Balboa Group

OptImage Interactive Services, L.P.

Balboa Video Users Guide

DRAFT #3

Bruce W. Englar
Balboa Product Development Manager

David Crowe
Balboa Technical Support

OptImage Balboa Group, 1501 50th Street, West Des Moines, Iowa 50265
Phone 515/225-7000 Fax 515/222-2022

Contents

1. Introduction

This note attempts to give some introduction to the Balboa video managers in a order which is a little more relevant to users than the current programmers guide. In the interests of clarity, the code examples here leave out error checking, status manager and any code optimizations such as register variables. They should not be taken as being the most efficient implementation of the functionality described here.

2. Basic Philosophy

The video section of Balboa is structured as a linked set of managers. These are structured as a core of managers which are highly inter-dependant and a wider set of managers which are optional components and in general only depend on the core.

Many of the managers are driven by an application building structures and then calling a high level function to implement what is described in those data structures.

3. The Managers

3.1 Video Screen

The video screen manager is the core of the Balboa video manager setup. Most of the other managers have connections to this in some form. It provides the link between what the application requires to be displayed and the video system of the CD-I player.

3.2 Video Environment

The video environment manager is a low level manager which connects together the video system, the cursor/hot spot system and the Balboa system state kernel. For applications using the main part of the video manager, this manager is internal and almost need never be seen. The reason it is a separate manager is so that applications can use the cursor/hot spot system and the system state kernel without having to use the main part of the video managers.

3.3 Picture

The picture manager is the main method by which applications define what should be displayed on the CD-I video hardware. Using the various picture manager calls, applications build up a list of PICTURE structures which the video screen manager translates onto what the video hardware is capable of displaying. Also included in the picture manager are various low level functions for displaying PICTUREs which are called by the video screen manager as part of the display process.

3.4 LCT

The LCT manager is normally an internal manager used by various of the other video managers to ensure coordination between them. Those video managers which need to use LCT space request it from the LCT manager which then locks out that space until the owning manager frees it. All of the video related managers apart from the text manager access this in some form. Applications should only ever need to access this manager if they are writing directly to LCTs and wish to be able to work cleanly with the rest of the video managers.

3.5 Clut

The clut manager provides an interface to the clut (palette) mechanism in the CD-I video hardware. The higher level functions in this manager provide a clut space tracking mechanism to help applications which are making advanced use of the clut organize and control what they are doing. This layer also includes support for the text manager where that uses anti-aliased fonts. Applications which use the entire clut in one operation can use the lower level functions as described in section 5.1 below.

3.6 Text

The Balboa text manager supports the use of runtime text by applications. Using the manager, applications define rectangular sections of PICTUREs into which text can be written. The structures which define these areas also include definitions of all the text formatting features for that rectangle. A larger description of the text manager is below.

3.7 Matte

The matte mechanism in CD-I gets very complex to use once applications get beyond a single shape on the screen. The Balboa matte manager provides an interface to this section of the hardware which makes it easy to do complex things with mattes. Due to the complexity of the hardware, this manager is quite slow and applications which need to do simple things with mattes are recommended to do it themselves at the CDRTOS level. For an example of how to do this, see section 7.2 below.

3.8 Video Synchronization

The video synchronization manager is a very low level package to handle video synchronization. Applications are unlikely to gain anything from using this manager directly and should normally use the video interrupt manager. The separation between the two is mostly for historical reasons.

3.9 Video Interrupt

The video interrupt manager is a wrapper round the video synchronisation manager to allow applications to define particular scan lines on which interrupts should be generated and to specify function(s) which should be called when those interrupts happen.

3.10 Video Effects

The video effects manager provides a library of video effects for applications. All the effects are asynchronous. Most of the effects cause a PICTURE currently displayed on the screen to be gradually replaced by a second PICTURE. Look in the manual for a description of the full range of effects.

4. Initialisation

4.1 Video Screen Manager

Before an application can access any of the features of the CD-I video system, it must open a path to the CDRTOS video device. The normal interface to this layer is via the Balboa function vs_init(). There are two parameters to this call, the first is one of the standard memory types as defined for the function bp_allocate(). The second defines the number of internal buffers created. These buffers are used by various functions within the video managers as temporary workspace to avoid repeated allocation and de-allocation of memory. These internal buffers are managed by the internal buffer manager, whose function names all start with the prefix buff_. Unless an application is intending to use these itself, a value of one should be adequate. A typical call to this function could look like :-

```
#define INTERNAL_BUFFER_COUNT 1

         vs_init(BP_MEM_DONTCARE,INTERNAL_BUFFER_COUNT);
```

Most of the other managers which need specific initialisation require to be attached to a video screen structure. The normal call to create one of these is vs_open(). The first parameter to this is a set of flags which defines various options to be used when creating the video screen. The two shown here cause a video screen to be created with two sets of LCTs ready for double buffering and also enable the cursor. Double buffering of LCTs is very strongly recommended. The second parameter defines the number of display segment structures created, a more detailed explanation of these structures is given in section 7.1 later in this note. The minimum value here is three for NTSC applications and one for PAL applications. Five is a reasonable value, but this number should be increased further for applications which use any complex screen layouts. The last parameter defines the memory bank used for Balboa memory allocations while in vs_open(). A typical call to this looks like :-

```
#define MAX_DSEG 5

         VS *vidscreen;

         vidscreen = vs_open( LCT_DOUBLE|CURSOR_ON, MAX_DSEG,BP_MEM_PLANEA );
```

4.2 Video Environment Manager

Applications which are not using the video screen manager but are using the cursor and hot spot manager need to use the video environment manager. The video environment manager has no specific initialisation function, however the function to create a video environment needs a path to the CDRTOS video device as input. One way to get this path is to use the video screen manager function vs_open_path() and then use the global vm_vidpath as the input to the video environment manager.

For applications which are using the video screen manager, the call to vs_open_path() is performed from within vs_init() and the call to create a video environment is performed from within vs_open() and these applications need not be concerned with the video environment manager.

4.3 Clut Manager

If applications are using the higher levels of the clut manager then it needs initializing for each video screen on which it is to be used. The first parameter to this is the video screen for which the clut manager is to be initialized. It must be separately initialized for each video screen where it is to be used. The second parameter is the usual Balboa memory type.

The last parameter defines the number of TCMAP structures to be created. These structures are used to record clut usage for the text manager when using anti-aliased text. For applications which are not using anti-aliased text, this number can be 0, otherwise it should be set to the maximum number of different combinations of foreground and background colours which the application expects to use. A more detailed explanation of these structures and how they are used is given in section 9.2 later in this note.

A typical initialisation looks like :-

```
#define TCMAP_STRUCTURE_COUNT 4

         VS *vidscreen;

         vidscreen = vs_open( LCT_DOUBLE|CURSOR_ON, 5, BP_MEM_PLANEA );
         cl_init( vidscreen, BP_MEM_DONTCARE, TCMAP_STRUCTURE_COUNT);
```

The initialisation calls for the clut manager, the matte manager and the video interrupt manager can be performed in any order. The only requirement is that all must follow the creation of a video screen using vs_open().

4.4 Matte Manager

Like the clut manager, the matte manager needs initializing for each video screen on which it is to be used. A typical initialisation looks like :-

```
         VS *vidscreen;

         vidscreen = vs_open( LCT_DOUBLE|CURSOR_ON, 5, BP_MEM_PLANEA );
         ma_install( vidscreen, BP_MEM_DONTCARE );
```

This call will also set some pointers to functions so that matte manager routines are called as part of each vs_update() call. These will slow down vs_update() considerably. Applications should use the calls ma_enable() and ma_disable() to enable/disable the calling of this matte manager code so that it is only called when it is really needed.

The initialisation calls for the clut manager, the matte manager and the video interrupt manager can be performed in any order. The only requirement is that all must follow the creation of a video screen using vs_open().

4.5 Video Sync Manager

Applications which use the video sync manager directly rather than through the video interrupt manager should use vsync_init() to initialize the manager and vsync_kill() before they exit. Without the vsync_kill() call, the application will not restart cleanly and a reset of the CD-I player will probably be required.

For applications which are using the video interrupt manager, the call to vsync_init() will be performed by vi_init() as described below. The call to vsync_kill() will be performed by vs_finish().

4.6 Video Interrupt Manager

Just as that the clut and matte managers are attached to video screens, the video interrupt manager is attached to video environments and must be specifically initialized for each video environment with which it is to be used. The initialisation call here is vi_init() which takes 4 parameters. The first two of these are the video environment pointer and the conventional Balboa memory type.

The other two parameters define the quantity of two internal structures created when the manager is initialized. The first of these will set the maximum number of scan lines which have video interrupt callbacks attached to them. It is strongly recommended that this number not exceed two or three since many more interrupts than that can saturate the OS9 interrupt mechanism. The second number will set the maximum number of callbacks which can be attached to those scan lines. There can be many more of these than the number of scan line structures.

Typical initialisation code for the video interrupt manager looks like :-

```
#define LINE_STRUCTURE_COUNT 2
#define CALLBACK_STRUCTURE_COUNT 4

        VS *vidscreen;
        vidscreen = vs_open( LCT_DOUBLE|CURSOR_ON, 5, BP_MEM_PLANEA );
        vi_init( vidscreen->vs_videnv,BP_MEM_DONTCARE,LINE_STRUCTURE_COUNT,
              CALLBACK_STRUCTURE_COUNT);
```

This function includes the call to vsync_init() required for the video sync manager. There is no explicit vi_kill() function, applications should either use the video screen manager function vs_finish() or call vsync_kill() themselves.

The initialisation calls for the clut manager, the matte manager and the video interrupt manager can be performed in any order. The only requirement is that all must follow the creation of a video screen using vs_open().

4.7 Other Managers

The picture, LCT, text and VFX managers do not have any specific initialisation functions of their own. However since the video effects library uses the video interrupt manager for timing, the video interrupt manager must be initialized before any video effects can be used.

5. Displaying an Image

5.1 The PICTURE Structure

The Balboa PICTURE structure is a container for video assets. To display an image in Balboa, an application should load the image data into an appropriate picture, attach the picture to a video screen and then update that video screen.

There are two ways of creating a PICTURE in Balboa, pi_create() and pi_iffpic(). The function pi_iffpic() is an interface to pi_create() which goes out to an IFF video file, creates a PICTURE to match the contents of the IFF file and then loads those contents into the PICTURE. This function can be very useful in prototyping but should not be used in titles or from a CD. Usage of this function requires the presence of the IFF library which is a considerable code size overhead. It is also very slow when working off a CD.

The PICTURE structure is a container for image data. The structure contains all the information needed to fully define how the picture is required to appear on the CD-I video hardware. When a picture is created by pi_create(), these are all set to sensible default values and no changes are required for simple images. The list of PICTURE structure contents includes :-

- picture size
- pixel data location in memory
- location on screen to display picture at
- section of picture to be displayed
- image coding type
- vertical and horizontal pixel resolution of image
- CD-I hardware values (transparency colour, mask colour, DYUV start value, pixel hold value, image contribution factor, transparency condition)
- application callback for when the picture is displayed
- pointer to palette data for picture

5.2 Displaying an Image From an IFF File

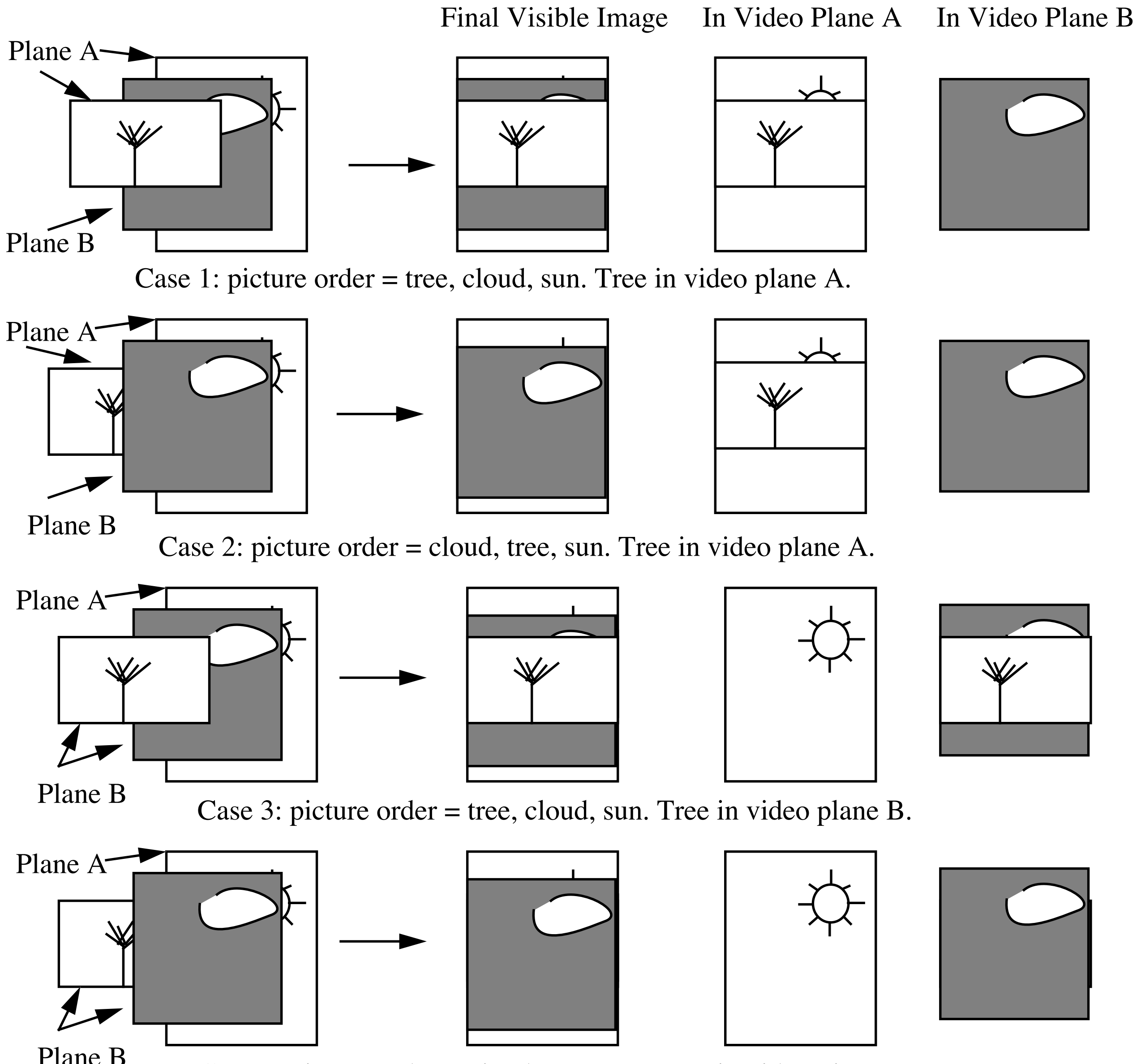
To display a picture, all that is needed is to attach that picture to a video screen and then to update that video screen. The absolute simplest code to do this is :-

```
VS *vidscreen;
PICTURE *pic;

vs_init(BP_MEM_DONTCARE,1);
vidscreen = vs_open( LCT_DOUBLE|CURSOR_ON, 5, BP_MEM_PLANEA );
pic = pi_iffpic( PLANE_A, 0, "my_file.dyu", BP_MEM_PLANE_A);
pi_front( vidscreen, pic );
vs_update( vidscreen, 0, DISPLAY_625, NULL, NULL );
vs_show( vidscreen );
```

This fragment introduces 3 functions which have not been previously mentioned, pi_front(), vs_update and vs_show(). The first of these, pi_front() is one of a set of functions which build an ordered list of PICTUREs attached to a video screen. This list, together with the contents of the individual PICTURE structures comprises a definition from the application of what should appear on the video output of the CD-I video hardware. The second of these functions vs_update() performs a mapping of these requirements onto the video hardware of the particular CD-I player.

