TN 99

Interactive
Media
Systems

CD-I

AUTHOR DISC

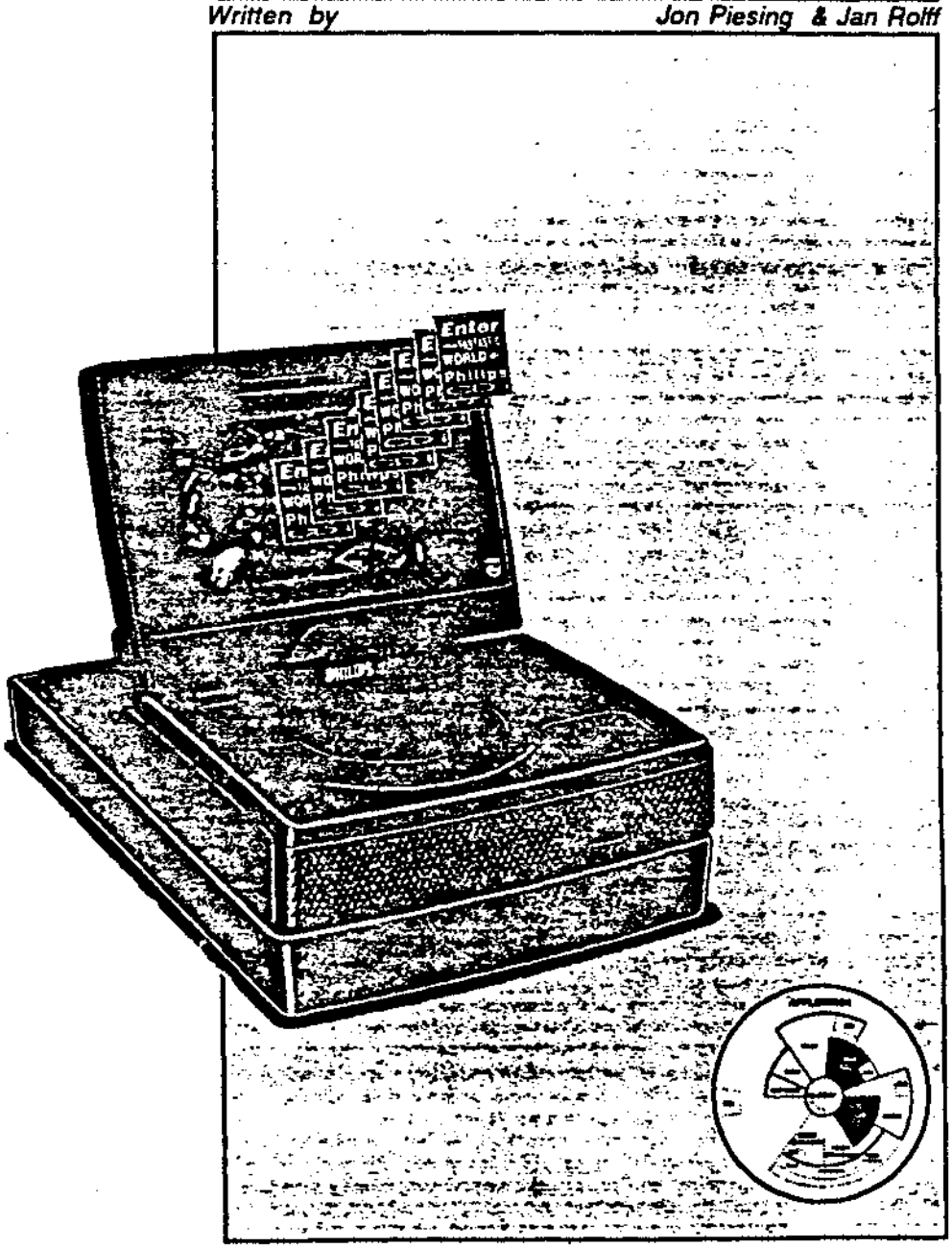
Philips Consumer Electronics B.V. Interactive Media Systems Bld. SFH - P.O. Box 80002 5600 JB EINDHOVEN NL

TSA APPLICATION NOTES NR. TSA-009 June 9 1993

Balboa Video Manager insights 1

This document contains an introduction to the Balboa video manager. It concludes with an example of a DYUV movie.

