain than compared to a normal speed play.

Two possible implementations of slow motion will be discussed here. The first is more memory intensive but requires less CPU throughput. The second requires less memory but a greater amount of CPU.

Method #1 - More Memory, Less CPU

The first method requires a PCL-buffer chain twice as large as that required to hold a second's worth of data to decode in slow motion. One can think of it as two halves, one of which is being decoded by the MPEG video driver while the other is being filled by CDFM. The worst case memory size for this technique is when the playback speed is highest because the decoder empties PCLs faster than at slower speeds and therefore a greater number of PCL-buffers are required to cover the latency in ss_cont(). At half speed playback, the decoder will empty approximately 37.5 sectors per second (worst case), so the actual number of sectors required is 75. Here is a general description of the processing required for slow-motion mode:

- Allocate a signal number for use in the PCLs during slow motion playback.
- Allocate and initialize 75 PCL-buffers for the video decoder. Two PCL-buffers for the audio decoder will still suffice.
- When entering slow motion mode, set the PCL_Sig field of the PCL at the head of the list (the one pointed to by the channel index list) to the signal value allocated above. Also set the PCL_Sig field of the PCL halfway down the list to the same value. Set these in the video PCLs only, not the audio PCLs. The call to mv_chspeed() is as follows:

```
mv_chspeed(     mv_path,                    /* MPEG Video path */
                MV_SPEED_SLOW(speed),       /* Speed parameter */
                 0,                         /* Offset parameter */
                NULL);                      /* New PCL chain */
```

The speed parameter is a value in the range 2 through 8, denoting speeds of 1/2 through 1/8. The offset parameter is zero for the transition from normal speed to slow motion, and if the 75 PCL-buffers are already in use during normal speed playback, the NULL parameter for the new PCL chain indicates that the PCLs currently in use are also to be used for slow motion.

- Immediately, and also every time the slow motion PCL signal is raised, check the to see if the PCL halfway down the list (the one that also has the PCL_Sig field set) is full. If it is full, then pause the disc. If it is empty, then make sure the disc is playing, calling ss_cont() when necessary.

This method will ensure that the buffers are either more than half full with the disc paused, or that the buffers may be less than half full with the disc continued. It will guarantee that the buffer will not overflow resulting in a read error from CDFM or an underflow error from the MPEG decoder.

Method #2, Less Memory - More CPU

The second method of implementing slow motion requires less memory but more CPU. This method is implemented in the Balboa play manager to support FMV slow motion mode. The technique still guarantees that there will be one second's worth of data in the PCL buffers when the ss_cont() call is made. Here is how it works:

- Allocate a signal for use in the PCLs during slow motion mode.
- Allocate and initialize 42 PCL-buffers for the video decoder. Two PCL-buffers for the audio decoder will still suffice.
- When entering slow motion mode, set the PCL_Sig field in all PCLs in the video chain. Refer to the previous method for an example of the mv_chspeed() call.

- Immediately, and whenever the slow motion PCL signal is raised, check the four PCLs at the head of the list. If any of the four PCLs are full, pause the disc. If they are all empty, ensure that the disc is playing, calling ss_cont() when necessary.

This method requires more CPU simply because it generates more signals than the other method. Signals are generated for each PCL by CDFM and the MPEG Video driver at an average rate of 75 per second, although the rate is sporadic. The frequency of the ss_cont()/ss_pause() pairs is not as high as one would imagine because the latency in ss_cont() allows the buffers to empty quite a bit before data delivery resumes.

Using either method, the video may be paused or continued by simply calling mv_pause() and mv_continue(). When the video is paused and continued in slow motion modethe disc is handled automatically by the slow motion algorithm that keeps the buffers from over- and underflowing. When resuming to normal speed play, clear any PCL_Sig fields that are set. Keep in mind that the decoder will not resume decoding in normal speed after slow motion until it senses that new sectors are being delivered by CDFM. Therefore, if the disc is paused, the application must also restart the disc when resuming to normal speed.