In many ways, vs_update() is the key to the Balboa video managers. It takes the list of PICTUREs attached to a video screen and analyses this to find which PICTURE contributes to each scan line of the display in each of the two CD-I hardware video planes. The diagram below shows the effect of changes of video plane and order within the PICTURE list. The most obvious effect of the change of PICTURE list order is the appearance and disappearance of the tree picture. A less obvious effect only becomes apparent if transparency is used. If the tree picture is made partially transparent then the user of the application will see what is in the other plane which is the cloud picture in case 1 and the sun picture in case 3.



Case 1: picture order = tree, cloud, sun. Tree in video plane A.

Case 2: picture order = cloud, tree, sun. Tree in video plane A.

Case 3: picture order = tree, cloud, sun. Tree in video plane B.

Case 4: picture order = cloud, tree, sun. Tree in video plane B.

Another area that vs_update() addresses is PAL-NTSC compatibility. The third parameter to vs_update() is the video mode which a title has been authored to. The vs_update() function performs a mapping between this and the CD-I player video hardware mode. Where this parameter does not match the CD-I player hardware, vs_update() will keep the image central on the screen. This mapping works using a screen coordinate system consisting of 600 lines, numbered from -40 to 560. In the case of a PAL title on an NTSC player, 40 lines will be ignored at the top and bottom of the screen and so only lines 40-520 will be visible. For an NTSC title on a PAL player, 40 lines will be added to the top and bottom of the screen resulting in lines -40 to 520 being visible. The use of -40 as the top line on the screen rather than 0 is required in order to keep the title vertically centered.

The last of the 3 new functions in the code fragment, vs_show(), activates a particular video screen. This is required in order for that video screen to be visible. Under normal circumstances, an application need only do this once. The only exception to this is applications which need to use more than one video screen. This is quite rare since the LCTs associated with video screens normally consume 60K (NTSC) or 71K (PAL). Usage of more than one video screen would allow an application to instantaneously cut from one video display to another without the time required for vs_update() to perform its mapping.

One of the few situations where multiple video screens are likely to prove useful is where a title wants to use high quality QHY images but does not have the memory for the LCTs which that normally requires - 120K (NTSC) or 144K (PAL). By creating a special type of video screen, an application can cut to/from full screen high resolution images at a tiny fraction of the memory cost of the fully functional version. This special type of video screen is created by setting the LCT_MINIMUM bit set as well as the DCP_HIGH_RES bit in the flags input to vs_open(). In this case, vs_show() can be used to switch between a fully functional normal resolution video screen and a very reduced functionality high resolution one.

5.3 Displaying an Image From a Real Time File

The previous code fragment uses pi_iffpic() to load an image from an IFF file. As mentioned earlier, the use of this function in a real application is strongly discouraged. The recommended route for a real application is using a real time file, example code for which is shown on the next page.

The main differences between a real time file and a normal file are that the real time file access is asynchronous and that the data coming into memory arrives as an integer number of 2324 byte sectors. These differences are in addition to those required by the change from using pi_iffpic() to pi_create(). The example uses the Balboa play manager to handle the playback of the real time file. This is described in more detail in Appendix 2 for those who are not familiar with it.

The parameters to pi_create() as used in the example are video plane, coding type and 3 size parameters, x, y and the number of bytes to allocate. This parameter is especially important for real time files since the data arriving will be an integer number of sectors and so this parameter must be used to force the memory for the picture to also be an integer number of sectors. Without this, arbitrary application memory would be over-written by some of the data in the last sector of the image.

Due to the asynchronous nature of the real time file play, this example uses the Balboa play manager callback on end of play to display the image. If this was not done then the loading of the image from disc would be visible.

```
/*
         example1.c - load a dyuv image from a real time file and display it
*/

#include <modes.h>
#include <stdio.h>
#include <vm_vs.h>
#include <vm_pic.h>
#include <bp_mem.h>
#include <pm_low.h>
#include <status.h>
#include <vm_video.h>

/* various #defines */

#define FILE_NAME          "example.rtf"
#define PAL_WIDTH          384         /* PAL picture's width in pixels */
#define PAL_HEIGHT         280         /* PAL picture's height in scan lines */
#define PAL_BYTES          PAL_WIDTH * PAL_HEIGHT
#define MAX_INTERNAL_BUF   1           /* Max # of internal buffers */
#define MAX_DSEG           5           /* Max # of DSEG's for picture build up */
#define ARBITRARY_SIZE     4096        /* any conveniently big size */
#define F2_BYTES           2324        /* form 2 sector size */
#define F2_SECTORS(s)      ((F2_BYTES - ((s) % F2_BYTES)) + (s)) / F2_BYTES
#define NA                 0           /* Not Applicable */
#define UCM(x)             (x<<1)      /* Convert to UCM */
#define CHANNEL(c)         c           /* Trivial, for readability only */

/* now the global variables and function prototypes */

PICTURE *picture;                      /* the picture to use */
VS       *vsScreen;                    /* the video screen */
int      File;                         /* the OS9 file as returned by open()   */
char     clut_buffer[F2_BYTES];        /* one sectors worth */
void runit(),play_done(),vs_finish();

main()
{
         dispatch_loop( runit, NULL, NULL );
}

void runit()
{
         char  *buffer;

         /* play manager initialisations */

         sgm_init();   /* initialise signal manager, needed for play manager */
         buffer = (char*) bp_allocate( ARBITRARY_SIZE, BP_MEM_PLANEA );
         pmb_set_block( buffer, ARBITRARY_SIZE);
         pml_init();
         File= open( FILE_NAME, S_IREAD);

         /* video manager initialisations */

         vs_init (BP_MEM_DONTCARE, MAX_INTERNAL_BUF);
         vsScreen = vs_open( LCT_DOUBLE|CURSOR_ON, MAX_DSEG, BP_MEM_PLANEA );
```

```
        /* create the picture to load the data into */

        picture = pi_create( PLANE_A, D_DYUV, UCM(PAL_WIDTH), UCM(PAL_HEIGHT),
              F2_SECTORS(PAL_BYTES) * F2_BYTES, NA);

        /* declare the picture and the clut buffer to the play manager */

        pml_add_buffer(CHANNEL(0), VIDEO_TYPE,F2_SECTORS(PAL_BYTES),
                    picture->pi_pstart, NULL, NA, DISPATCHED);

        /* now go and do the play */

        pml_play( File, 0, /* position */
                    1, /* Potentially active mask: channel 0 */
                    0, /* direct_audio_mask */
                    1, /* nr. of EOR to mark End of Play */
                    0, /* channels to be switched between active/de-active */
                    play_done, NULL, NULL );
}

void play_done()
{
        /* display the picture */

        pi_front( vsScreen, picture );
        vs_update( vsScreen, 0, DISPLAY_625, NULL, NULL );
        vs_show( vsScreen );
}
```

Example 1 - Displaying a DYUV Image from a Real Time File

5.4 What It Will Do

Balboa will display any picture the application creates on any CD-I video setup. In some cases, this will require translations, for example a high resolution picture on a normal resolution display will have lines dropped. A normal resolution picture on a high resolution picture will have lines repeated. If the picture needs reload display start address LCT instructions on each line then these will be written. If the picture needs reload DYUV start value instructions on each line then these will be written. PAL pictures on NTSC players and vice-versa will be vertically centred as appropriate.

If there is something about the picture which Balboa cannot handle then the application can specify a callback function which will be called when the picture is displayed and will which can then do whatever it needs. See pi_install_cb() for more information on this feature. The Balboa functions pi_lct_clut() and pi_clutshade() are examples of this for functionality so specialized that it was not relevant to include it in the core of Balboa.

5.5 Using Clut Images

The examples above only deal with DYUV images, clut images need more work. The Balboa clut manager contains two levels of support for these depending on how complex the needs of the application are. For applications which always use the whole clut at once, the simpler interface is more appropriate. Applications which use lots of blit-ed graphics or anti-aliased text will probably find the more complex interface closer to their needs.

The simple clut manager interface consists of a call called cl_write(). This call writes clut data from memory into the FCT of a video screen. Use of this call does NOT require the clut manager to be initialized but does require temporary use of one of the internal buffers mentioned in the description of the vs_init() call on page 4.

The higher level clut manager interface includes a mechanism for tracking usage of CLUT space and hence does require cl_init() to have been called. Also, this level has the concept of a CLUT structure which the lower level does not. The equivalent of cl_write() at this level is cl_exec(), this function checks that.the clut space it has been asked to write to is not used and then marks it as used before writing to it. Once an application has finished with some clut space, cl_free() can be used to mark the clut space concerned as no longer used. If an application does not care about any previous clut space usage then cl_clear() will mark all of the clut as being unused.

The simple pi_iffpic() function will create a CLUT object if the file specified contains an IFF PLTE chunk. To make the code fragment on page 7 handle clut files, the following line should be added after the call to pi_iffpic() :-

```
         if( pic->pi_clut ) cl_exec( vidscreen, PLANE_A, pic->pi_clut );
```

Since pi_create() does not know whether an image has a palette or how large that palette is, it does not create any CLUT objects or use the pi_clut structure entry. To upgrade the first complete example to handle clut images, the normal approach is to put the clut data in a data sector and make that a clut object.A new version of the example program on page 10 which shows this behaviour is on the next page. The changes for CLUT support are marked with change bars in the margin.

```
/*
         example2.c - load a clut image from a real time file and display it
*/
#include <modes.h>
#include <stdio.h>
#include <vm_vs.h>
#include <vm_pic.h>
#include <bp_mem.h>
#include <pm_low.h>
#include <status.h>
#include <vm_video.h>

/* various #defines */

#define FILE_NAME           "example.rtf"
#define PAL_WIDTH           384         /* PAL picture's width in pixels */
#define PAL_HEIGHT          280         /* PAL picture's height in scan lines */
#define PAL_BYTES           PAL_WIDTH * PAL_HEIGHT
#define MAX_INTERNAL_BUF    1           /* Max # of internal buffers */
#define MAX_DSEG     5                  /* Max # of DSEG's for picture build up */
#define ARBITRARY_SIZE      4096        /* any conveniently big size */
#define F2_BYTES            2324        /* form 2 sector size */
#define F2_SECTORS(s)       ((F2_BYTES - ((s) % F2_BYTES)) + (s)) / F2_BYTES
#define NA                  0           /* Not Applicable */
#define UCM(x)              (x<<1)      /* Convert to UCM */
#define CHANNEL(c)          c           /* Trivial, for readability only */

/* now the global variables and function prototypes */
```

```
  PICTURE *picture;                     /* the picture to use */
  VS      *vsScreen;                    /* the video screen */
  int     File;                         /* the OS9 file as returned by open()   */
| char    clut_buffer[F2_BYTES];        /* one sectors worth */
  void runit(),play_done(),vs_finish();
  main()
  {
          dispatch_loop( runit, NULL, NULL );
  }
  void runit()
  {
          char  *buffer;

          /* play manager initialisations */

          sgm_init();   /* initialise signal manager, needed for play manager */
          buffer = (char*) bp_allocate( ARBITRARY_SIZE, BP_MEM_PLANEA );
          pmb_set_block( buffer, ARBITRARY_SIZE);
          pml_init();
          File= open( FILE_NAME, S_IREAD);

          /* video manager initialisations */

          vs_init (BP_MEM_DONTCARE, MAX_INTERNAL_BUF);
          vsScreen = vs_open( LCT_DOUBLE|CURSOR_ON, MAX_DSEG, BP_MEM_PLANEA );
|         cl_init( vsScreen, BP_MEM_DONTCARE,4 );

          /* create the picture to load the data into */

|         picture = pi_create( PLANE_A, D_CLUT7, UCM(PAL_WIDTH), UCM(PAL_HEIGHT),
                F2_SECTORS(PAL_BYTES) * F2_BYTES, NA);

          /* declare the picture and the clut buffer to the play manager */

          pml_add_buffer(CHANNEL(0), VIDEO_TYPE,F2_SECTORS(PAL_BYTES),
                      picture->pi_pstart, NULL, NA, DISPATCHED);
|         pml_add_buffer(CHANNEL(0), DATA_TYPE,1,clut_buffer,NULL,NA,DISPATCHED);

          /* now go and do the play */
          pml_play( File, 0, /* position */
                      1, /* Potentially active mask: channel 0 */
                      0, /* direct_audio_mask */
                      1, /* nr. of EOR to mark End of Play */
                      0, /* channels to be switched between active/de-active */
                      play_done, NULL, NULL );
  }
  void play_done()
  {
          /* display the picture */

          pi_front( vsScreen, picture );
|         cl_exec( vsScreen, PLANE_A, ((CLUT*) clut_buffer) );
          vs_update( vsScreen, 0, DISPLAY_625, NULL, NULL );
          vs_show( vsScreen );
  }
```

Example 2 - Displaying a CLUT Image from a Real Time File

A more advanced topic concerned with using clut images comes where an application needs to be able to seamlessly cut from one clut 8 image to another. Synchronising this so that the clut changes at the same time as the image requires double buffering of the FCT. This is done at the time a video screen is created by setting the FCT_DOUBLE bit in the first parameter to vs_open. For example :-

```
VS *vidscreen;

vidscreen = vs_open( LCT_DOUBLE|FCT_DOUBLE|CURSOR_ON, 5,
      BP_MEM_PLANEA );
```

The Balboa call to setup for a synchronised change of image and clut is vs_switch_fct(). This call sets flags so that on the next call to vs_update() FCTs will be changed as well as LCTs. Once vs_switch_fct() has been called, any clut manager calls to write to the FCT will be re-directed to the new FCTs rather than the set currently in use. If this was not done then there would probably be a momentary flash when the images were changed due to the old image being shown with the new clut or vice versa.