Written by *Jon Piesing & Jan Rolff*



copyright 1993 iMS TSA

nr. of pages: 75

PHILIPS

PHILIPS

ONNOTES

Responsibility

The text and program examples in this note originates from Jon Piesing, Philips Research Labs, Redhill.

However, any kind of correspondence whether by telephone or written word should be addressed to:

Jan Rolff
Philips Consumer Electronics B.V.
IMS
Building SFH 5
PO Box 80002
5600 JB Eindhoven
The Netherlands

Tel: +31 40 733 514
Fax: +31 40 734 234
E-mail: rolff@cdi-as.ce.philips.nl

TABLE OF 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 optimisations 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.

ION NOTES

3. The Managers

3.1. Video Screen

The video screen manager is the core of the Balboa video manager set-up. 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 co-ordination 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 organise 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 4.3. 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.

JONNOTES

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 6.3. below.

3.8. Video Synchronisation

The video synchronisation manager is a very low level package to handle video synchronisation. 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 MAX_INTERNAL_BUF      1
    vs_init (BP_MEM_DONTCARE, MAX_INTERNAL_BUF);
```

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 6.1. 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()***.

IONNOTES

A typical call to this looks like :-

```
#define MAX_DSEG        5
VS    *vsScreen;

      vsScreen =  vs_open
                  (
                        LCT_DOUBLE,
                        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 initialising 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 initialised. It must be separately initialised 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 typical initialisation looks like :-

```
#define     TCMAP_STRUCTURE_COUNT    4
#define     MAX_DSEG                 5

VS *vidscreen;

vidscreen = vs_open
                    (
                          LCT_DOUBLE | CURSOR_ON,
                          MAX_DSEG,
                          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 initialising 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,
                          MAX_DSEG,
                          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 initialise 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 initialised 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 initialised. The first of these will set the maximum number of scan lines which have video interrupt call-backs 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 call-backs 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,
                        MAX_DSEG,
                        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 initialised before any video effects can be used.

ION NOTES

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 call-back for when the picture is displayed
- pointer to palette data for picture

5.2. Displaying an Image From an IFF File

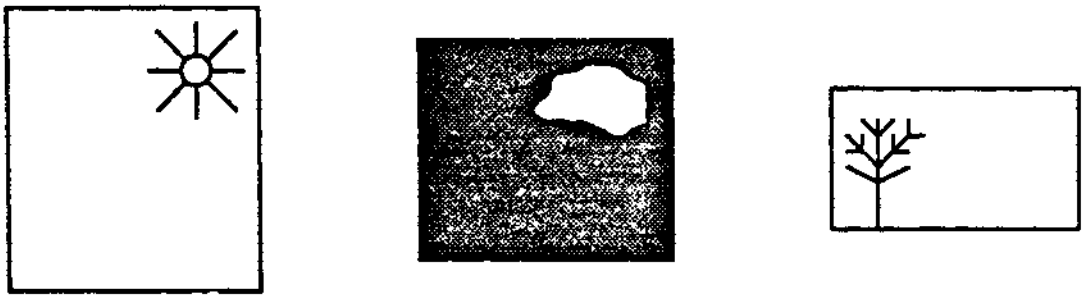
To display a picture, all that is needed is to attach that picture to a video screen and then to update that video screen. The absolute simplest code to do this is :-

```
#define     MAX_INTERNAL_BUF        1

VS          *vidscreen;
PICTURE     *pic;

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

pic = pi_iffpic( PLANE_A, 0, "my_file.dyu", BP_MEM_PLANE_A);
pi_front( vidscreen, pic );

vs_update
(
      vidscreen,
      NON_INTERLACE,
      DISPLAY_62,
      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.