Single Step Mode

Single step mode allows the application to control the advance of pictures in the video sequence one picture at a time. For this mode, the same PCL buffering scheme used for slow motion can be used to ensure that there is always enough data in the PCLs in order to decode and display the next picture when the application requests it. The application changes to single step mode by changing speed to single step:

```
/* Change to single step. Offset and PCL parameters are zero */
mv_chspeed( mv_path, MV_SPEED_SINGLE_STEP, 0, NULL);
```

Once in single step mode, the application requests that the next picture be displayed with the mv_cdnext() call:

```
/*    Request the next picture be displayed. Offset, PCL and still page parameters are
      zero. */
mv_cdnext( mv_path, 0, NULL, 0);
```

When using mv_cdnext(), the application must enable the PIC event and wait for the PIC signal to arrive before calling mv_cdnext() again.

Scan Mode Playback

Scan mode allows the application to quickly scan through a series of images within a sequence. This mode is provided by CD-RTOS so that the application can implement fast forward and fast rewind functions. Scan mode looks somewhat like CAV laser disc fast forward - still pictures from the sequence appear rapidly on the display.

Scan mode requires that the application use a scan table. Refer back to the section titled "Entry points and Scan Tables" for more information. For the time being, it will be assumed that the scan table is simply an array of sector numbers corresponding to the entry points and scan points for the sequence. For example:

```
int scan_table[] = { 0, 28, 67, 98, 133, 171, 207, 243, 278, ... };
```

This array is simply the sector numbers containing the entry points (sequence headers) and GOPs for our sequence. This information can be obtained from the output of Master or by parsing the output of Pink. Refer to the attached program, findgops.c, for an example of how to parse the output of Pink.

From normal speed play forward mode, the application enters scan mode by changing speed to scan speed, aborting the disc play, and enabling the PIC signal. The application also need to know where to start scanning from within the scan table. This can be done by finding the current sector offset using _gs_pos() and then searching for the closest sector number in the scan table:

```
mv_chspeed( mv_path, MV_SPEED_SCAN, 0, NULL);
ss_abort( rt_file_path);

mv_trigs |= MV_TRIG_PIC;
mv_trigger( mv_path, mv_trigs);

current_sector = _gs_pos( rt_file_path) / 2048;

/*      This function searches the scan table for the closest sector and returns an index into
        the scan table. */
scan_index = find_scan_index( current_sector);
```