Here is an example code fragment showing the use of this feature :-

```
PICTURE *pic1, *pic2;

         pic1 = pi_create( PLANE_A,D_CLUT8,768,560,47*2324,0);
         pic2 = pi_create( PLANE_A,D_CLUT8,768,560,47*2324,0);

         /* load data and clut into pic1 and pic2 */
         ....
         /* display the first image */

         pi_front(vidscreen,pic1);
         cl_exec(vidscreen,PLANE_A,pic1->pi_clut);
         vs_update(vidscreen,0,DISPLAY_625,NULL,NULL);

         /* switch to the second image */
         vs_switch_fct(vidscreen);
         cl_clear(vidscreen);/* new FCT so clear the clut manager */
         pi_front(vidscreen,pic2);
         cl_exec(vidscreen,PLANE_A,pic2->pi_clut);
         vs_update(vidscreen,0,DISPLAY_625,NULL,NULL); /* the seamless cut */
```

In Balboa 1.3, there is a bug in vs_switch_fct() which results in a bus trap. This bug is new to 1.3 and fixed in 1.3.1 and following versions. The work around is to replace the call with the equivalent ’C’ code :-

```
vs -> vs_flags ^= VSF_WHICH_FCT;
vs->vs_videnv->ve_fct_a= vs->vs_fcts [((vs->vs_flags&VSF_WHICH_FCT)>>16)];
vs->vs_videnv->ve_fct_b= vs->vs_fcts [((vs->vs_flags&VSF_WHICH_FCT)>>16)+1];
vs -> vs_flags |= VSF_FCT_SWITCHED;
```

Use of double buffered FCTs may result in memory fragmentation. A detailed description of this is rather involved and is contained in Appendix 6.

5.6 Changing PICTUREs Between Coding Methods

One of the key requirements for robust CD-I applications is to minimise the repeated allocation and de-allocation of memory. One of the design requirements for Balboa was to be able to change the image coding type of a PICTURE which cannot be done with a CDRTOS drawmap since the structures for that are read only to the application.

The Balboa function to do this is called pi_set_type(). This function will change the coding type of a PICTURE into any of the other coding types. At the time of writing (Balboa 1.3.2 beta), this function does not test that the PICTURE structure has enough memory for the destination type. The two main examples of this are changing a PICTURE created as runlength into any other type and changing a picture into QHY or RGB555 which was not created as that type. Both of these changes of coding method should not be attempted by applications.

Another specific instance where changing the coding method can be confusing is changing from DYUV/CLUT to runlength. PICTUREs created as runlength do not have a line pointer table (pi_lstart) and will always be displayed starting at line 0 using the address in pi_pstart. PICTUREs created as another coding method and then changed to runlength will have a line pointer table. In Balboa 1.3, if the pi_lstart structure entry is not null then it is assumed to be a valid line pointer table for the runlength image and it will be used as such. This means that a PICTURE created as runlength may not display correctly. In this instance, it is recommended to preserve the contents of the pi_lstart structure entry, set it to NULL and then return it to the previous value when the coding method is changed from runlength or before the PICTURE is deleted.

Here is a simple example of pi_set_type() showing the approach recommended in the previous paragraph. This example uses the pi_application field to save the old contents of the pi_lstart structure entry, if an application is using this field then another place should be found to save this value. The Balboa function pi_lct_clut() also uses this field and so will conflict with this example.

```
PICTURE *pic1;

         pic1 = pi_create( PLANE_A,D_CLUT8,768,560,47*2324,0);

void set_to_rl()
{
         pic1->pi_application = (void*) pic1->pi_lstart;
         pic1->pi_lstart = NULL;
         pi_set_type( pic1, D_RL7 );
}

void set_to_clut()
{
         pic1->pi_lstart = pic1->pi_application;
         pi_set_type( pic1, D_CLUT8);
}
```

6. Playing a Slideshow

2 vs 4 pictures, basic elements including play manager and use of map files.

6.1 Using QHY in Sequences

(tbd)

7. How to get Round Balboa When it is Too Slow

The high level interface to the video managers via the various structures and vs_update() is very powerful and generic. This means that in some circumstances it is just too slow. Balboa does include support to allow the programmer to work at the CDRTOS level in these cases but these facilities are not very well described in the programmers guide.

7.1 Display Segments

As well as doing the mapping described previously, the vs_update() function generates a set of data structures called display segments to describe what it did. These structures are the primary mechanism by which applications can access LCTs in a way which cooperates with Balboa rather than fighting it. An understanding of these is fundamental to being able to access the LCTs in a way which is compatible with Balboa rather than fighting it.

Each display segment structure describes a number of consecutive scan lines all of which share the same PICTURE in plane A and the same PICTURE in plane B. A linked list of these fully defines the CD-I display which vs_update() generated. In Balboa 1.3, this structure looks like :-

```
typedef struct vm_dseg {
        struct vm_vs     *ds_vs;          /* pointer to associated video screen */
        struct vm_dseg   *ds_next;        /* display segment below this one on the screen */
        PICTURE          *ds_pic0;        /* picture to use for plane a */
        PICTURE          *ds_pic1;        /* picture to use for plane b */
        LCT              *ds_lct0;        /* lct to use for plane a */
        LCT              *ds_lct1;        /* lct to use for plane b */
        int              ds_icm;          /* image coding instruction */
        int              ds_tci;          /* transparency control info instruction */
        short            ds_screenline;   /* start line on screen of this segment */
        short            ds_nlines;       /* number of lines in segment */
        char             ds_plord;        /* plane order, 0 or 1 */
        char             ds_reserved;     /* reserved */
        short            ds_line0;        /* start line ( logical ) for plane 0 LCT */
        short            ds_line1;        /* start line ( logical ) for plane 1 LCT */
} DSEG;
```

On the first line of a display segment, certain instructions are written into certain fixed locations in the LCT. These locations are :-

	LCT Column Number							
Plane	0	1	2	3	4	5	6	7
A	display start address	display parameters	image contrib-ution factor	image coding methods	plane order	transparency control	spare	spare
B				spare	spare	spare	spare	spare

7.2 Accessing LCTs Yourself

In order for an application to be able to use the CDRTOS dc_wr???? calls to write to LCTs, it must be able to obtain the UCM LCT id. The best way to do this is using the video screen manager function vs_find_segment() which returns a pointer to the display segment structure which describes a given scan line. As can be seen from the structure listing on the previous page, once this is available, everything else the application might want to know is a pointer or two away. As an example, here is a code fragment which will get the UCM LCT id for the plane A LCT at line 120 :-

```
        DSEG *ds;
        VS *vidscreen;
        int id, lctline;

        ds = vs_find_segment( vidscreen, 120 );
        id = lc_idof( ds->ds_lct0 );
```

A more complex example of this is in section 7.3 below, creating a rectangular matte using CDRTOS functions. The other input required before UCM level functions can be used is the line number within the LCT. The line number on the screen and the line number within the LCT can be the same but do not have to be. A specific example of where they are not is the case of an NTSC designed title running on a PAL player or a PAL title on an NTSC player. Using the same variables as the previous code fragment, here is how to derive this for scan line 120 :-

```
        lctline = 120 - ds->ds_screenline + ds->ds_line0;
```

Putting these two fragments together, you could create a new function vs_wrli() which took almost the same parameters as the UCM dc_wrli() but included these two translations :-

```
int vs_wrli( videoscreen, plane, line_number, column_number, instruction )
VS *videoscreen;
int plane, line_number, column_number, instruction;
{
        DSEG *ds;
        int id, lctline;

        ds = vs_find_segment( vidscreen, line_number );
        if( plane )
        {
              lctline = line_number - ds->ds_screenline + dc->ds_line1;
              id = lc_idof( ds->ds_lct1 );
        }
        else
        {
              lctline = line_number - ds->ds_screenline + dc->ds_line0;
              id = lc_idof( ds->ds_lct0 );
        }
        return dc_wrli( vm_vidpath, id, lctline, column_number, instruction);
}
```

When an application is working at this level, another factor which should be considered is whether to use the LCT manager or not. If all an application needs to do is to update the fixed instructions at the start of a display segment, then using the LCT manager will generate an error since these locations were marked as in use at that time.

Applications which need to write to arbitrary locations within the LCT have two choices to avoid breaking future calls to vs_update(). Either the LCT manager should be used to mark the LCT space so that vs_update() is aware of it or the application must remove those LCT instructions before the next call to vs_update(). Any other approach will leave instructions around in the LCT to cause visual glitches some arbitrary time later. An example of the first of these is below.

7.3 A Simple Rectangular Matte

One common reasons for wanting to write directly to the LCT is to create a simple rectangular matte. The Balboa matte manager could be used to do this but it would be complete overkill and would slow down any subsequent calls to vs_update(). A brief description of the CD-I matte mechanism is given in Appendix 3.

To create a rectangular matte 3 LCT instructions need to be written. On the first line of the matte, each side of the matte requires one instruction to define it. On the line after the matte, one instruction is required to turn off the matte mechanism. A simple function to setup such a matte could look like :-

```
make_rect_matte( vs, plane, mw )
VS             *vs;
int            plane;
SIZE_RECT     *mw;
{
         DSEG    *ds;
         LCT     *lct;
         short   top_column, top_line, btm_column, btm_line;
         int     instructions[2];
         int     left_x = mw->ul.x,
               top_y   = mw->ul.y,
               right_x = left_x + mw->width,
               bottom_y= top_y + mw->height;

         /* decide which LCT we need to write to and where */

         ds = vs_find_segment( vs, top_y );
         if( plane == PLANE_B )
         {
               lct = ds->ds_lct1;
               top_line = top_y    - ds->ds_screenline + ds->ds_line1;
               btm_line = bottom_y - ds->ds_screenline + ds->ds_line1;
         }
         else
         {
               lct = ds->ds_lct0;
               top_line = top_y    - ds->ds_screenline + ds->ds_line0;
               btm_line = bottom_y - ds->ds_screenline + ds->ds_line0;
         }

         /* get two spaces on the first line of our rectangle */

         top_column = lc_request( lct, top_line, 1, 2); /* 1 line, 2 columns */

         /* check that we did get what we asked for */
```

```
        if( top_column==-1)
        {
              STATUS(ST_APPL,ST_ERROR,errno,"ran out of LCT space !");
              return -1;
        }
        /* build the instructions we want to write */

        instructions[0]=
        cp_matte( 0,      /* matte register zero */
                    MO_SET, /* set matte flag to true */
                    0,      /* use matte flag zero */
                    0x3f,   /* the image contribution factor, not used here */
                    left_x);/* the left hand side of the rectangle */

        instructions[1]=cp_matte(1, MO_RES, 0, 0x3f, right_x);

        /* now do the same for the single instruction at the end. This should
        be on the first scan line which is not required to be transparent,
        hence the +2 on the next line */

        btm_column= lc_request( lct, btm_line,1,1); /* 1 line, 1 column */
        if( btm_column == -1 )
        {
              STATUS(ST_APPL,ST_ERROR,errno,"ran out of LCT space !");
              return -1;
        }

        /* write the instruction to turn off mattes */

        dc_wrli(vm_vidpath,lc_idof(lct),btm_line,btm_column,cp_matte(
              0,          /* matte register zero */
              MO_END,     /* end of all matte registers */
              0,          /* matte flag zero */
              0x3f,       /* icf, not used here either */
              0));        /* position doesn't matter, just turn it off */

        /* write the starting instructions last to avoid scan sync problems */

        dc_wrlct( vm_vidpath, lc_idof(lct), top_line,
              top_column, 1, 2,instructions );

        /* done */

        return(OK);
}
```

For the matte to become visible, the picture which is required to be transparent should be setup to be transparent when matte flag zero is true. There are two ways of doing this, either using vs_update() or by just writing it to the LCT. If vs_update() is used, the call to make_rect_matte() must follow the call to vs_update() otherwise vs_update() will delete the matte instructions. Using the vs_update() route, this could look like :-

```
PICTURE *pic1;
VS *vidscreen;
```

```
        pi_settrans( pic1, TC_MF0TRUE);
        vs_update(vidscreen,0,DISPLAY_625,NULL,NULL);
        make_matte_rect( vidscreen, 100,100,600,200);
```

As mentioned above, vs_update will delete the matte instructions from the LCT before it regenerates the display. If an application needs to make changes to the display but keep the matte up then the best way to do this is using the hooks provided by the last two parameters to vs_update() which allow an application to provide a function which will be called as part of the execution of vs_update(), just before the results of the vs_update() are displayed. This function is passed a definition of what vs_update() has just built in a VS_STATE structure.

Using the previous example function, code to do this could look like :-

```
        PICTURE *pic1;
        VS *vidscreen;
        void my_update_func();

        /* setup transparency and then activate the matte */

        pi_settrans( pic1, TC_MF0TRUE);
        vs_update( vidscreen, 0, DISPLAY_625, my_update_func ,NULL);

        /* do some more stuff */
        .......

        /* as long as my_update_func is the fourth parameter to vs_update() the matte will
        be re-created as part of each vs_update() call *.

        vs_update( vidscreen, 0, DISPLAY_625, my_update_func, NULL);


void my_update_func(state)
VS_STATE *state;
{
        make_matte_rect( state.vs, 100,100,600,200);
}
```

8. Playing a DYUV Movie

An example which brings together much of what has been discussed previously is playing some partial screen motion video from a real time file. The example discussed here is a simple example of how to play a DYUV movie, such as those used with Media Mogul. A complete listing of this program is contained in appendix 1.

The Media Mogul dyuvmovie format has a header followed by a set of frames all of identical size. There is a program called animinfo which prints out various information from that header. For the movie file used here, the output from animinfo is :-

```
+ animinfo /emu1/SIMPLE/BH10FPS.dym
********************************************************************
             ANIMINFO   v1.0
             Compacted file: /emu1/SIMPLE/BH10FPS.dym
********************************************************************
Magic Number:                  -892726732
File Type:                     DYUV
Size of Clut:                  0
File Count:                    612
Buffer (Frame) Size:           16000
X:                             0
Y:                             0
Width:                         160
Height:                        100
Linelength:                    160
Pltereserved:                  0
Pltecount:                     0
Ystart:                        128
Ustart:                        128
Vstart:                        128
Largest Frame:                 0
Largest 5 Frames:              0
Largest 10 Frames:             0
Largest 15 Frames:             0
```

From this are derived some #defines which define the size of the movie and also the layout of the frames in the real time file.

```
#define MOVIE_WIDTH        160   /* width of movie in pixels     */
#define MOVIE_HEIGHT       100   /* height of movie in scan lines    */
#define MOVIE_BYTES        MOVIE_WIDTH * MOVIE_HEIGHT
```

In addition, two other #defines are used to define the location of the movie on the screen :-

```
#define MOVIE_X            100   /* movie x position ( ucm coordinates ) */
#define MOVIE_Y            100   /* movie y position ( ucm coordinates ) */
```

The real time file for this movie is generated by a script which looks like :-

```
block
        video in channel 0 from
                "/u3/jon/data/IFF/girlski1.d">
block
        video in channel 1 from
                "BH10FPS.dym" [dyuv] offset 68 length 16000 at 0,
                "BH10FPS.dym" [dyuv] offset 16068 length 16000 at 7 ,
                "BH10FPS.dym" [dyuv] offset 32068 length 16000 at 15 ,
                "BH10FPS.dym" [dyuv] offset 48068 length 16000 at 22 ,
```

This real time file starts with the background image in channel 0 and then continues with the frames of the movie. Each frame is defined by a byte offset of where it starts in the DYUV movie file, a length and a starting sector. The length is taken straight from the output of animinfo. The offsets start with the size of the DYUV movie file header (in this case 68) and then increase by the frame size each time. The starting sectors are generated in such a way as to achieve an average frame rate of 10 frames/ second.