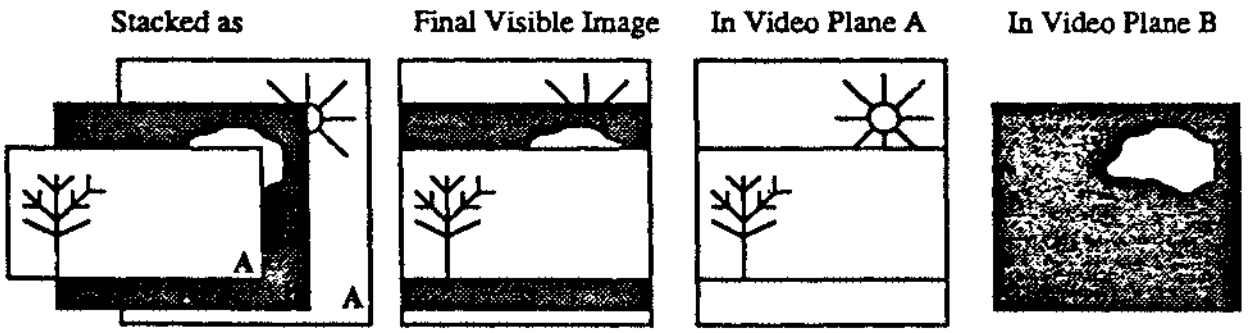
TION NOTES

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.

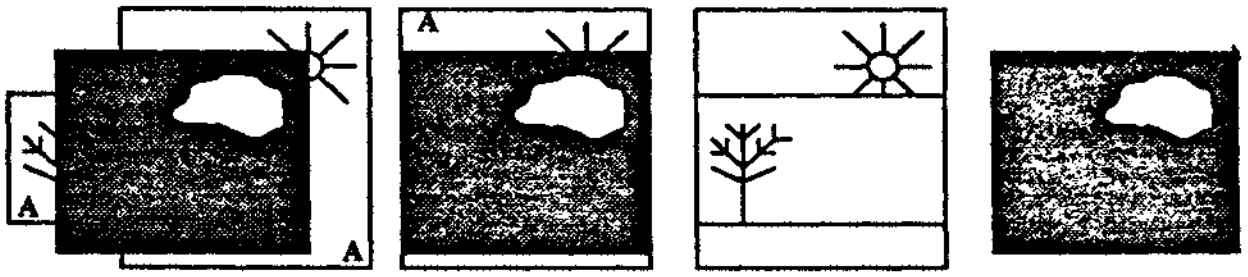


The individual PICTURE Objects

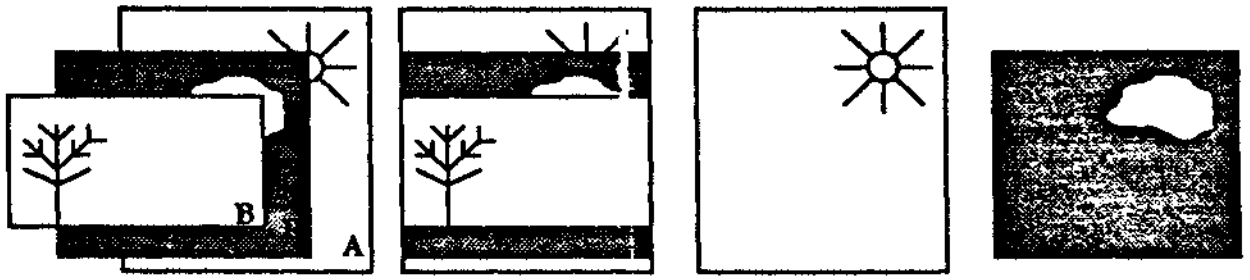
The diagram on page 16 shows the effect of changes of video plane and order within the PICTURE list. The most obvious effect of the change of PICTURE list order is the appearance and disappearance of the tree picture. A less obvious effect only becomes apparent if transparency is used. If the tree picture 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.



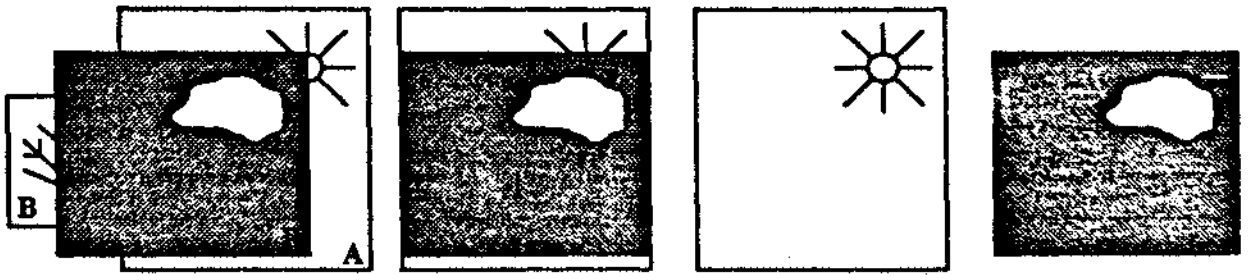
Case 1: picture order = tree, cloud, sun. Tree in video Plane A



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 co-ordinate 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 centred.

