TN 91

Interactive
Media
Systems

disc interactive NOTES

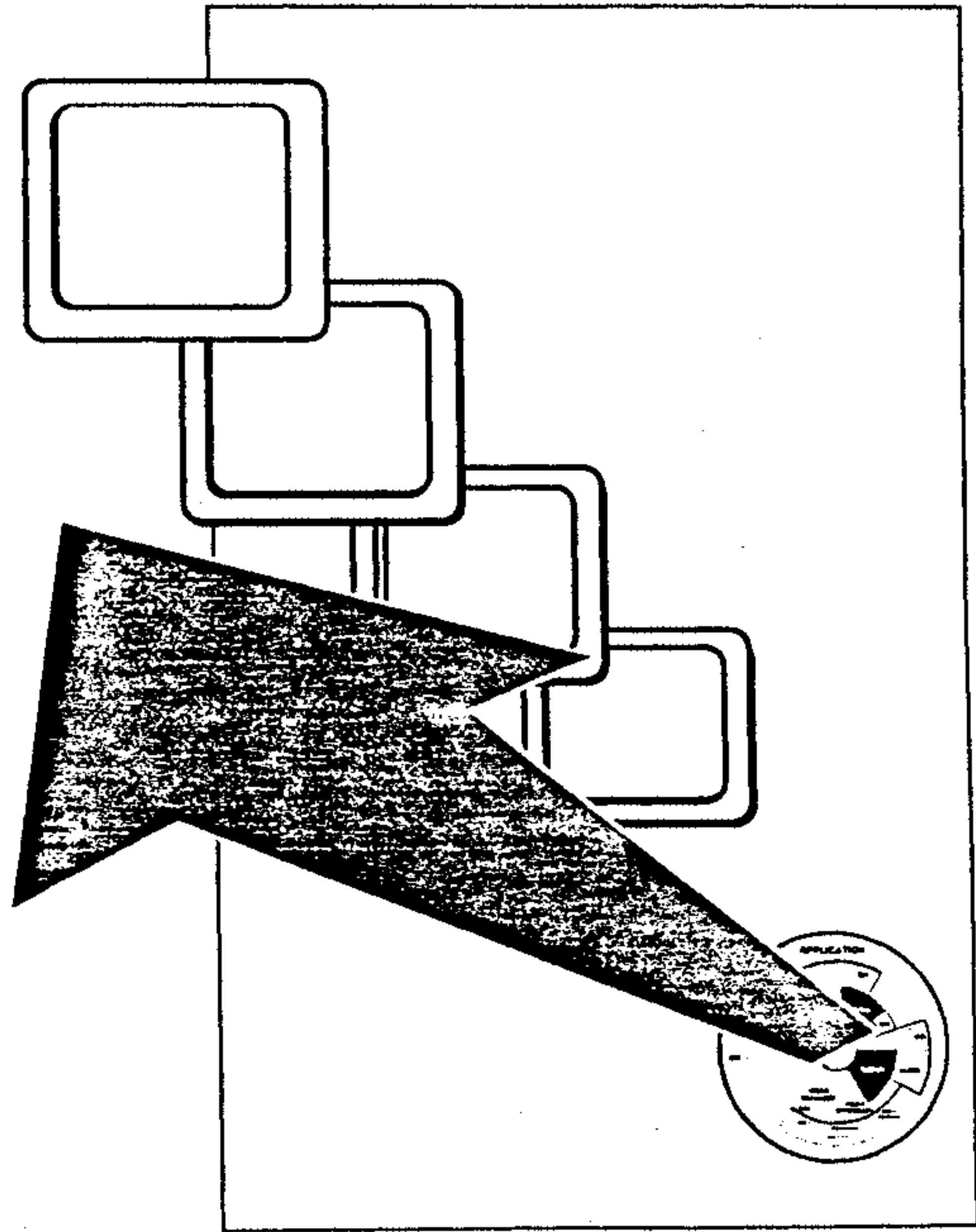
CD-I

TSA APPLICATION NOTES NR. TSA-001 April 9 1992

A Beginner's Guide to Balboa: Edition 1

This document provides a simple overview of Balboa and includes a simple slide-show application.