The playback program uses one full screen PICTURE to contain the background image and 3 smaller PICTUREs into which the individual frames of the movie are decoded as they come in from disc. The three smaller PICTUREs are in the opposite video plane from the full screen PICTURE, this is so that mattes may be used to create a transparent window in the full screen PICTURE through which the movie is visible as it plays.

One of the outputs from animinfo was Y, U and V start values of 128,128,128. These are different from the default so the loop which creates the small PICTUREs also sets up the pi_dstart entry to be 128,128,128. Also in this loop is a call to bp_memset() to clear each of the small PICTUREs to all zeros. This call is required because pi_create() does not initialise the pixel memory and for DYUV data in the middle of a scan line to display correctly, the pixels to the left of it must be zero.

```
#define Y_START      0x80
#define U_START      0x80
#define V_START      0x80

         piMain = pi_create( PLANE_A, D_DYUV, UCM(PAL_WIDTH), UCM(PAL_HEIGHT),
               F2_SECTORS(PAL_BYTES) * F2_BYTES, NA);
         for( x = 0; x < MAX_OUTBUF; x++ )
         {
               piPartial[x]=pi_create(PLANE_B, D_DYUV, UCM(PAL_WIDTH),
                                       UCM(MOVIE_HEIGHT), NA, NA);
               piPartial[x]->pi_dstart = cp_yuv( PB, Y_START,U_START,V_START);
               bp_memset (piPartial[x]->pi_pstart, 0, piPartial[x]->pi_size);
         }
         /* setup the display */

         pi_settrans( piMain, TC_MF0TRUE);
         pi_front( vsScreen, piMain );
         pi_back( vsScreen, piPartial[0]);
         pi_position( piPartial[0], srMovieWin.ul.y);
         vs_update( vsScreen, NON_INTERLACE, DISPLAY_625, NULL,NULL );
```

This program uses 3 buffers to hold the frames as they come in from CD. The function pml_add_buffer() is used once for each of these to build a circular set of 3 PML_BUFFER structures to hold the frames. Each time one of these buffers becomes full, the function movieCB() is called.

```
#define MAX_INBUF          3     /* Max # of input buffers */

         /* allocate the memory for the input buffers and make them known to the
         play manager */

         for(x=0; x < MAX_INBUF; x++)
         {
               InBuf[x]=(char*)bp_allocate(F2_SECTORS(MOVIE_BYTES) * F2_BYTES,
                     BP_MEM_DONTCARE);
               pml_add_buffer (CHANNEL(1), VIDEO_TYPE, F2_SECTORS(MOVIE_BYTES),
                     InBuf[x], movieCB, NA, DISPATCHED);
         }
```

In order to use the circular facility for handling the frames from CD, the full screen image must be in a different channel. In this case, the full screen image is in channel 0 and the movie in channel 1. The buffer full callback facility of the play manager is used here so that when the full screen image is loaded, the video screen will be activated and transparency enabled through to the movie using the matte code described previously in section 7.4.

```
         pml_add_buffer(CHANNEL(0), VIDEO_TYPE,F2_SECTORS(PAL_BYTES),
                        piMain->pi_pstart, background, NA, DISPATCHED);

SIZE_RECT   srMovieWin =         /* Size and position of movie window */
{
    {
         MOVIE_X, MOVIE_Y        /* ul.x, ul.y */
    },
    UCM(MOVIE_WIDTH),            /* width    */
    UCM(MOVIE_HEIGHT)            /* height   */
};

void background(context,buffer)
int context;
PML_BUFFER *buffer;
{
         /* show the image by showing the video screen now */

         vs_show( vsScreen );

         /* put up the matte for this movie. We will use plane B for the LCT
         instructions since that normally has more space in it. This has
         no effect on the results of the matte */

         make_rect_matte (vsScreen, PLANE_B, &srMovieWin);
}
```

As each frame is loaded from CD, the function movieCB() is called. This function implements a form of graceful degradation such that if frames are arriving faster than they can be decoded then it will drop frames. This allows for an application to be doing other things while the DYUV movie is playing. This graceful degradation is implemented through a variable called pending and by splitting the actual frame decoding into a separate function called display_frame(). If there is not a frame waiting for display then movieCB() saves the address at which the last frame was loaded into a variable called source and then dispatches a call to display_frame().

```
int      Pending = 0;            /* flag to say whether a frame is*/
                                 /* waiting to be decoded*/
char     *Source;                /* the source data for the next frame*/
                                 /* to be decoded*/

void movieCB (context, buffer)
int context;
PML_BUFFER *buffer;
{
         void display_frame();

         /* if we already have a frame pending for decoding then forget about
         this one. If not then we do have one pending now */

         if( Pending )   return;
         Pending ++;

         /* remember where this data was loaded for the decode function */

         Source = buffer->Buf;

         /* dispatch the decode function */

         dispatch_function( 0,display_frame,0);
}
```

The display_frame() function can be split into three parts, the byte copying, switching the display to the just copied image and various housekeeping. The byte copying consists of two for loops which copy 2 bytes at a time using pointers to shorts. This usage of short pointers is why the references to MOVIE_X and MOVIE_WIDTH are divided by two in the example below. The source address for these bytes is the source variable written to by the movieCB() function. The destination address is taken from the line start table of the next one of the 3 small pictures to be used. This next one is shown by a variable called output.

```
int      Output = 1;             /* the next output picture to use*/

void display_frame()
{
         register    unsigned short *dest;
         register    unsigned short *local_source;
         register             int    x, y;
         unsigned int    *linestart;

         /* sort out the source and destination. This copy function is slightly
         optimised and does 2 bytes at a time, hence we have to cast the source
         pointer to be a pointer to unsigned short */

         local_source = (unsigned short*) Source;*/
         linestart = (unsigned int*)( piPartial[ Output ]->pi_lstart);
         /* loop copying */

         for( y = 0; y < MOVIE_HEIGHT; y++)
         {
               /* sort out where we are copying to */

               dest = (unsigned short*)( (*(linestart++)) + MOVIE_X / 2 );
               for (x = 0; x < MOVIE_WIDTH / 2; x++)   /* /2 => short vs. char */
                     *(dest++) = *(local_source++);
         }
```

The display is switched between the 3 small PICTUREs by writing a new load display start address instruction to the LCT for the plane containing the small PICTUREs at the top of the display segment which includes them. This display segment is found once during the start-up of the program by using vs_find_segment() and then remembered for later use in a variable called playing.

```
DSEG     *dsPlaying;             /* the display segment in which the*/
                                 /* movie is playing */

         /* get the address of the display segment in which the movie is playing */

         dsPlaying = vs_find_segment (vsScreen, srMovieWin.ul.y);
```

The actual switching between buffers is accomplished by the following call to dc_wrli(). Most of the parameters to this have been discussed previously, the only new one is the use of the pi_pstart entry of the picture we want to display as being the start address for the video.

```
         dc_wrli( vm_vidpath, lc_idof( dsPlaying->ds_lct1), dsPlaying->ds_line1,
               DSEG_INS_DADR, cp_dadr((int)(piPartial[Output]->pi_pstart)) );
```

Finally at the end of the display_frame() function there is a little housekeeping code. There are two things here. Firstly the output variable which tracks the next of the 3 small PICTUREs to be used is incremented and set back to zero when it reaches the end of the set of PICTUREs. Also the pending variable is cleared so that movieCB() is now free again to setup the decoding of another frame of the movie.

```
         /* increment the output buffer counter, looping back to zero when
         we have used all the buffers */

         if( ++ Output == MAX_OUTBUF ) Output = 0;

         /* clear the frame pending flag so we display another one */

         Pending = 0;
}
```

9. Text

9.1 Font Formats

9.2 How the Clut Manager Can Help You

10. Introduction to Video Synchronisation

Video synchronisation is critical to achieving high quality CD-I titles. The CD-I system offers a number of ways of addressing these issues and Balboa includes two managers specifically targeted at this area, the video interrupt manager and the video synchronisation manager. As mentioned earlier, the video synchronisation manager is a low level manager which the video interrupt manager calls and the discussion here will only address the video interrupt manager level.

10.1 CDRTOS Video Synchronisation Facilities

CDRTOS includes support for two specific methods of video synchronisation. Both rely on an instruction which can be written to LCTs or FCTs which causes an interrupt to be generated when the display reads that instruction. By writing this instruction into a specific line of an LCT or into the FCT, an application can be notified that the display scan has passed that point. There are two ways of in which applications can be notified one of these interrupts has happened, one is using the CDRTOS call dc_ssig() to arm a signal to be sent to the application. The other is by linking to the OS9 event called "line_event" and then waiting for that event to be pulsed by the CDRTOS video driver.

The Balboa video synchronisation manager contains a very efficient interface to the event option involving a system state trap handler to minimise OS9 overheads in user state to system state transitions. Balboa used to use the signal method but due to the inherent delays in the OS9 signal mechanism, that can be very unpredictable.

10.2 Strategies, Pitfalls and Warnings

The simplest way to minimise video synchronisation problems is using double buffering. The CD-I hardware includes a number of features which make this much easier than many computer systems. The ability to point the display hardware at any address in 512K makes double buffering of video information very easy and efficient. In addition to this there are facilities for double buffering both LCTs and FCTs. Of course, double buffering has a cost, the memory used for the alternate copy.

As mentioned previously in the discussion of vs_update(), Balboa makes extensive use of double buffered LCTs so that it can build LCTs without having to worry about explicit scan synchronisation. The example code in this section is a hardware scroll which also uses double buffered LCTs as does the Balboa sprite manager.

One of the major causes of video synchronisation problems in CD-I is the LCT/FCT mechanism itself. It is very easy to assume that once an LCT instruction has been updated that the change will have immediate effect. In fact, the change will only take effect the next time the display scan reads that instruction which could be anything upto one field time later. One very good example of this is vs_update(), this function returns as soon as it has done its work. If an application requires to know when those results will be visible, it should use the function vs_update_done() to provide a specific callback for that purpose.

Some applications may have compelling reasons for not wanting to wait for end of field. For these applications, triple buffering can be useful if they have enough memory. With three buffers, an application can start accessing the third buffer where it would otherwise have to wait for the second buffer to start being displayed. The DYUV movie example earlier in this document uses three output buffers specifically for this reason.

Another use for the interrupt on a scan line facility is to generate a time base from the video signal of the CD-I player. Where this is the case, applications must take great care that the timing remains the same between PAL and NTSC players. To help in this, Balboa provides a global variable, vm_display_freq which contains the display frequency of the player on which the application is running. This variable is setup by vs_init().

10.3 Some Hidden Nasties

If an application does not have the memory for double buffering then it will have to use the interrupt on a scan line facility so that it can know when it is safe to access the currently visible screen or active LCT. This can take quite a lot of care to get right in a way which will still work when CD-I players with faster processors appear. A good example of this is where an application is doing something whose effect moves down the screen more slowly than the display scan. In this case, the application could set a display interrupt on a scan line near the top of the screen and then chase the display scan down the screen and never catch it on a current player. Such a strategy could easily break on a future CD-I player with a faster processor if the processor is fast enough such that the action of the processor now catches the display scan instead of being behind it all the time.

One particular problem where video synchronisation may be required is where instructions are being written to the currently active LCT. The path into the CD-I video memory need only be 16 bits wide and the video processor can steal cycles between the two 16 bit accesses needed to write an LCT instruction. If the video hardware was reading the same instruction as the application was writing then it is theoretically possible that the video hardware could read 16 bits of the previous value and 16 bits of the new value. I have not seen an instance of this being a problem but if an application experiences some form of flashes which are one field in duration then it may be worth looking at scan synchronising LCT accesses.

Another hidden nasty concerns the CDRTOS call dc_flnk(). Due to historical differences between the Philips CD-I chips and the Green Book, this function has a lot more work to do than the Green Book description implies. Calls to this function should generally be scan synchronised to avoid it being called during the vertical retrace. Using the function dc_llnk() to connect LCTs together can reduce this problem. In Balboa vs_update() normally switches LCTs using dc_flnk(). By specifying the option VSF_LLNK in the flags input to vs_open(), vs_update() will use dc_llnk() instead. This might be something to try if experiencing obscure scan synchronisation problems near where vs_update() is called.

The possibilities of deriving a time base from the video display have been mentioned above. Applications should take great care with this since there are actually 3 independent clocks in the CD-I player which may drift with respect to each other. These are the audio/cd clock, the video clock and the 100Hz system timer. For more discussion of this subject, there is an application note from Philips IMS Eindhoven, "Various time bases in CD-I", number TSA-003, dated 5th March 1992.

10.4 The Video Interrupt Manager

The Balboa video interrupt manager provides an interface onto the hardware scan line interrupt facility. Using it, the application can specify a callback to be called when the display passes a particular scan line. They key function involved in this is vi_add() which creates such callbacks. A typical call to this could look like :-

```
VIDEO_INT *vidint;
VS *vsScreen;
void my_cb();

vidint = vi_add(
      vsScreen->vs_videnv,    /* the video environment */
      100,              /* the scan line to wait for */
      PLANE_A,          /* plane to write the LCT instruction */
                        /* into. Normally does not matter which */
      0,                /* number of fields to skip between each */
                        /* calling of the function. */
                        /* 0-call each field, 1-call alternate fields */
      0,                /* maximum number of calls to function. 0-none */
      my_cb,            /* function to call */
      0,                /* parameter for the function */
      DISPATCHED);      /* mode for callback. IMMEDIATE should be used */
                        /* with great care */
```

There may be a start-up delay of upto one field time between the calling of vi_add() and the first opportunity to generate the callback. This is because the only way to identify more than one scan line interrupt instruction is by counting and when a new instruction is required to be written, it will only be written at the end of field. Once there is one callback on a given scan line, any subsequent ones will not have that delay.