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 3 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 call-back on end of play to display the image. If this was not done then the loading of the image from disc would be visible.

ON NOTES

```
/*
     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
#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;        /* OS9 file returned by open()    */
void
```

```
        runit(),
        play_done(),
        vs_finish();


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

void runit()
{
        char    *buffer;

        /* play manager initialisations */

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

        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 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 set-up. 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 picture's 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 call-back 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 specialised 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 initialised but does require temporary use of one of the internal buffers mentioned in the description of the ***vs_init()*** call on page 8.

ION NOTES

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 21 handle clut files, the following line should be added after the call to ***pi_iffpic()*** :-

```
if (pic->pi_clut)
     cl_exec( videoscreen, 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 arrows ⇒ 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
#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;        /* OS9 file returned by open()    */
⇒ char
⇒    clut_buffer[F2_BYTES];  /* one sectors worth */
```

ON NOTES

```
void
     runit(),
     play_done(),
     vs_finish();


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

void runit()
{
     char    *buffer;

     /* play manager initialisations */

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

     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 clut buffer and the picture to the play
    manager */

⇒   pml_add_buffer
⇒   (
⇒         CHANNEL(0),
⇒         DATA_TYPE,
⇒         F2_SECTORS(F2_BYTES),
⇒         clut_buffer,
⇒         NULL,
⇒         NA,
⇒         DISPATCHED
⇒   );

    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 */
```

ION NOTES

```
                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 sa ne time as the image requires double buffering of the FCT. This is done at the tim e 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,
                        MAX_DSEQ,
                        BP_MEM_PLANEA
                );
```

The Balboa call to set-up 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,
                  NA
            );

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

/* 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, NON_INTERLACE, 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, NON_INTERLACE, 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 FCT 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);
}
```

ION NOTES

6. 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.

6.1. 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 co-operates 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 */
            *ds_pic1;     /* picture to use for plane B */
      LCT
            *ds_lct0,     /* lct to use for plane A */
            *ds_lct1;     /* lct to use for plane B */
      int
            ds_icm,       /* image coding instruction */
            ds_tci;       /* transparency control info
                          instruction */
      short
            ds_screenline,     /* start line on screen of
                               this segment */
            ds_nlines;         /* number of lines in segment */
      char
            ds_plord,         /* plane order, 0 or 1 */
            ds_reserved;      /* reserved */
      short
            ds_line0,     /* start line ( logical ) for plane A
                          LCT */
      short
            ds_line1;     /* start line ( logical ) for plane B
                          LCT */
} DSEG;
```

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

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

6.2. 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 6.3. 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
      (
            VS    *videoscreen,
            int   plane,
            int   line_number,
            int   column_number,
            int   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
                  );
}
```

ON NOTES

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.

6.3. 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 4.

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 set-up such a matte could look like :-

```
make_rect_matte( vs, plane, mw )          /* create a
                                          rectangular matte */
VS          *vs;
int         plane;
SIZE_RECT   *mw;
{
     LCT          *lct;
     short        top_column,
                  top_line,
                  btm_column,
                  btm_line;
     int          instructions[2],
                  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 */
     if( plane == PLANE_B )
     {
          lct = dsPlaying->ds_lct1;
          top_line =  top_y -
                      dsPlaying->ds_screenline +
                      dsPlaying->ds_line1;
          btm_line =  bottom_y -
                      dsPlaying->ds_screenline +
                      dsPlaying->ds_line1;
     }
     else
     {
          lct = dsPlaying->ds_lct0;
          top_line =  top_y -
                      dsPlaying->ds_screenline +
                      dsPlaying->ds_line0;
          btm_line =  bottom_y -
                      dsPlaying->ds_screenline +
                      dsPlaying->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 */
```

ION NOTES

```
        );

    /* 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 set-up 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;
```

ION NOTES

```
VS          *vidscreen;

pi_settrans( pic1, TC_MP0TRUE);
vs_update(vidscreen,0,DISPLAY_625,NULL,NULL);
make_rect_matte( 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();

/* set-up transparency and then activate the matte */

pi_settrans( pic1, TC_MP0TRUE);
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 ( VS_STATE      *state)
{
     make_rect_matte( state.vs, 100,100,600,200);
}
```

7. 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. A complete listing of this program is contained in Appendix 1.