Upon reception of the first signal after entering scan mode (usually the PCB_Sig resulting from the ss_abort()) and every PIC signal thereafter, the application must abort the disc play (if called by a PIC event), clear the PCL buffers, call mv_cdnext() to ready the driver for the arrival of the next picture to decode and display, seek to another scan point, and restart the disc play. The choice of how many scan points will be skipped each time will determine the rate at which the application moves through the sequence in this mode. Here is an example of the logic described above:

```
void fmv_scan_sig_handler( signal)
int signal;
{
        PCL *pcl;

        /* If the signal is not the result of a PIC event, abort the disc play. */
        if (signal != E_ABORT) ss_abort( rt_file_path);

        /* Empty the PCL buffers */
        pcl = fmv_pcb.PCB_Video[MPEG_VIDEO_CHANNEL];
        while (!pcl->PCL_Ctrl) pcl++;
        while  (pcl->PCL_Ctrl)
        {
                pcl->PCL_Ctrl = 0; pcl->PCL_Cnt = 0; pcl++;
        }

        /* Are we still in scan mode? */
         if (mode != SCAN_MODE)
        {
                /*      Resume normal speed play at the current scan point (or whatever is
                        required by the application design) */

                fmv_play( scan_index); return;
        }

        /*      Determine the next scan point to seek to - direction and distance determine
                forward or reverse (1, -1) and distance jumped to next scan point */
        scan_index += direction * distance;

        /* Call mv_cdnext() passing the pointer to the head of the PCL list */
        mv_cdnext( mv_path, 0, fmv_pcb.PCB_Video[ MPEG_VIDEO_CHANNEL], 0);

        /* Seek to the scan point */
        lseek( rt_file_path, scan_table[ scan_index] * 2048, 0);

        /* Reset the PCB_Sig field - must be reset after ss_abort() */
        fmv_pcb.PCB_Sig = fmv_pcb_sig;

        /* Start the disc play */
        ss_play( rt_file_path, &fmv_pcb);
} /
```

```
*****************************************************************************
*
* Filename:        fmv.h
* Purpose:         Constants used with CD-RTOS Full Motion Video & Audio
*
* Copyright 1993, Philips Interactive Media of America. All rights reserved. *
*****************************************************************************/
#ifndef _FMV_H
#define _FMV_H
#include <mv.h>

/* Speed constants for video */
#define MV_SPEED_SINGLE_STEP    0x7FFFFFFF
#define MV_SPEED_SCAN           0x80000000
#define MV_SPEED_SLOW(rate)     (rate)
#define MV_SPEED_NORMAL         0x00000000

/* Attenuation values for MPEG Audio */
#define MA_ATTEN_FULL           0x80808080 /* No audio - full attenuation */
#define MA_ATTEN_NORM           0x00800080/*  Normal - full volume */

/* Type values for ma_create() */
#define MA_TYPE_HOST            0x0000
#define MA_TYPE_CD              0x0001

/* Types for mv_create() */
#define MV_TYPE_HOST            0x0000
#define MV_TYPE_CD              0x0001
#define MV_TYPE_STILL(page)  ((page) << 12) |0x0800)

/* Bits in the signal mask used for ma_trigger() and mv_trigger() */
#define MA_TRIG_EOI   0x0001 /* End of ISO stream */
#define MA_TRIG_CSU   0x0002 /* Decoder changed to a new stream */
#define MA_TRIG_UPD   0x0004 /* Decoder updated the frame header */
#define MA_TRIG_UNF   0x0008 /* Decoder data underflow */
#define MA_TRIG_DEC   0x0010 /* Decoder started decoding */
#define MA_TRIG_ALL   0x001F /* All triggers */

#define MV_TRIG_DER   0x0001 /* Data Error */
#define MV_TRIG_PIC   0x0002 /* Picture Displayed */
#define MV_TRIG_GOP   0x0004 /* Group of Pictures */
#define MV_TRIG_SOS   0x0008 /* Start of Seq */
#define MV_TRIG_LPD   0x0010 /* Last Picture Displayed */
#define MV_TRIG_CNP   0x0020 /* Old PCL not used */
```

```
#defineMV_TRIG_EOI     0x0040 /* End of ISO strm */
#defineMV_TRIG_EOS     0x0080 /* End of Sequence */
#defineMV_TRIG_BUF     0x0100 /* Buffer underflow */
#defineMV_TRIG_NIS     0x0200 /* New Sequence Parms */
#defineMV_TRIG_ALL     0x03FF /* All triggers */

/* literals for the audio board */
#define MA_AUD_MODE          0x000000C0
#define MA_AUD_STEREO        0x00000000
#define MA_AUD_JOINT         0x00000040
#define MA_AUD_DUAL          0x00000080
#define MA_AUD_SINGLE        0x000000C0

/* literals for the video board */
#define MX_NO_SYNC    -1
#define MX_WAIT_SYNC -2

#endif _FMV_H


/
*****************************************************************************
*
* Filename:   findgops.c
* Project:
* Purpose:    Parsean ISO stream and determine the location of SOS or GOPs
*             and write a scan table to stdout.
*
* Functions:
*
*
 * Author:    Ken Ellinwood
* Date:       February 22, 1993
* Revisions:
* Tests:
* Dependencies:
* Notes:
*
*
* SCCS:       %W% %G%
*
* Copyright 1993,Philips Interactive Media of America, All rights reserved.
*
*****************************************************************************/
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

```
FILE *isofp;
FILE *dltfp;

unsigned char packData[4640];     /* Room to read in a few packs */

int sector;     /* Current sector number */
int isopos;     /* Current iso file position */
int psize;      /* Return value from fread() */
int *iptr;      /* Pointer to an integer */

#define PACK_CODE 0x000001BA

typedef enum
 {
        GOP,
        SOS
};


char sdelta;   /* Sector delta from delta file */

/*****************************************************************************
*
*Function:    usage()
* Purpose:
*
*****************************************************************************/
 usage()
{

        fprintf(stderr, "USAGE: findgops <multiplexed stream>\n");
        fprintf(stderr, "Generates a scan table for an ISO MPEG stream.\n");

        exit( 0);
}

*****************************************************************************
*
 * Function:   main() - findgops
* Purpose:
* Passed:
*
*****************************************************************************/
main( argc, argv)
        int argc;
```