Callbacks may be removed by using vi_remove() or by setting the fifth parameter to vi_add() to be some non zero value. In this latter case, vi_remove() will be called automatically once the function has been called that many times.

In order to identify multiple scan line interrupt instructions, the video interrupt maintains an interrupt of its own on the last actual displayed line. This is used for synchronising when to write new instructions. Applications can attach callbacks to this instruction themselves by using the ve_screenend entry of the video environment structure minus 2 as the line number input to vs_update(). This structure entry is not valid until after the first call to vs_update() of an application. Here is a code fragment which shows this :-

```
VS *vsScreen;

vi_add( vsScreen->vs_videnv, vsScreen->vs_videnv->ve_screenend-2, ....
```

The vs_update() function uses two callbacks for its scan synchronisation, one on the last scan line and one on the second. These callbacks time out and remove themselves if another vs_update() is not called soon after the first. To use the scan line interrupt on the second line of the screen, the line input to vi_add() should be the ve_screenstart entry of the video environment plus 2. As with ve_screenend, this structure entry is not valid until after an applications first call to vs_update().

11. Video Synchronisation Example - Scrolling a Large Image

A good example where video synchronisation is very important is scrolling images around the screen using the hardware facilities. Complete source code for an example program do this is in appendix 2. The basic initialisation code is the same as the other examples. This example function will scroll a full screen image horizontally, vertically or diagonally.

11.1 Basic Theory

When vs_update() is asked to display a PICTURE whose line length is different from the hardware line length of the CD-I player the application is running on, it automatically puts a load display start address instruction in the LCT for each scan line where that PICTURE is to be displayed. Such PICTUREs can be scrolled by replacing these with load display start address instructions pointing to other addresses within the PICTURE.

The writing of a full screen height column of LCT instructions takes a finite time, whilst this could be done synchronised to the vertical retrace, in order to keep the example simple it will be done using double buffered LCTs. This example uses the alternate set of LCTs created by vs_open() when called with the LCT_DOUBLE bit set.

To get a smooth scroll requires a very even scroll speed. An integer number of pixels per field interval will certainly give this. A half integer number of pixels/field may be smooth enough, a decision on this is left to the user. This imposes many restrictions if a scroll is to run at the same apparent speed on both NTSC and PAL players. In this example, the interface forces the scroll speed to be a multiple of 150 pixels per second which becomes 3 pixels per field on a PAL player and 2.5 pixels per field on an NTSC player. If this is not smooth enough then by using only even numbers as input, the scroll speed will be 6 pixels/second on PAL and 5 pixels/second on NTSC.

Unlike the Balboa video effects library, this example will not work with a split screen and will not work with a high resolution or interlaced display. Also it will only work with clut images. For a DYUV image much more work is required, see PIMA technical note #34 "DYUV Panning Algorithms" for more information on this.

The example function is split into 5 sections, my_scroll(), my_scroll2(), my_scroll3(), my_scroll_cb() and my_scroll_done(). The first 3 of these handle the setup of the effect, including starting a video interrupt using vi_add(). This video interrupt calls my_scroll_cb() once per field while the effect is running. At the end of the effect, my_scroll_cb() puts a call to my_scroll_done() into the Balboa dispatcher using dispatch_function().

11.2 Start-up and Inputs

The scroll function, my_scroll() takes 6 parameters, the first of these is the PICTURE to scroll, the next two are the x and y direction scroll speeds, these are followed by the duration of the effect and then a callback and parameter for when the effect is finished. The first action of the function when it starts is to preserve 3 of these in global variables for later use.

```
PICTURE  *working;             /* copy of input parameter             */
int      (*a_callback)();      /* callback when scroll completed      */
int      a_parameter;          /* parameter for callback              */
```

```
void my_scroll( picture, x_velocity, y_velocity, duration, callback, parameter )
PICTURE *picture;
int x_velocity,y_velocity,duration;
int (*callback)();
int parameter;
{
        /* save our input into global variables so they are available
        throughout the effect */

        working = picture;
        a_callback = callback;
        a_parameter = parameter;
```

11.3 Timing

The timing and positioning of the scroll is controlled by 4 global variables, the x and y positions and increments per field. In order to handle fractional pixel increments per field, the first four of these are stored as binary fractions with 8 fractional bits below the binary point.

```
int     line_in_picture;        /* current y position of scroll          */
int     x_position;             /* current x position of scroll          */
int     x_increment;            /* x increment per field                 */
int     y_increment;            /* y increment per field                 */
```

The second action within the main my_scroll() function is to translate the scroll speed parameters to the function from pixels per second into the increments in pixels per field, taking account of the video display type of the player in question. By changing the way these are calculated, less restrictive forms of scrolling are possible at the risk of not being smooth.

```
        if( vm_display_type == DISPLAY_625 )
        {
              /* for PAL, multiply by 768 = ( 3 pixels per field multiplied by
              256 for into binary fraction format ) */

              x_increment = x_velocity * 768 ;
              y_increment = y_velocity * 768 ;
        }
        else
        {
              /* for NTSC, multiply by 640 = ( 2.5 pixels per field multiplied by
              256 for binary fraction format ) */

              x_increment = x_velocity * 640 ;
              y_increment = y_velocity * 640 ;
        }
```

The other remaining input is the duration of the effect which is supplied already in the binary fraction format. This gets converted from that format into the number of video fields in that time which is then used as a counter to determine when the effect finishes.

```
        /* convert the duration from 256ths of a second into number of
        fields taking into account the display frequency of the player */

        number_of_fields = ( duration * vm_display_freq ) >> 8;
```

11.4 Video Setup and Initialisation

As discussed previously, this example makes use of double buffered LCTs by using the alternate set from a video screen. In order that the only change as the image scrolls is the position of the image, the effect must start by making the two sets of LCTs identical. This could be done by reading
in one set and writing it out again, but the simplest way to do it is to call vs_update() once for each set of LCTs.

To properly scan synchronise vs_update() requires the use of vs_update_done() to specify a callback to be executed when the results of vs_update() are displayed. This requires the main setup routine to be split into 3 sections, the first two finishing with a vs_update() and using vs_update_done() to invoke the next section.

After each vs_update(), a function find_which_lcts_to_use() is used to find the various lcts to write to and to link to. This information is stored in 3 arrays with those lcts which Balboa thinks are active stored as array element zero and the inactive ones as stored as element one. The array element in which to store the lct information is the first parameter to find_which_lcts_to_use().

```
        /* do the first vs_update to get the first set of LCTs correctly setup */

        vs_update_done( vsScreen, my_scroll2, 0, DISPATCHED);
        vs_update( vsScreen, 0, DISPLAY_625,NULL,NULL);
}
void my_scroll2()
{
        /* extract the various ids and things from the first vs_update(),
        these will become the alternate set of LCTs and hence go into element 1 */

        find_lcts_to_use( 1, working, vsScreen );

        /* now do the second vs_update to get the other set of lcts setup */

        vs_update_done( vsScreen, my_scroll3, 0, DISPATCHED);
        vs_update( vsScreen, 0, DISPLAY_625,NULL,NULL);
}

void my_scroll3()
{
        /* extract the ids and things from the second vs_update
        these are the active set of LCTs and hence go into element zero */

        find_lcts_to_use( 0, working, vsScreen );
```

The three arrays which find_lcts_to_use() stores information in contain the LCT structure pointer which the load display start address instructions are going to be written to, the line within that LCT to start writing at and the CDRTOS LCT id to switch to in order to make the change take effect.

```
LCT      *lcts[2];              /* the two LCTs we are working with       */
int      lctlines[2];           /* the lines to start writing at          */
int      lctids[2];             /* the LCT ids to link to                 */
```

This function also finds which display segment the scroll is to happen within, the height in scan lines of that display segment (which is the number of scan lines to write into the LCT), and the line within the PICTURE displayed on the first line of the display segment.

```
int     number_of_lines_to_write;/* number of lines to write to lct */

find_lcts_to_use( index, picture, vsScreen )
int index;
PICTURE *picture;
VS *vsScreen;
{
```

The display segment within which the effect is to run is found by looping over all the display segments looking for the first one to contain the picture to scroll. To avoid bus traps, if none is found then an error is reported to the status manager and the function gives up and returns.

```
         DSEG *ds;

         /* loop over all the display segments looking for the first one
         which includes our picture */

         for( ds= vsScreen->vs_dseg; ds; ds= ds->ds_next)
         {
               if( picture->pi_plane )
               {
                     if( ds->ds_pic1 == picture ) break;
               }
               else  if( ds->ds_pic0 == picture ) break;
         }

         /* check we actually found something */

         if( ! ds )
         {
               STATUS(ST_APPL,ST_ERROR,0,"no display segment found for scroll");
               return;
         }
```

The LCT structure pointer and line within LCT at which to start writing are taken from this display segment structure according to the which video plane the PICTURE to scroll is in.

```
         /* save the LCT pointer and line */

         if( picture->pi_plane )
         {
               lcts [index] = ds->ds_lct1;
               lctlines [index]= ds->ds_line1;
         }
         else
         {
               lcts [index] = ds->ds_lct0;
               lctlines [index]= ds->ds_line0;
         }
```

This example switches between LCTs using the CDRTOS call dc_flnk(). The inputs to this are an LCT id and an FCT id. The LCT ids are found using the vs_lcts[] entry in the video screen. This contains the 4 LCT structure pointers for a double buffered LCT, stored in alternating order, plane A, plane B, plane A, plane B. The VSF_WHICH_LCT bit of the vs_flags structure entry indicates which set of pointers are currently used.

```
      /* get the LCT id to use for dc_flnk() */

      if( vsScreen->vs_flags & VSF_WHICH_LCT )
          lctids[index]=lc_idof(vsScreen->vs_lcts [2+picture->pi_plane]);
      else  lctids[index]=lc_idof(vsScreen->vs_lcts [picture->pi_plane]);
```

The FCT id for the dc_flnk() call are found outside of find_which_lcts_to_use() from the ve_fct_a or ve_fct_b entries in the video environment structure.

```
int   fctid;                /* the fct id to link from              */

      if( working->pi_plane ) fctid= vsScreen->vs_videnv->ve_fct_b;
      else                    fctid= vsScreen->vs_videnv->ve_fct_a;
```

Also from the display segment comes the number of lines of LCT instructions to write. Since this example only concerns itself with normal resolution video screens, the number of lines to write is just the number of scan lines in the display segment which is the ds_nlines entry in the display segment structure divided by two to convert from UCM coordinates to pixel coordinates.

```
      number_of_lines_to_write = ds->ds_nlines >>1 ;
```

The line within the picture displayed at the top of the display segment can be found with a simple subtraction. That will give a number in UCM coordinates, to obtain one in scan lines we should divide by two and then multiply by 256 to get the number into our binary fraction format. The two of these together are equivalent to a shift up by 7 places.

```
      line_in_picture = ( ds->ds_screenline - picture->pi_line ) << 7;
}
```

One of the position variables has not been discussed, this is x_position, the initial x position of the image on the screen. This is taken from the pi_viewport entry in the PICTURE structure and then converted from UCM coordinates to our binary fraction format using a shift up by 7 places.

```
      x_position = working->pi_viewport.ul.x << 7;
```

11.5 Temporary Workspace

The fastest way to write the load display start address instructions to the LCT is to write them all at once using the CDRTOS call dc_pwrlct(). To do this a buffer will be needed to hold the instructions prior to writing. For a full screen image on a PAL player, this buffer will need to be at least 280 scan lines * 4 bytes = 1120 bytes. Rather than allocating some memory and then de-allocating it, this example uses the video manager internal buffer manager mentioned in the discussion of vs_init(). The buffers created by vs_init() are 2240 bytes in size which is twice what this example needs. The buffer is reserved in my_scroll3() using buff_get().

```
      buffer = (int *) buff_get (vs_large) ;
```

And released in my_scroll_done() using buff_return().

```
    buff_return (vs_large, buffer);
```

11.6 Video Synchronisation

This example uses the Balboa video interrupt manager as its basic source of timing information. A video interrupt callback is setup using vi_add() during my_scroll(). This callback calls the step function my_scroll_cb() once per field until the end of the effect when it is removed using vi_remove() from within my_scroll_done().

```
VIDEO_INT*vidint;              /* video interrupt for timing            */

         /* in my_scroll() */

         vidint= vi_add( vsScreen->vs_videnv,
               vsScreen->vs_videnv->ve_screenstart+2,
               PLANE_A,0,0,my_scroll_cb,0,DISPATCHED);

         /* in my_scroll_done() */

         vi_remove( working->pi_vs->vs_videnv, vidint);
```

As described earlier, this example uses two calls to vs_update() to force the current and alternate set of LCTs of the video screen to be the same. For this reason, the call to vi_add() must be in my_scroll() before the two vs_update() calls so that the video interrupt instruction is also copied into both LCTs. This will result in callbacks to my_scroll_cb() being generated before the rest of the effect is setup. To avoid the effect starting early, a global variable ’running’ is used to control when the effect actually starts. This is set to false before the call to vi_add(), to true at the end of my_scroll3() when the effect is ready to start, and to false when the effect is finished.

```
int      running;              /* flag - effect running true/false      */
```

11.7 The Step Function - my_scroll_cb()

As described just above, a global variable ’running’ is used to control whether the effect is ready to start. The first action in my_scroll_cb() is to check if this variable is false and if that is the case to return without doing anything.

```
void my_scroll_cb()
{
         /* is the video effect actually running at the moment. if not
         then do nothing */

         if( ! running ) return;
```

The next code in the step function handles the termination of the effect, this is discussed separately in section 11.8 later in this document. Following that, the variables which control the display and motion of the image are incremented ready for the next step. In addition to the ’line_in_picture’ and ’x_position’ variables discussed previously, there is a new variable involved here, ’current_lct’. This variable indicates which set of LCTs are being used for the display, it is used as an index into the

3 arrays lcts[], lctlines[] and lctids[] setup by find_which_lct_to_use().

```
int      current_lct;            /* flag to chose which LCT to use         */

         /* switch the flag to a new LCT */

         current_lct ^= 1;

         /* apply the increment to the position variables */

         line_in_picture += y_increment;
         x_position += x_increment;
```

In preparation for the for loop which will generate the load display start address instructions, the variables which define the position of the image must be converted from our binary fraction format into pixels ready to use in addresses. The x position becomes a variable called pixel_x_offset and the line_in_picture is used as an offset into the line start table of the PICTURE, held in the pi_lstart entry in the PICTURE structure.

```
         register int line_counter,pixel_x_offset;
         register int *lstart;

         pixel_x_offset = x_position >> 8;
         lstart = (int*)( working->pi_lstart) + (line_in_picture>>8);
```