The real time file for this movie is generated by a script as contained in Appendix 6.

This real time file starts with the background image in channel 0 and then continues with the frames of the movie. The starting sectors are generated in such a way as to achieve an average frame rate of 10 frames/second.

The actual listing in Appendix 1 starts to ***#define*** the dimensions of the DYUV partials.

```
/* The Movie */
#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    /* x pos. in ucm co-ordinates */
#define MOVIE_Y         100    /* y pos. in ucm co-ordinates */
```

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.

In the *for* loop a call to *bp_memset()* is made 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.

```
InitPictMngr ();  /* picture manager initialisation */
InitPlayMngr ();  /* play manager initialisations */

/* set-up the display */
pi_settrans ( pxMain,   TC_MYOTRUE);
pi_front    ( vsScreen, pxMain );
pi_back     ( vsScreen, pxPartial[0]);
pi_position ( pxPartial[0], srMovieWin.ul.y);

vs_update
(
      vsScreen,
      NON_INTERLACE,
      DISPLAY_625,
      NULL,NULL
);
```

In *InitPictMngr()* the 4 PICTURE objects are created:

- One to contain the foreground picture
- Three to act as a buffer to hold / display the DYUV partials

```
#define MAX_OUTBUF     3      /* Max. # of display buffers */
void       InitPictMngr()
(
     int   i;

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

     pxMain =    pi_create
                 (
                       PLANE_A,
                       D_DYUV,
                       UCM(PAL_WIDTH), UCM(PAL_HEIGHT),
                       F2_BYTES(PAL_BYTES),
                       NA
                 );

     for( i = 0; i < MAX_OUTBUF; i++ )
     (
           pxPartial[i] =
```

```
                pi_create
                (
                        PLANE_B,
                        D_DYUV,
                        UCM(PAL_WIDTH), UCM(MOVIE_HEIGHT),
                        NA,
                        NA
                );
            bp_memset
            (
                pxPartial[i]->pi_pstart,
                0,
                pxPartial[i]->pi_size
            );
        }
        return;
}
```

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 *cbMovieLoaded()* is called.

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

void        InitPlayMngr()
{
    char  *buffer;
    int   i;

    buffer =    (char*) bp_allocate
                (
                    ARBITRARY_SIZE,
                    BP_MEM_PLANEA
                );
    pmb_set_block( buffer, ARBITRARY_SIZE);
    pml_init();

    for(i=0; i < MAX_INBUF; i++)
    {
        /* allocate the memory for the input buffers */
        InBuf[i] =  (char*) bp_allocate
                    (
                        F2_BYTES(MOVIE_BYTES),
                        BP_MEM_DONTCARE
                    );
```

ON NOTES

```
            /* and declare this lot to the play manager */
            pml_add_buffer
            (
                  CHANNEL(1),
                  VIDEO_TYPE,
                  F2_SECTORS(MOVIE_BYTES),
                  InBuf[i],
                  cbMovieLoaded, NA,
                  DISPATCHED
            );
      }

      pml_add_buffer
      (
            CHANNEL(0),
            VIDEO_TYPE,
            F2_SECTORS(PAL_BYTES),
            pxMain->pi_pstart,
            cbMainLoaded, NA,
            DISPATCHED
      );
      return;
}
```

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 call-back 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 6.3.

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


/* the call-back when the pxMain is loaded */

void       cbMainLoaded (context, pml_buffer)
int        context;
PML_BUFFER  *pml_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);
     return;
}
```

As each frame is loaded from CD, the function ***cbMovieLoaded()*** 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 ***cbMovieLoaded()*** saves the address at which the last frame was loaded into a variable called source and then dispatches a call to ***display_frame()***.

ION NOTES

```
int     Pending = 0;        /* flag to say whether a frame
                               is waiting to be decoded  */
```

```
/* the call-back for each time a frame comes in from the CD
*/

void        cbMovieLoaded (context, pml_buffer)
int         context;
PML_BUFFER *pml_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 ++;

            /* dispatch the decode function */

      dispatch_function
      (
            context,
            display_frame,
            pml_buffer->Buf
      );
      return;
}
```

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 *cbMovieLoaded()* function. The destination address is taken from the line start table of the next one of the 3 small picture's to be used. This next one is shown by a variable called output.
Also the pending variable is cleared so that *cbMovieLoaded()* is now free again to set-up the decoding of another frame of the movie.