```
        char *argv[];
{

        int i;
        char *rindex();
        int hours;
        int minutes;
        int seconds;
        int pictures;
        int mode = GOP;
        int epCount = 1;
        int packCode;
        int vPackCount;
        int pDataSize;
        int holdSectorCount;

        if (argc < 2) usage();

        isofp = fopen( argv[1], "r+");

        *rindex( argv[1], '.') = '\0';
        strcat( argv[1], ".dlt");

        dltfp = fopen( fot.filename[0], "r");

        fprintf(stdout, "#include <fmv.h>\n\nint scan_table[] = {\n\t0, \n");

        for (;;)
        {
                /* Seek to the beginning of the pack */
                if (fseek( isofp, isopos, 0) ==         -1) break;

                vPackCount = 0;
                pDataSize = 0;
                holdSectorCount = 0;

                /* Read two video packs into the buffer so that we can parse for start codes
                event when they are interrupted by pack codes */
                while  (vPackCount < 2)
                {
                        /* The first 4 bytes must be the pack start code */
                        fread(  &packCode, 1,4, isofp);
                        if (packCode != PACK_CODE)
                                exit (errmsg( "main: iso pack not found"));

                        /* Read the pack, assume its a video pack for now */
                        psize = fread( &packData[ vPackCount * 2320], 1, 2320, isofp);
                        if (psize != 2320 && psize != 2300) break;
```

```
        /* Determine if this pack contains audio */
        iptr = (int *)&packData[vPackCount * 2320 + 2300];
        if (*iptr == PACK_CODE || psize == 2300)
        {
                /* This is an audio pack, skip it */

                /* Get the sector delta from the delta file */
                fread( &sdelta, 1, 1, dltfp);

                if (!vPackCount) sector += sdelta;
                else holdSectorCount +=sdelta;

                isopos += 2304;

                if (fseek( isofp, isopos, 0) == -1)
                stop_parsing();
                continue;
        }

        if (!vPackCount)
        {
                /* Get the sector delta from the delta file */
                fread( &sdelta, 1, 1, dltfp);
                holdSectorCount += sdelta;

                isopos += 2324;
        }

        pDataSize += psize;

        vPackCount += 1;
}


if (pDataSize == 0) break;

/* Look for group start codes in this pack */
 for (i = 0; i < 2320;  i++)
{
        if (packData[i] == 0 && packData[i+1] == 0
                && packData[i+2] == 1) /* Start code */
        {
                if (packData[i+3] == 0xb3) /* Sequence start code */
                {
                        mode = SOS;
                        fprintf( stdout, "\t%6d, ", sector);
                }
```

```
                    if (packData[i+3] == 0xb8) /* Group start code */
                    {
                        /* Find the timecode */
                        hours = (packData[i+4] & 0x7c) >> 2;
                        minutes = ((packData[i+4] & 0x03) << 4) |
                            ((packData[i+5] & 0xf0) >> 4);
                        seconds = ((packData[i+5] & 0x07) << 3) |
                            ((packData[i+6] & 0xe0) >> 5);
                        pictures = ((packData[i+6] & 0x1f) << 1) |
                            ((packData[i+7] & 0x80) >> 7);

                        if (mode == SOS)
                        fprintf(stdout, "
                            /* ep: %4d, SOS @%2d:%02d:%02d.%02d */\n",
                            epCount++, hours, minutes, seconds, pictures);

                        else fprintf(stdout,"\t%6d,
                            /* ep: %4d, GOP @ %2d:%02d:%02d.%02d */
                            \n", sector, epCount++, hours, minutes, seconds,
                            pictures);

                            mode = GOP;
                        }
                    }
                }

                /* Go on to the next pack. */
                sector += holdSectorCount;

        }
        stop_parsing();
}

stop_parsing()
{

        fprintf( stdout, "};\n");
        exit( 0);
 }
```