Now we have all the inputs for the loop to create the instructions. As discussed previously, the instructions are written into the buffer obtained by buff_get(). The loop to generate the instructions uses the cp_dadr() macro from the CDRTOS header file ucm.h to translate an address into the load display start address LCT instruction for that address.

```
         register int *bufptr;

         bufptr = buffer;
         for(line_counter=number_of_lines_to_write;line_counter;line_counter--)
         {
               /* build the instruction */

               *(bufptr++) = cp_dadr( (*(lstart++) + pixel_x_offset) );
         }
```

The CDRTOS call dc_pwrlct() is used to write the instructions just generated into the LCT. For normal resolution LCTs, this is a lot faster than the more conventional dc_wrlct(). Most of the parameters required for this call were found by find_which_lcts_to_use(). For the column number parameter, DSEG_INS_DADR must be used since that is the column number which vs_update() will have written its load display start address instructions to.

```
         dc_pwrlct(vm_vidpath,lc_idof(lcts[current_lct]),lctlines[current_lct],
               DSEG_INS_DADR,number_of_lines_to_write,1,buffer);
```

The final requirement for each step is to switch the display to the LCTs which have just been written to. The example does this using the CDRTOS dc_flnk() call.

```
         dc_flnk(vm_vidpath,fctid,lctids[current_lct],0);
```

11.8 Terminating the Video Effect

The global variable number_of_fields was setup before the scroll started to be the number of field periods for which the effect was to run. The second group of code within the step function decrements this variable and tests whether the end of the effect has been reached.

```
/* are we finished ? */

if( ! number_of_fields-- )
{
```

For correct termination, the scroll function must clean up after itself so that the contents of the Balboa data structures match the reality of the CD-I video display. For the LCTs, there is a problem if the effect has an odd number of fields. In this case the active LCTs at the end of the effect will be the ones which Balboa thinks are inactive. The global variable 'current_lct' shows which set of LCTs are in use and will be set to one if this situation has happened. The step function checks this variable to decide how to terminate.

```
        /* are we using the right LCTs or not ? */

        if( current_lct )
        {
```

Making Balboa and reality match could be done using a call to dc_flnk() at this point in the step function however it would still be necessary to wait one field period before that change took effect. Rather than writing special code to handle that condition, the simplest option is to perform one extra step in the effect. In order to avoid this step making the displayed image move, the position increments are subtracted from the position variables so that when those increments are added back in later in my_scroll_cb() the net change is zero. Since we are adding an extra step, the number_of_fields variable is be incremented to match this.

```
                line_in_picture -= y_increment;
                x_position -= x_increment;

                number_of_fields++;
        }
```

If the active LCT is the one which Balboa expects then the effect is really finished, so the step function sets the 'running' variable to false to make sure that no more steps happen and then puts a call to the effect termination routine, my_scroll_done() into the Balboa dispatcher using dispatch_function().

```
        else
        {
                /* end of scroll so clear running flag and call the
                termination routine */

                running = 0;
                dispatch_function(0,my_scroll_done,0);
                return;
        }
```

```
      }
```

The function my_scroll_done() performs the non time critical aspects of the termination of the effect. There are two parts to this, first is adjusting the PICTURE structure to match the section of the image which is now being displayed. What is needed is to move the viewport of the PICTURE the same distance as the image moved during the scroll. This distance is computed during the setup for the effect and held in two global variables, total_x and total_y. The division is by 128 rather than 256 since the inputs to pi_viewport() are in UCM coordinates not pixel ones. The last parameter to pi_viewport defines where on the screen the top of the viewport should appear. In this case we have scrolled the image within that portion of the screen it contributed to before and so we need the top line of the viewport to be displayed in the same place as it was before.

```
int      total_x;                /* total x displacement in effect         */
int      total_y;                /* total y displacement in effect         */

         total_x = (x_increment * number_of_fields ) >> 7;
         total_y = (y_increment * number_of_fields ) >> 7;

void my_scroll_done()
{
         /* set the viewport of the picture to match where we now are on
         the screen */

         pi_viewport( working,
               working->pi_viewport.ul.x + total_x,
               working->pi_viewport.ul.y + total_y,
               working->pi_viewport.lr.x + total_x,
               working->pi_viewport.lr.y + total_y,
               working->pi_viewport.ul.y + working->pi_line );
```

The very last action of the effect is to generate the callback to the application that the effect is finished. The address of this function was saved in the global variable a_callback when the effect started.

```
         /* generate the callback to the rest of the application */

         if( a_callback )
         {
               (*a_callback)(0,a_parameter);
         }
}
```

11.9 The Demonstration Program

The example program listed in Appendix 2 shows vertical, horizontal and diagonal scrolls at a variety of speeds. For each speed upto the maximum defined by the MAXIMUM_SPEED symbol, it does 6 scrolls (defined by the NUMBER_OF_TESTS symbol). For each test, the velocities are defined by the contents of the x_directions and y_directions arrays. The first test is from the top left of the image to the top right (i.e. increasing x), then top right to bottom right followed by bottom right towards the top left at a 45 degree angle. The last 3 tests are the reverse of the first 3 to take the image exactly back to where it started.

```
int test_number = -1;           /* the current test number              */
int multiplier = 2;             /* speed multiplier                     */
#define NUMBER_OF_TESTS 6       /* number of tests for each speed       */
#define MAXIMUM_SPEED 10        /* maximum speed multiplier             */
#define SCROLL_VELOCITY         150   /* scroll base velocity is 150 pixels per
                                      second */
int x_directions[NUMBER_OF_TESTS] = { 1, 0, -1, 1, 0, -1 };
int y_directions[NUMBER_OF_TESTS] = { 0, 1, -1, 1, -1, 0 };
int distances[NUMBER_OF_TESTS] = { 384, 280, 280, 280, 280, 384 };

void do_scroll()
{
         int duration;

         /* setup for the next test */

         if( ++test_number == NUMBER_OF_TESTS )
         {
               /* increment the velocity multiplier */

               multiplier += 2;

               /* stop when we reach the maximum velocity */

               if( multiplier == MAXIMUM_SPEED ) dispatch_quit(0);

               /* back to the first test for this new velocity */

               test_number =0 ;
         }
         /* now work out how many steps we can do so that we don't run off the end.
         The base scroll velocity is 150 pixels/second regardless of PAL or NTSC */

         duration=(distances[test_number] <<8) / ( multiplier * SCROLL_VELOCITY );

         /* in Balboa 1.3, we need to sleep for one field between the end of one
         effect and the start of the next one. This should be fixed for 1.4 */

         sleep(3);

         /* now start the effect */

         my_scroll(picture,multiplier*x_directions[test_number],
               multiplier *y_directions[test_number],duration,do_scroll,0);
}
```

Appendix 1. Listing of DYUV Movie Playback Program - Example 3

```
/*
        playmovie.c - example code to display a DYUV movie from a real time
        file

        Jon Piesing PRL Redhill

*/

/* NON-BALBOA include files */

#include <errno.h>
#include <modes.h>
#include <stdio.h>

/* BALBOA include files */

#include <vm_vs.h>
#include <vm_pic.h>
#include <bp_mem.h>
#include <pm_low.h>
#include <status.h>
#include <vm_video.h>
#include <vm_defs.h>

/* define the various parameters for this movie */

#define DYUV_MOVIE         "RTF/dyuv.movie"

#define PAL_WIDTH          384   /* PAL picture's width in pixels */
#define PAL_HEIGHT         280   /* PAL picture's height in scan lines */
#define PAL_BYTES          PAL_WIDTH * PAL_HEIGHT

#define MOVIE_WIDTH        160   /* width of movie in pixels     */
#define MOVIE_HEIGHT       100   /* height of movie in scan lines    */
#define MOVIE_X            100   /* movie x position ( UCM coordinates ) */
#define MOVIE_Y            100   /* movie y position ( UCM coordinates ) */
#define MOVIE_BYTES        MOVIE_WIDTH * MOVIE_HEIGHT

#define MAX_INBUF          3     /* max. # of input buffers */
#define MAX_OUTBUF         3     /* Max # of display buffers */
#define MAX_INTERNAL_BUF   1     /* Max # of internal buffers */
#define MAX_DSEG           5     /* Max # of DSEG's for picture build up */
#define ARBITRARY_SIZE     4096  /* any conveniently big size */

#define F2_BYTES           2324  /* form 2 sector size */
#define F2_SECTORS(s)      ((F2_BYTES - ((s) % F2_BYTES)) + (s)) / F2_BYTES

#define NA                 0     /* Not Applicable */

#define UCM(x)             (x<<1)/* Convert to UCM */
#define CHANNEL(c)         c     /* Trivial, for readability only */

/* DYUV start values */
```

```
#define Y_START      0x80
#define U_START      0x80
#define V_START      0x80

/* now the global variables */

PICTURE *piMain;                /* the full screen picture to be used*/
                                /* ask background for the movie*/
PICTURE *piPartial[MAX_OUTBUF]; /* the small pictures to copy the*/
                                /* frames into for display*/
VS       *vsScreen;             /* the video screen*/
char     *InBuf[MAX_INBUF];     /* buffers to hold the frames from CD*/
int      File;                  /* the OS9 file as returned by open()*/
int      Pending = 0;           /* flag to say whether a frame is*/
                                /* waiting to be decoded*/
char     *Source;               /* the source data for the next frame*/
                                /* to be decoded*/
int      Output = 1;            /* the next output picture to use*/
DSEG     *dsPlaying;            /* the display segment in which the*/
                                /* movie is playing */
SIZE_RECT   srMovieWin =        /* Size and position of movie window */
{
    {
         MOVIE_X, MOVIE_Y       /* ul.x, ul.y */
    },
    UCM(MOVIE_WIDTH),           /* width    */
    UCM(MOVIE_HEIGHT)           /* height   */
};

/* now the functions in the order in which they will be called */

main()
{
         void runit();
         dispatch_loop( runit, NULL, NULL );
}

void runit()
{
         int     x;
         void    movieCB(),play_done(),background();
         char    *buffer;
         int     vs_finish();

         dispatch_atquit( vs_finish, NA);

         /* play manager initialisations */

         sgm_init();   /* initialise signal manager, needed for play manager */

         buffer = (char*) bp_allocate( ARBITRARY_SIZE, BP_MEM_PLANEA );
         pmb_set_block( buffer, ARBITRARY_SIZE);
         pml_init();
         File= open( DYUV_MOVIE, S_IREAD);

         /* video manager initialisations */
```

```
        vs_init (BP_MEM_DONTCARE, MAX_INTERNAL_BUF);
        vsScreen = vs_open( LCT_DOUBLE|CURSOR_ON, MAX_DSEG, BP_MEM_PLANEA );

        /* create the full screen picture and the smaller ones */


        piMain = pi_create( PLANE_A, D_DYUV, UCM(PAL_WIDTH), UCM(PAL_HEIGHT),
              F2_SECTORS(PAL_BYTES) * F2_BYTES, NA);

        for( x = 0; x < MAX_OUTBUF; x++ )
        {
              piPartial[x]=pi_create(PLANE_B, D_DYUV, UCM(PAL_WIDTH),
                                    UCM(MOVIE_HEIGHT), NA, NA);
              piPartial[x]->pi_dstart = cp_yuv( PB, Y_START,U_START,V_START);
              bp_memset( piPartial[x]->pi_pstart, 0, piPartial[x]->pi_size);
        }

        /* setup the display */

        pi_settrans( piMain, TC_MF0TRUE);
        pi_front( vsScreen, piMain );
        pi_back( vsScreen, piPartial[0]);
        pi_position( piPartial[0], srMovieWin.ul.y);
        vs_update( vsScreen, NON_INTERLACE, DISPLAY_625, NULL,NULL );

        /* get the address of the display segment in which the movie is playing */

        dsPlaying = vs_find_segment (vsScreen, srMovieWin.ul.y);

        /* allocate the memory for the input buffers and make them known to the
        play manager */

        for(x=0; x < MAX_INBUF; x++)
        {
              InBuf[x]=(char*)bp_allocate(F2_SECTORS(MOVIE_BYTES) * F2_BYTES,
                    BP_MEM_DONTCARE);
              pml_add_buffer (CHANNEL(1), VIDEO_TYPE, F2_SECTORS(MOVIE_BYTES),
                    InBuf[x], movieCB, NA, DISPATCHED);
        }
        /* make the background picture known to the play manager */

        pml_add_buffer(CHANNEL(0), VIDEO_TYPE,F2_SECTORS(PAL_BYTES),
                       piMain->pi_pstart, background, NA, DISPATCHED);

        /* now go and do the play */

        pml_play( File, 0, /* position */
                    3, /* Potentially active mask: channel 0 & 1 */
                    0, /* direct_audio_mask */
                    1, /* nr. of EOR to mark End of Play */
                    0, /* channels to be switched between active/de-active */
                    play_done, NULL, NULL );
}


/* the callback when the background is loaded */
```

```
void background(context,buffer)
int context;
PML_BUFFER *buffer;
{
         /* show the image by showing the video screen now */

         vs_show( vsScreen );

         /* put up the matte for this movie. We will use plane B for the LCT
         instructions since that normally has more space in it. This has
         no effect on the results of the matte */


         make_rect_matte (vsScreen, PLANE_B, &srMovieWin);
}
/* create a rectangular matte */

make_rect_matte( vs, plane, mw )
VS           *vs;
int          plane;
SIZE_RECT   *mw;
{
         DSEG    *ds;
         LCT     *lct;
         short   top_column, top_line, btm_column, btm_line;
         int     instructions[2];
         int     left_x = mw->ul.x,
         top_y   = mw->ul.y,
         right_x = left_x + mw->width,
         bottom_y= top_y + mw->height;

         /* decide which LCT we need to write to and where */

         ds = vs_find_segment( vs, top_y );
         if( plane == PLANE_B )
         {
               lct = ds->ds_lct1;
               top_line = top_y    - ds->ds_screenline + ds->ds_line1;
               btm_line = bottom_y - ds->ds_screenline + ds->ds_line1;
         }
         else
         {
               lct = ds->ds_lct0;
               top_line = top_y    - ds->ds_screenline + ds->ds_line0;
               btm_line = bottom_y - ds->ds_screenline + ds->ds_line0;
         }

         /* get two spaces on the first line of our rectangle */

         top_column = lc_request( lct, top_line, 1, 2); /* 1 line, 2 columns */

         /* check that we did get what we asked for */

         if( top_column==-1)
         {
               STATUS(ST_APPL,ST_ERROR,errno,"ran out of LCT space !");
               return -1;
```

```
        }
        /* build the instructions we want to write */

        instructions[0]=
        cp_matte( 0,      /* matte register zero */
                MO_SET, /* set matte flag to true */
                0,      /* use matte flag zero */
                0x3f,   /* the image contribution factor, not used here */
                left_x);/* the left hand side of the rectangle */

        instructions[1]=cp_matte(1, MO_RES, 0, 0x3f, right_x);

        /* now do the same for the single instruction at the end. This should
        be on the first scan line which is not required to be transparent,
        hence the +2 on the next line */

        btm_column = lc_request( lct, btm_line,1,1); /* 1 line, 1 column */
        if( btm_column == -1 )
        {
            STATUS(ST_APPL, ST_ERROR, errno, "ran out of LCT space !");
            return -1;
        }

        /* write the instruction to turn off mattes */

        dc_wrli(vm_vidpath,lc_idof(lct),btm_line,btm_column,cp_matte(
            0,          /* matte register zero */
            MO_END,     /* end of all matte registers */
            0,          /* matte flag zero */
            0x3f,       /* icf, not used here either */
            0));        /* position doesn't matter, just turn it off */

        /* write the starting instructions last to avoid scan sync problems */

        dc_wrlct( vm_vidpath, lc_idof( lct ), top_line,
            top_column, 1, 2,instructions );

        /* done */

        return(OK);
}

/* the callback for each time a frame comes in from the CD */

void movieCB (context, buffer)
int context;
PML_BUFFER *buffer;
{
        void display_frame();

        /* if we already have a frame pending for decoding then forget about
        this one. If not then we do have one pending now */

        if( Pending )   return;
        Pending ++;

        /* remember where this data was loaded for the decode function */
```

```
         Source = buffer->Buf;

         /* dispatch the decode function */

         dispatch_function( 0,display_frame,0);
}
/* the function which actually copies a frame onto the screen and displays it */

void display_frame()
{
         register    unsigned short *dest;
         register    unsigned short *local_source;
         register             int    x, y;
         unsigned int    *linestart;

         /* sort out the source and destination. This copy function is slightly
         optimised and does 2 bytes at a time, hence we have to cast the source
         pointer to be a pointer to unsigned short */

         local_source = (unsigned short*) Source;*/
         linestart = (unsigned int*)( piPartial[ Output ] -> pi_lstart);

         /* loop copying */

         for( y = 0; y < MOVIE_HEIGHT; y++)
         {
               /* sort out where we are copying to */

               dest = (unsigned short*)( (*(linestart++)) + MOVIE_X / 2 );
               for (x = 0; x < MOVIE_WIDTH / 2; x++)   /* /2 => short vs. char */
                     *(dest++) = *(local_source++);
         }

         /* switch to displaying the buffer we have just decoded into
         by writing a new display start address instruction into the first
         line of the LCT for the display segment in which the movie is playing */

         dc_wrli( vm_vidpath, lc_idof( dsPlaying -> ds_lct1), dsPlaying->ds_line1,
               DSEG_INS_DADR, cp_dadr((int)(piPartial[Output]->pi_pstart)) );

         /* increment the output buffer counter, looping back to zero when
         we have used all the buffers */

         if( ++ Output == MAX_OUTBUF ) Output = 0;

         /* clear the frame pending flag so we display another one */

         Pending = 0;
}

/* the function called when the play is finished */

void play_done()
{
    dispatch_quit(0);
}
```