Written by *Andy Maxfield, Jon Piesing*



Philips Consumer Electronics B.V. Interactive Media Systems Bld. SFH - 6 P.O. Box 80002 5600 JB EINDHOVEN

copyright 1992 IMS TSA nr. of pages: 28

PHILIPS

2. Balboa Philosophy

There are a number of key philosophical principles behind Balboa.

2.1. A Callback Philosophy

Balboa uses a callback philosophy. A callback is an application function which is executed when an event occurs. The application instructs Balboa which events are of interest and specifies which application function should be executed when this event occurs. When the event occurs Balboa initiates the callback/application function. Callbacks must be short and should never, never wait for events or anything else that can happen in the external world.

Refer to chapters 2-9 and 2-10 from volume 1 of the programmers guide for more information.

2.2. Modularity

Balboa is structured as a number of groups of functions, each group dealing with a different area of functionality. These groups are called managers. There are three managers at the central core and others grouped around these. Applications need only use those managers relevant to their needs together with appropriate dependencies.

As far as possible, Balboa is organised so that managers which are not referenced by the application will not be included by the linker, this is also taken down to the individual C function level .

Refer to chapters 2-20 from volume 1 of the programmers guide for more information.

2.3. Open Systems and Multi-Level Access

When Balboa and its components were designed, it was recognised that it is important to allow multiple levels of access for applications which need to do something which Balboa does not include or does not do fast enough. A number of managers have what would normally be internal data structures and functions documented in order to allow applications to perform what is important to them without having to fight Balboa or to cease using Balboa in that area. There are also a number of low level managers documented which are normally used only internally.

2.4. Application Control of Memory Usage

One of the most critical issues for CD-I applications is memory usage. One of the design requirements for Balboa was that advanced applications which needed total control over their memory use should be able to have it. The way this has been implemented is that nearly all functions which allocate memory come in two or three versions, one which allocates memory, one which allows the application to specify exactly which memory is to be used and a third which calculates the memory required for the second one. The same applies for functions which de-allocate memory. All these functions have consistent naming conventions for example:

pi_create - create a picture, Balboa will allocate the memory required
pi_create_mem - create a picture using memory provided by the application
pi_create_size - calculate size of memory required by the above

For a beginner, this has the effect of making Balboa seem more complex than it really is since there can be 2 or 3 functions all concerned with the same thing. The easiest function to use is to have Balboa allocate the memory, so this is used throughout the example.

Refer to chapters 2-20 from volume 1 of the programmers guide for more information.

3. Balboa Fundamentals

This section provides a description of the core functionality of Balboa which are immediately relevant to new users of the package.

3.1. Managers

There are three managers at the core of Balboa, the signal manager, the dispatcher and the status manager.

3.1.1. Signal Manager

Signals are the principal mechanism which CD-RTOS uses to inform CD-I applications that various events have happened. Balboa provides a programmable CDRTOS intercept routine, which allows multiple Balboa managers and applications all to receive signals without interfering with each other.

Applications or Balboa managers which want to use the signal manager ask the signal manager to create a class of signals for them to use. The application/ manager can specify a Balboa callback for each signal number in the class. When a callback is defined for a specific signal number, along with the function address and parameter, a mode must be defined. This mode can either be dispatched or immediate. Dispatched mode callbacks will be put into the dispatcher to be executed on first come, first served basis. Immediate mode callbacks are more like CPU interrupts and should not be used by beginners.

3.1.2. Dispatcher Manager

The dispatcher manager implements the basic callback methodology of Balboa. It manages and maintains a first in, first out queue of functions to call. Functions can be put in the queue by other Balboa managers or by the application. They are pulled from the queue in the order in which they were inserted and are then called. The dispatcher creates the illusion of multiple threads. Threads can be compared to a process in a multi-tasking environment.