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

      void display_frame();
```

ION NOTES

```
        /* 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 ++;

              /* dispatch the decode function */

        dispatch_function
        (
              context,
              display_frame,
              pml_buffer->Buf
        );
        return;
}


/* the function which actually copies a frame onto the screen
and displays it */

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

        /* sort out the source and destination */

        linestart =
              (unsigned int*)(pxPartial[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; x +=
                                          sizeof(*source))
                    *(dest++) = *(source++);
        }
```

```
/* clear the frame pending flag so we allow to enter
another one */

Pending = 0;
```

The display is switched between the 3 small *PICTURE*s by writing a new load display start address instruction to the LCT for the plane containing the small *PICTURE*s 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)(pxPartial[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 *PICTURE*s to be used is incremented and set back to zero when it reaches the end of the set of *PICTURE*s.

ONNOTES

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

        if( ++ Output == MAX_OUTBUF )    Output = 0;
        return;
}
```

Appendix 1 Listing of DYUV Movie Playback Program

```
/*
 *  Function      : jp_dyuv.c
 *  Author        : Jan Rolff
 *  Date          : 08-Mar-1993
 *  Purpose       :
 *  Description   :
 *
 *                      History
 *
 *          Date      By                    Reason
 *      ------------+--------------------+------------------
 *      18-Feb-1993 Jon Piesing           Creation
 *      08-Mar-1993 Jan Rolff             Brush up
 *                  Tel: +31 40 733514
 *                  Fax: +31 40 734234
 */

/* #dcif# Include files */

/* 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>
```

ON NOTES

```
/* #dfdm# #define's & macro's */

#define DYUV_MOVIE      "RTF/dyuv.movie"

/* The Main picture */

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

/* The Movie */
#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

#define MOVIE_X         100    /* x pos. in ucm co-ordinates */
#define MOVIE_Y         100    /* y pos. in ucm co-ordinates */

#define MAX_INBUF       3      /* Max. # of input buffers */
#define MAX_OUTBUF      3      /* Max. # of display buffers */
#define MAX_DSEG        5

#define MAX_INTERNAL_BUF      1
#define ARBITRARY_SIZE        4096  /* any conveniently big
                                       size */
#define NA              0      /* Not Applicable */

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

/* FORM2 constants and macro's */
#define F2_BYTES_PER_SECTOR  2324  /* form 2 sector size */
#define F2_SECTORS(s)        ((F2_BYTES_PER_SECTOR -\
                             ((s) % F2_BYTES_PER_SECTOR)) + \
                             (s)) / F2_BYTES_PER_SECTOR
#define F2_BYTES(s)          F2_SECTORS(s) * \
                             F2_BYTES_PER_SECTOR
```

```
/* #dcff# Declaration of Forward functions */

void
     InitPictMngr(),
     InitVidMngr(),
     InitPlayMngr(),
     cbMovieLoaded(),
     cbEndOfPlay(),
     cbMainLoaded(),
     DoIt();
int  vs_finish();

/* #dcgd# Declaration of Global data */

/* #dfgd# Definition  of Global data */

PICTURE
     *pxMain,                    /*  the full screen picture to
                                 be used. Ask cbMainLoaded for
                                 the movie */
     *pxPartial[MAX_OUTBUF];     /* the small picture's 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,                       /* OS9 file as returned by
                                 open() */
     Pending = 0;                /* flag to say whether a frame
                                 is waiting to be decoded    */
     Output = 1;                 /* next output picture to use */
DSEG
     *dsPlaying;                 /* display segment in which the
                                 movie is playing */
SIZE_RECT
     srMovieWin =                /* Size and pos. movie window */
     {
          {
               MOVIE_X,          /* ul.x */
               MOVIE_Y           /* ul.y */
          },
          UCM(MOVIE_WIDTH),      /* width */
          UCM(MOVIE_HEIGHT)      /* height */
     };
```