Appendix 2. Listing of Scroll Program - Example 4

```
/*
        scroll.c - simple demonstration program of scrolling

        Jon Piesing PRL
*/

/* NON-BALBOA include files */

#include <errno.h>
#include <modes.h>
#include <stdio.h>

/* BALBOA include files */
#define     BP_DEBUG

#include <vm_vs.h>
#include <vm_pic.h>
#include <bp_mem.h>
#include <pm_low.h>
#include <status.h>
#include <vm_video.h>
#include <vm_defs.h>
#include <vm_buff.h>

/* define the various parameters for this program */

#define IMAGE_FILE   "example.rtf"

#define IMAGE_WIDTH             768   /* width in pixels of test image */
#define IMAGE_HEIGHT            576   /* height in scan lines of test image */
#define IMAGE_BYTES  IMAGE_WIDTH * IMAGE_HEIGHT

#define MAX_INTERNAL_BUF        1     /* Max # of internal buffers */
#define MAX_DSEG                5     /* Max # of DSEG's for picture build up */
#define MAX_TCMAP               0     /* Max # of TCMAPs for clut manager */
#define MAX_INTERRUPT_LINES     4     /* Max # of scan line interrupts */
#define MAX_INTERRUPT_FUNCS     4     /* Max # of video interrupt callbacks */

#define ARBITRARY_SIZE          4096  /* any conveniently big size */

#define F2_BYTES                2324  /* form 2 sector size */
#define F2_SECTORS(s)           ((F2_BYTES - ((s) % F2_BYTES)) + (s)) / F2_BYTES

#define NA                      0     /* Not Applicable */

#define UCM(x)                  (x<<1)      /* Convert to UCM */
#define CHANNEL(c)              c     /* Trivial, for readability only */

#define SCROLL_VELOCITY         150   /* scroll base velocity is 150 pixels per
                                      second */
```

```
/* the global variables */

PICTURE  *picture;               /* the full screen picture to be used     */
                                 /* ask background for the movie           */
VS       *vsScreen;              /* the video screen                       */
int      File;                   /* the OS9 file as returned by open()     */
char     plte_buffer[F2_BYTES];  /* a buffer to hold the clut data         */

/* the inputs for the scroll */

LCT      *lcts[2];               /* the two LCTs we are working with       */
int      lctlines[2];            /* the lines to start writing at          */
int      line_in_picture;        /* current y position of scroll           */
int      x_position;             /* current x position of scroll           */
int      number_of_lines_to_write;/* number of lines to write to lct       */

/* the inputs for dc_flnk */

int      fctid;                  /* the fct id to link from                */
int      lctids[2];              /* the LCT ids to link to                 */

/* copies of our input variables */

PICTURE  *working;               /* copy of input parameter                */
int      (*a_callback)();        /* callback when scroll completed         */
int      a_parameter;            /* parameter for callback                 */

/* timing and control variables */

int      x_increment;            /* x increment per field                  */
int      y_increment,number_of_fields;/* y increment per field             */
int      current_lct;            /* flag to chose which LCT to use         */
VIDEO_INT*vidint;                /* video interrupt for timing             */
int      running;                /* flag - effect running true/false       */
int      *buffer;                /* buffer to build lct instructions in    */
int      total_x;                /* total x displacement in effect         */
int      total_y;                /* total y displacement in effect         */

/* function prototypes */

void runit(),play_done(),dispatch_quit(),do_scroll(),my_scroll();
void my_scroll_cb(),my_scroll_done(),my_scroll2(),my_scroll3(),vs_finish();

main()
{
         dispatch_loop( runit, NULL, NULL );
}
void runit()
{
         char *buffer;

         /* play manager stuff */

        STATUS_INIT(stderr,ST_MGR_ALL,ST_TYP_ALL,0);
         sgm_init();/* initialise signal manager, needed for play manager */

         buffer = (char*) bp_allocate( ARBITRARY_SIZE, BP_MEM_PLANEA );
         pmb_set_block( buffer, ARBITRARY_SIZE);
         pml_init();
         File=open(IMAGE_FILE,S_IREAD);

         /* various testing stuff */

         sgm_enable(2,dispatch_quit,2,DISPATCHED);
         sgm_enable(3,dispatch_quit,3,IMMEDIATE);
         dispatch_atquit( vs_finish,0);

         /* video manager initialisations */

         vs_init (BP_MEM_DONTCARE, MAX_INTERNAL_BUF);
         vsScreen = vs_open( LCT_DOUBLE|CURSOR_ON, MAX_DSEG, BP_MEM_PLANEA );
         cl_init( vsScreen, BP_MEM_DONTCARE,MAX_TCMAP );
         vi_init( vsScreen->vs_videnv,BP_MEM_DONTCARE,MAX_INTERRUPT_LINES,
               MAX_INTERRUPT_FUNCS);

         /* create a 4 times full screen size picture to scroll around in */

         picture = pi_create( PLANE_A, D_CLUT8, UCM(IMAGE_WIDTH),
               UCM(IMAGE_HEIGHT),F2_SECTORS(IMAGE_BYTES) * F2_BYTES, NA);

         /* play the file */

         pml_add_buffer(CHANNEL(0), VIDEO_TYPE,F2_SECTORS(IMAGE_BYTES),
               picture->pi_pstart, NULL, NA, DISPATCHED);
         pml_add_buffer(CHANNEL(0), DATA_TYPE,1,plte_buffer,NULL,0,DISPATCHED);

         /* start the play */

         pml_play( File, 0, /* position */
                     1, /* Potentially active mask: channel 0 */
                     0, /* direct_audio_mask */
                     1, /* nr. of EOR to mark End of Play */
                     0, /* channels to be switched between active/de-active */
                     play_done, NULL, NULL );
}

void play_done()
{
         /* now we have loaded our image data, display it together with the clut */

         pi_front( vsScreen, picture );
         cl_exec( vsScreen, PLANE_A, ((CLUT*) plte_buffer) );
         vs_update( vsScreen, 0, DISPLAY_625, NULL, NULL );
         vs_show( vsScreen );

         /* start our test routine */

         do_scroll();
}
```

```
/* define our test data */

int test_number = -1;           /* the current test number              */
int multiplier = 2;             /* speed multiplier                     */

#define NUMBER_OF_TESTS 6       /* number of tests for each speed       */
#define MAXIMUM_SPEED 10        /* maximum speed multiplier             */

/* define the directions and durations for each of the tests */

int x_directions[NUMBER_OF_TESTS] = { 1, 0, -1, 1, 0, -1 };
int y_directions[NUMBER_OF_TESTS] = { 0, 1, -1, 1, -1, 0 };
int distances[NUMBER_OF_TESTS] = { 384, 280, 280, 280, 280, 384 };

void do_scroll()
{
         int duration;

         /* setup for the next test */

         if( ++test_number == NUMBER_OF_TESTS )
         {
               /* increment the velocity multiplier */

               multiplier += 2;

               /* stop when we reach the maximum velocity */

               if( multiplier == MAXIMUM_SPEED ) dispatch_quit(0);
               printf( "multiplier = %d\n", multiplier);

               /* back to the first test for this new velocity */

               test_number =0 ;
         }
         /* now work out how many steps we can do so that we don't run off
         the end. The base scroll velocity is 75 pixels/second regardless
         of PAL or NTSC */

         duration=(distances[test_number] <<8) / ( multiplier * SCROLL_VELOCITY );

         /* due to a bug in 1.3, we need to sleep for one field between the
         end of one effect and the start of the next one. This should be fixed
         for 1.4 */

         sleep(3);

         /* now start the effect */

         my_scroll(picture,multiplier*x_directions[test_number],
               multiplier *y_directions[test_number],
               duration,do_scroll,0);
}
```

```
void my_scroll( picture, x_velocity, y_velocity, duration, callback, parameter )
PICTURE *picture;
int x_velocity,y_velocity,duration;
int (*callback)();
int parameter;
{
         /* save our input into global variables so they are available
         throughout the effect */

         working = picture;
         a_callback = callback;
         a_parameter = parameter;

         /* convert the x and y velocities into pixels per field increments,
         stored multiplied by 256 */

         if( vm_display_type == DISPLAY_625 )
         {
               /* for PAL, multiply by 768 = ( 3 pixels per field multiplied by
               256 for the binary fraction format ) */

               x_increment = x_velocity * 768 ;
               y_increment = y_velocity * 768 ;
         }
         else
         {
               /* for NTSC, multiply by 640 = ( 2.5 pixels per field multiplied by
               256 for the binary fraction format ) */

               x_increment = x_velocity * 640 ;
               y_increment = y_velocity * 640 ;
         }

         /* convert the duration from 256ths of a second into number of
         fields taking into account the display frequency of the player */

         number_of_fields = ( duration * vm_display_freq ) >> 8;

         /* make sure the running flag is set to false so we don't start
         any effects until we have finished the setup */

         running = 0;

         /* work out where we will end up */

         total_x = (x_increment * number_of_fields ) >> 7;
         total_y = (y_increment * number_of_fields ) >> 7;

         /* setup the video interrupt we will be using for timing */

         vidint= vi_add( vsScreen->vs_videnv,
               vsScreen->vs_videnv->ve_screenstart+2,
               PLANE_A,0,0,my_scroll_cb,0,DISPATCHED);

         /* do the first vs_update to get the first set of LCTs correctly setup */

         vs_update_done( vsScreen, my_scroll2, 0, DISPATCHED);
```

```
        vs_update( vsScreen, 0, DISPLAY_625,NULL,NULL);
}

void my_scroll2()
{
        /* extract the various ids and things from the first vs_update(),
        these will become the alternate set of LCTs and hence go into element 1 */

        find_lcts_to_use( 1, working, vsScreen );

        /* now do the second vs_update to get the other set of lcts setup */

        vs_update_done( vsScreen, my_scroll3, 0, DISPATCHED);
        vs_update( vsScreen, 0, DISPLAY_625,NULL,NULL);
}

void my_scroll3()
{
        /* extract the ids and things from the second vs_update
        these are the active set of LCTs and hence go into element zero */

        find_lcts_to_use( 0, working, vsScreen );

        /* get a buffer to build our LCT instructions in */

        buffer = (int *) buff_get (vs_large) ;

        /* extract the fct id we are going to link to */

        if( working->pi_plane ) fctid = vsScreen->vs_videnv->ve_fct_b;
        else                    fctid = vsScreen->vs_videnv->ve_fct_a;

        /* the currently displayed LCT is entry 0 in the array */

        current_lct = 0;

        /* get the current starting positions for the horizontal aspect
        of the scroll. The reason this is shifted by 7 and not by 8 is
        to include a /2 to convert from ucm to pixel coordinates */

        x_position = working->pi_viewport.ul.x << 7;

        /* set the 'running' flag to start the callbacks actually doing
        something */

        running = 1;
}
```

```
/* find the LCTs we have to switch between */

find_lcts_to_use( index, picture, vsScreen )
int index;
PICTURE *picture;
VS *vsScreen;
{
         DSEG *ds;

         /* loop over all the display segments looking for the first one
         which includes our picture */

         for( ds = vsScreen->vs_dseg; ds; ds = ds->ds_next)
         {
               if( picture->pi_plane )
               {
                     if( ds->ds_pic1 == picture ) break;
               }
               else  if( ds->ds_pic0 == picture ) break;
         }
         /* check we actually found something */

         if( ! ds )
         {
               STATUS(ST_APPL, ST_ERROR, 0, "no display segment found for scroll");
               return;
         }
         /* save the LCT pointer and line */

         if( picture->pi_plane )
         {
               lcts [index] = ds->ds_lct1;
               lctlines [index] = ds->ds_line1;
         }
         else
         {
               lcts [index] = ds->ds_lct0;
               lctlines [index] = ds->ds_line0;
         }
         /* get the LCT id to use for dc_flnk() */

         if( vsScreen->vs_flags & VSF_WHICH_LCT )
               lctids[index]=lc_idof(vsScreen->vs_lcts [2+picture->pi_plane]);
         else  lctids[index]=lc_idof(vsScreen->vs_lcts [picture->pi_plane]);

         /* get the number of lines from the display segment and convert this
         from UCM coordinates into pixel ones since everything we are working
         with is normal resolution */

         number_of_lines_to_write = ds->ds_nlines >>1 ;

         /* get the starting position for the vertical aspect of the
         scroll. */

         line_in_picture = ( ds->ds_screenline - picture->pi_line ) << 7;
}
```

```
void my_scroll_cb()
{
        register int line_counter,pixel_x_offset;
        register int *bufptr, *lstart;

        /* is the video effect actually running at the moment. if not
        then do nothing */

        if( ! running ) return;

        /* are we finished ? */

        if( ! number_of_fields-- )
        {
              /* are we using the right LCTs or not ? */

              if( current_lct )
              {
                    /* we want to switch to the correct set of LCTs without
                    applying any further position increment. To do this, subtract
                    the position increment from the position variables so that
                    when it is added on later we get to the correct answer */

                    line_in_picture -= y_increment;
                    x_position -= x_increment;

                    /* now add one onto number_of_fields so that the correct
                    termination code gets called at the start of the next field */

                    number_of_fields++;
              }
              else
              {
                    /* end of scroll so clear running flag and call the
                    termination routine */

                    running = 0;
                    dispatch_function(0,my_scroll_done,0);
                    return;
              }
        }

        /* switch our flag to a new LCT */

        current_lct ^= 1;

        /* apply the increment to the position variables */

        line_in_picture += y_increment;
        x_position += x_increment;

        /* convert the new values of the position into pixel values and
        addresses ready to use */

        pixel_x_offset = x_position >> 8;
        lstart = (int*)( working->pi_lstart) + (line_in_picture>>8);
```

```
         /* loop for each line we have to write, building the LCT instructions */

         bufptr = buffer;
         for(line_counter=number_of_lines_to_write;line_counter;line_counter--)
         {
               /* build the instruction */

               *(bufptr++) = cp_dadr( (*(lstart++) + pixel_x_offset) );
         }
         /* now write the instructions we just built */

         dc_pwrlct(vm_vidpath,lc_idof(lcts[current_lct]),lctlines[current_lct],
               DSEG_INS_DADR,number_of_lines_to_write,1,buffer);

         /* switch LCTs via dc_flnk */

         dc_flnk(vm_vidpath,fctid,lctids[current_lct],0);
}

void my_scroll_done()
{
         /* set the viewport of the picture to match where we now are on
         the screen */

         pi_viewport( working,
               working->pi_viewport.ul.x + total_x,
               working->pi_viewport.ul.y + total_y,
               working->pi_viewport.lr.x + total_x,
               working->pi_viewport.lr.y + total_y,
               working->pi_viewport.ul.y + working->pi_line );

         /* remove video interrupts */

         vi_remove( working->pi_vs->vs_videnv, vidint);

         /* release the buffer we have been building LCT instructions in */

         buff_return (vs_large, buffer);

         /* generate the callback to the rest of the application */

         if( a_callback )
         {
               (*a_callback)(0,a_parameter);
         }
}
```