ION NOTES

```
/* #dfld# Definition  of Local data */

/* #dfgf# Definition  of Global functions */

/* #dflf# Definition  of Local functions */


main()
{
     STATUS_INIT( stderr, ST_MGR_ALL, ST_TYP_ALL, 0);
     dispatch_loop( DoIt, NULL, NULL );
}
```

```
void DoIt()
{
      dispatch_atquit (vs_finish, NA);

      sgm_init();          /* initialise signal manager, needed
                           for play manager */
      InitVidMngr ();      /* video manager initialisations */
      InitPictMngr ();     /* picture manager initialisation */
      InitPlayMngr ();     /* play manager initialisations */

      /* set-up the display */
      pi_settrans ( pxMain,    TC_MP0TRUE);
      pi_front    ( vsScreen, pxMain );
      pi_back     ( vsScreen, pxPartial[0]);
      pi_position ( pxPartial[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);

      /* now go and do the play */
      File = open(DYUV_MOVIE, S_IREAD);
      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 */
            cbEndOfPlay,         /* The CB after EOP */
            NULL,
            NULL
      );
}
```

ONENOTES

```
void        InitPlayMngr()
{
      char  *buffer;
      int   i;

      buffer =    (char*) bp_allocate
                  (
                        ARBITRARY_SIZE,
                        BP_MEM_PLANEA
                  );
      pmb_set_block( buffer, ARBITRARY_SIZE);
      pml_init();

      for(i=0; i < MAX_INBUF; i++)
      {
            /* allocate the memory for the input buffers */
            InBuf[i] =  (char*) bp_allocate
                        (
                              F2_BYTES(MOVIE_BYTES),
                              BP_MEM_DONTCARE
                        );

            /* and declare this lot to the play manager */
            pml_add_buffer
            (
                  CHANNEL(1),
                  VIDEO_TYPE,
                  F2_SECTORS(MOVIE_BYTES),
                  InBuf[i],
                  cbMovieLoaded, NA,
                  DISPATCHED
            );
      }

      pml_add_buffer
      (
            CHANNEL(0),
            VIDEO_TYPE,
            F2_SECTORS(PAL_BYTES),
            pxMain->pi_pstart,
            cbMainLoaded, NA,
            DISPATCHED
      );
      return;
}
```

```
void        InitVidMngr()
{
     vs_init (BP_MEM_DONTCARE, MAX_INTERNAL_BUF);
     vsScreen =  vs_open
                 (
                       LCT_DOUBLE,
                       MAX_DSEG,
                       BP_MEM_PLANEA
                 );
     return;
}
```

NXP NOTES

```
void        InitPictMngr()
{
     int    i;

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

     pxMain =    pi_create
                 (
                       PLANE_A,
                       D_DYUV,
                       UCM(PAL_WIDTH), UCM(PAL_HEIGHT),
                       F2_BYTES(PAL_BYTES),
                       NA
                 );

     for( i = 0; i < MAX_OUTBUF; i++ )
     {
           pxPartial[i] =
                 pi_create
                 (
                       PLANE_B,
                       D_DYUV,
                       UCM(PAL_WIDTH), UCM(MOVIE_HEIGHT),
                       NA,
                       NA
                 );
           bp_memset
           (
                 pxPartial[i]->pi_pstart,
                 0,
                 pxPartial[i]->pi_size
           );
     }
     return;
}
```

```
/* the call-back when the pxMain is loaded */

void        cbMainLoaded (context, pml_buffer)
int         context;
PML_BUFFER  *pml_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);
     return;
}
```

CAMNOTES

```
make_rect_matte( vs, plane, mw )          /* create a
                                             rectangular matte */
VS          *vs;
int         plane;
SIZE_RECT   *mw;
{
      LCT           *lct;
      short         top_column,
                    top_line,
                    btm_column,
                    btm_line;
      int           instructions[2],
                    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 */
      if( plane == PLANE_B )
      {
            lct = dsPlaying->ds_lct1;
            top_line =  top_y -
                        dsPlaying->ds_screenline +
                        dsPlaying->ds_line1;
            btm_line =  bottom_y -
                        dsPlaying->ds_screenline +
                        dsPlaying->ds_line1;
      }
      else
      {
            lct = dsPlaying->ds_lct0;
            top_line =  top_y -
                        dsPlaying->ds_screenline +
                        dsPlaying->ds_line0;
            btm_line =  bottom_y -
                        dsPlaying->ds_screenline +
                        dsPlaying->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;
}
```

ION NOTES

```
        /* 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 call-back for each time a frame comes in from the CD
*/

void        cbMovieLoaded (context, pml_buffer)
int         context;
PML_BUFFER *pml_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 ++;

          /* dispatch the decode function */

     dispatch_function
     (
          context,
          display_frame,
          pml_buffer->Buf
     );
     return;
}


/* the function which actually copies a frame onto the screen
and displays it */

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

     /* sort out the source and destination */

     linestart =
          (unsigned int*)(pxPartial[Output]->pi_lstart);

     /* loop copying */
```

ONNOTES

```
        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; x +=
                                          sizeof(*source))
                    *(dest++) = *(source++);
        }
        /* clear the frame pending flag so we allow to enter
        another one */

        Pending = 0;

        /* 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)(pxPartial[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;
        return;
  }
```

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

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

ION NOTES

Appendix 2 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 3

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

Example 4

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

Example 5

```
record
     video in channel 0 from
          "/u3/jon/data/IFF/girlskil_big.cl8">

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

ION NOTES

Appendix 3 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 3alboa and CDRTOS what to do with this data.

Appendix 3.1 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 initialised using ***sgm_init()***.

Appendix 3.2 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 call-back 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 call-back to be called when this data has been loaded.

When all the ***PML_BUFFER***s 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 call-back 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 call-back.

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 4 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 6.3.**Error! No sequence specified.** on page 35 of this note. Mattes are the only mechanism which can change the CDI video set-up 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 co-ordinates 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()***.

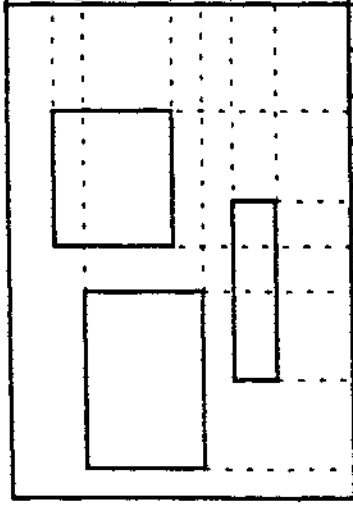
Within each set, the position field must increase with register number, this is because the video hardware only includes one comparison unit for each set. At the start of each scan line the video hardware starts comparing its output pixel position against the position field of the first matte register 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				
0	1	2	3	4 - 7
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 rectangle 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.

TONNOTES

Appendix 5 Memory Fragmentation with Double Buffered FCT's

The CDRTOS function call to change from one set of FCT's to another set of FCT's 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 FCT's. 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.

Appendix 6 The master and green file for the example in Appendix 1

```
!
! example input file for cti disc builder
!
define album "Disc Building Demo"        publisher ""
                                         preparer  "Jan Rolff"
volume "DYUV Movie Demo" in "e:/jp_dyuv.img"
!
! define the various files that must be present
!
message     from "H:/AUDIO/MESSAGE.cda"
copyright   file Copyright     from "H:/TEXT/COPYRIGHT.TXT"
abstract    file Abstract      from "H:/TEXT/ABSTRACT.TXT"
biblio      file Biblio        from "H:/TEXT/BIBLIOGRAPHIC.TXT"
application      file appl     from "H:/PROJECT/applicat.big"

green file demo from record
      video in channel 0 from
            "h:/video/imyuv/marina.d">CAT#IMAG at 00:00:00

      video in channel 1 from
            "h:/video/imyuv/bs01.d">CAT#IMAG at 00:01:15,
            "h:/video/imyuv/bs02.d">CAT#IMAG at 00:01:22,
            "h:/video/imyuv/bs03.d">CAT#IMAG at 00:01:29,
            "h:/video/imyuv/bs04.d">CAT#IMAG at 00:01:36,
            "h:/video/imyuv/bs05.d">CAT#IMAG at 00:01:43

(
   "copyright" protection 0x111 from Copyright
   "abstract"  protection 0x111 from Abstract
   "bibliographic" protection 0x111 from Biblio

   "CMDS"
   (
        "jp_dyuv" from appl
   )
   "RTF"
   (
        "dyuv.movie" from demo
   )
)
```

ION NOTES

INDEX