Appendix 3. Master and Green Scripts

Each of the examples in this note uses the same master script but a different green script. Here is the master script :-

```
!
!           master script for Balboa video examples
!
            define album "test" publisher "PRL" preparer "Jon Piesing"
            volume "simple_example" in "simple.cd"
            message from "message.cda"
!           copyright file Copyright from "copyright.txt"
!           abstract file Abstract from "abstract.txt"
!           biblio    file Biblio from "bibliographic.txt"
!           application file frontend from "CMDS/frontend"
            green file fred from <fred.g>
!
{
            "myfile.rtf" from fred
}
```

Here are each of the green scripts. Each should be in a file called "fred.g" in order to match the master script listed above.

Example 1

```
record
            video in channel 0 from
            "/u3/jon/data/IFF/girlski1.cl7">
```

Example 2

```
record
            video in channel 0 from
            "/u3/jon/data/IFF/girlski1.cl7">
            data in channel 0 from
            "/u3/jon/data/IFF/girlski1.cl7">CAT#IMAG>FORM#IMAG>PLTE
```

Example 4

```
record
            video in channel 0 from
            "/u3/jon/data/IFF/girlski1_big.cl8">
            data in channel 0 from
            "/u3/jon/data/IFF/girlski1_big.cl8">CAT#IMAG>FORM#IMAG>PLTE
```

Appendix 4. Introduction to the Balboa Play Manager

One of the methods provided by the CD-I standard for retrieving data from a CD-I disc is by playing a real time file. Balboa includes a play manager to make the interface to this more high level and to fit in with the Balboa philosophy. The Balboa play manager is multi-levelled, simple applications need only concern themselves with the low level play manager and that is what is described here.

The playing of a real time file is an asynchronous activity. Data comes in from the disc at a maximum rate of one sector every 75th of a second. The application must build up lists of structures which tell Balboa and CDRTOS what to do with this data.

Appendix 4.1. Initialisation

The play manager uses its own area of memory for the various internal structures. The function pmb_set_block() is used to set the size and memory bank to be used for this. An example call to this could look like :-

```
char *buffer;
buffer = (char*) bp_allocate( 4096, BP_MEM_PLANEA );
pmb_set_block(buffer,4096);
```

The size of 4096 used here is totally arbitrary. Smaller numbers may work depending on what the application is doing.

There is a second play manager initialisation function, pml_init(). This performs various calls to other Balboa managers needed for the play manager. It must be called after the signal manager has been initialized using sgm_init().

Appendix 4.2. Playback Calls

The basic structure which defines how the data from a real time file is to be used is the PML_BUFFER structure. These are created using the function pml_add_buffer(). Using this function, the application specifies that a certain number of sectors are to be loaded into memory at a certain address from a given channel and data type within the real time file. Applications can also specify a callback to be called when those sectors have been loaded. For the case of loading an image into a Balboa PICTURE structure, this call could look like :-

```
PICTURE *pic;
PML_BUFFER *pml;
void loaded();

        pml = pml_add_buffer( 0,VIDEO_TYPE, 40,pic->pi_pstart,loaded,loaded,DISPATCHED);
```

This example specifies that 40 sectors (parameter #3) of video data (parameter #2) that come in from the CD on channel 0 (parameter #1) are to be put in memory at the address given by pic->pi_pstart (parameter #4). The entry pi_pstart in the picture structure is the start address of the pixel memory. The last three parameters define a callback to be called when this data has been loaded.

When all the PML_BUFFERs for a particular data type and channel have been used up, the play manager will start again with the first buffer specified for that data type and channel. This is very useful when playing content such as DYUV movies. In the example earlier in this document, 3 PML_BUFFER structures are created and then this circular facility is used to handle the other frames within the movie.

Once an application has defined where the data from the real time file is to be put, it can then start the playing of the real time file. The function to do this is pml_play().

```
void play_done();
int file;

        pml_play( file,0,31,1,1,0,play_done,NULL,NULL );
```

The first parameter is the path to the file to play as returned by open(). The third parameter is a bit mask which selects which channels to receive from the real time file. Sectors in channels where the corresponding bit is not set will not come into memory even if the application has defined PML_BUFFER structures for these channels. The value of 31 in the example corresponds to channels 0 - 4 inclusive. If only channels 3 and 4 were required then this value would be 8 + 16 = 24.

The fourth parameter for pml_play() defines which audio channel should be sent directly to the audio decoder without passing through memory. It is a bit mask just organised in the same way as the previous parameter except that only one bit can be set at any one time. In the above example, ADPCM audio in channel 1 is sent direct to the decoder.

The fifth parameter defines the number of real time records to play. This can be used to automatically stop the real time file play at a particular instant in a real time file. Using the master disc building tool, each "record" statement marks the start of a new real time record and if it is in the middle of a script, the end of the previous one. There are also facilities to insert end of record bits (EORs) at specific locations in the real time file, details of these are given in the documentation for master.

The seventh and eighth parameters are a callback which will be called when the playing of the real time file has finished. This can be when the specified number of records has been played or at the end of the file. The address of the function is the seventh parameter, play_done in the example above. The eighth parameter above is a parameter passed to the callback.

The other three parameters of pml_play() which have not been mentioned here are for advanced use and so are not relevant in an introduction. These are the second, sixth and last all of which are zero or NULL in the example above.

When a real time file has finished playing, it is a very good idea to clean out the play buffer lists and play event lists in one operation. This is done using pml_cleanup_all(). An example call to this could look like :-

```
        pml_cleanup_all();
```

Appendix 5. Introduction to the CD-I Matte Mechanism

The CD-I matte mechanism allows applications to change transparency and image contribution factor at arbitrary locations within a scan line. Example code for changing image contribution factor is in the Balboa programmer's Guide Volume 1, pages 6-37 to 6-39. Example code for changing transparency is in section 7.3 on page 21 of this note. Mattes are the only mechanism which can change the CD-I video setup other than during the horizontal retrace period between scan lines. Their Green Book description can be found on pages V-82 to V-85.

The CD-I video hardware contains 8 matte registers. The contents of these are loaded by instructions in the LCT or FCT and persist until either they are re-loaded by another LCT/FCT instruction or until the display scan reaches the end of field. Each of these registers is divided into 4 sections :-

- a 4 bit op-code
- a 1 bit matte flag
- a 6 bit image contribution factor and
- a 10 bit position in UCM coordinates rather than pixels.

The 8 registers can either function as 1 set of 8 or 2 sets of 4, the choice between these two is made by a bit in the load image coding methods LCT/FCT instruction. The Balboa interface to this bit is the function vs_setnmatte().

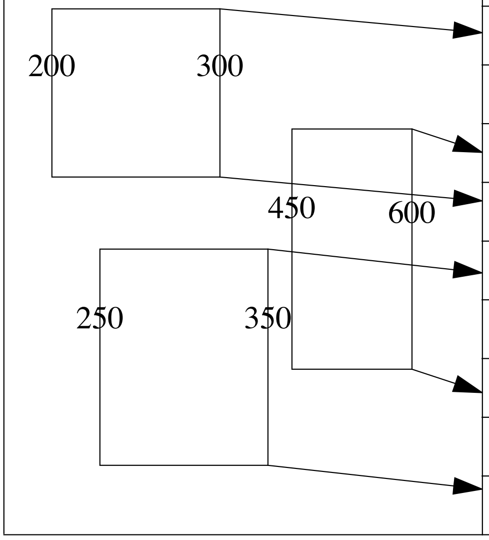
Within each set, the position field must increase with register number, this is because the video hardware only includes one comparision unit for each set. At the start of each scan line the video hardware starts comparing its output pixel position against the position field of the first matte register in each set. When the positions match, the action defined by that matte register happens and the comparison moves on to the next register in that set.

The 4 bit op-code field of a matte register allows a number of actions (see page V-84 of the Green Book for the full list), these include setting or clearing a matte flag, changing the image contribution factor of one of the two video planes and some combinations of both. There is a special op-code for terminating the matte register comparison so that higher numbered registers within that set are ignored.

Transparency with mattes is achieved by setting a PICTURE to be transparent either when a matte flag is set or a matte flag is true (in Balboa this is done using pi_settrans). Using this, either the inside or the outside of a matte can be made transparent. The big advantage of using mattes for this is that it works with any image coding method whereas the other forms of transparency only work with clut/runlength or RGB 555 images. Mattes have two disadvantages, the complexity of shapes is restricted due to there only being 8 matte registers and hence only 8 transitions per scan line. Also since matte registers are loaded from LCT or FCT instructions, using them requires a CDRTOS call and the time involved in that.

Although there is a matte flag bit in each matte register, its use is very limited. If the matte registers are functioning in one set of 8 then this bit must have the same value for all 8 instructions. If the registers are functioning in 2 sets of 4 then the bit in each register is ignored and registers 0 to 3 always effect matte flag 0 and registers 4 to 7 always effect matte flag 1. The Balboa function vs_setnmatte() controls this number of sets facility.

Here is an example of matte register use where 3 rectangles of transparency are required in a DYUV image. The table shows how the matte registers are used to achieve this and how the values in them need to change moving down the screen. The stop entry in the table means the special op-code to disable comparisons with higher numbered matte registers in that set.

	Register Number, Action and Position				
Line Number	0	1	2	3	4-7
In Field Control Table 	Stop	Stop	Stop	Stop	Stop
	Set at 200	Clear at 300	Stop	Stop	Stop
	Set at 200	Clear at 300	Set at 450	Clear at 600	Stop
	Set at 450	Clear at 600	Stop	Don’t care	Stop
	Set at 250	Clear at 350	Set at 450	Clear at 600	Stop
	Set at 250	Clear at 350	Stop	Don’t care	Stop
	Stop	Don’t care	Don’t care	Don’t care	Stop

Moving down the screen, the first rectangle always uses matte registers 0 and 1. The second rectange starts using registers 2 and 3 where it is to the right of the first. After the first rectangle has finished, the second rectangle must change to using registers 0 and 1. When the third rectangle starts, that uses registers 0 and 1 so the second rectangle must go back to using registers 2 and 3. The key to the Balboa matte manager is this translation of shapes into matte register usage on a scan line basis.

Appendix 6. Memory Fragmentation with Double Buffered FCTs

The CDRTOS function call to change from one set of FCTs to another set of FCTs is dc_exec(). This function call allocates and de-allocates memory in a way which is not at all obvious. When dc_exec() is first called, the CDRTOS video driver will allocate enough memory from each memory bank to hold the FCT for the corresponding video plane plus a little bit more. This is described in the Green Book on page VIII-20. When dc_exec() is next called, the same quantity of memory will be allocated again for both video planes and then the original memory de-allocated.

Normally, the first dc_exec() will happen as part of the call to vs_show() during application start-up. If an application uses vs_switch_fct() then another dc_exec() call will happen as part of the next call to vs_update(). At this point, a new set of shadow FCT memory will be allocated from whatever happens to be free and at the end of the call, the old set of shadow FCT memory will be de-allocated. This carries a severe risk of memory fragmentation unless the application is aware of this behaviour and has pre-planned for it.

The work-around for this behaviour is to pre-allocate twice the memory used for shadow FCTs. Before each call which could result in a dc_exec(), one half of this memory would be de-allocated ready for CDRTOS to allocate. At the end of the CDTROS call, CDRTOS would de-allocate the previous shadow FCT memory which would be the other half of the memory pre-allocated at the start. After the end of the CDRTOS call, the application should then re-allocate the half de-allocated by CDRTOS. This should all work as long as the pre-allocated memory was allocated very near application start-up and hence is the first free memory found in a search from high addresses down. This functionality may be included in a future version of Balboa after 1.4.