3.1.3. Status Manager

One of the major problems with a real time, multi threaded CD-I application is debugging it. The status manager provides a very powerful tool to assist the application programmer in this.

```
STATUS_INIT(stderr,ST_MGR_ALL,ST_TYP_ALL,0);
```

This will output all status manager messages generated onto stderr.

- **bp_allocate()**

The basic memory allocation function. All memory allocated by Balboa is allocated through this function. This is to enable a memory manager to be added either as part of Balboa at a later date or by the application itself. A code fragment could be :

```
ptr = (type*) bp_allocate( size, bank );
```

The second parameter should be one of the memory types defined in bp_mem.h, BP_MEM_PLANEA, BP_MEM_PLANEB, BP_MEM_SYSTEM, BP_MEM_DONTCARE.

- **dispatch_loop()**

This is the main loop of a Balboa application. It waits for events to happen and then calls the functions which have been setup earlier by the application and by Balboa. A typical code fragment for this could be :

```
void runit();
main(argc,argv)
int argc;
char **argv;
{
        dispatch_loop( runit, argc, argv )
}
void runit( argc,argv)
```

4. Fundamentals of Video

4.1. Structures and Objects

The two basic video objects are the video screen and the picture. The picture structure describes an array of pixels in memory. It is an object in the sense that it contains all the information needed to directly display it on the video hardware of the CD-I system. Some examples of the information contained in it are size, image coding method, which CD-I video plane the pixel memory is in, various CD-I video properties such as transparency condition and colour.

The video screen structure totally defines one particular configuration of the CD-I video hardware. It is central to the Balboa video architecture and all the other items are attached to it. It contains a number of links to other structures which are outside the scope of this introduction.

4.2. Managers

There are a number of Balboa managers (or sub-managers) associated with the video area. Only 4 are relevant for this introduction, these are video screen, picture, video interrupt and video effects (their function calls start with vs_, pi_, vi_ and vfx_). The video screen and picture managers contain functions concerned with creating, deleting and using the respective objects. The video effects manager provides a number of television style video effects between pictures. The video interrupt manager is required to control the timing of these.

4.3. Display Generation

Each video screen has a linked list of pictures attached to it. This list can be considered as an infinite number of logical video planes. The video screen manager function vs_update() performs a mapping from this infinite number of planes into the two actual planes of the CD-I video system. There are some diagrams showing this mapping on pages 6-16 and 6-17 of the Balboa programmers guide.

The vs_update() function also performs all operations necessary to display the results of the mapping operation. It takes a video screen and displays it and all the objects attached to it.

4.4. PAL / NTSC Compatibility.

One of the key requirements for CD-I applications is that they should work on any player, anywhere in the world. This imposes requirements on the title both at the visual design level and the programming level.

The Balboa video screen manager provides support to make this task easier for programmers. It works with a virtual screen area containing 600 lines, numbered -40 to 560. By convention, all coordinate references in UCM, CDRTOS and BALBOA are in high resolution. See also the Green Book chapter 7 page 83. A window within the 600 lines will be calculated based on combinations of the player video hardware type and the video hardware type requested by the application.

If the application requests a PAL display on a PAL player then lines 0 to 560 of the screen will be displayed. For an NTSC display on a NTSC player lines 0 to 480 will be displayed. If the application requests a PAL display and the player is NTSC then lines 40 to 520 of the screen will be displayed. If the application requests a NTSC display on a PAL player then lines -40 to 520 will be displayed. Applications designed for a PAL player should be aware that lines 0-39 and lines 520 to 559 will not be displayed on a NTSC player. Applications designed for an NTSC player should be aware that 40 black lines will be inserted above and below their image data. These applications can choose to provide image data for lines -40 to -1 and 480 to 519 if this letterboxing is undesirable.

4.5. Functions

- vs_init()

This function initializes the basic video system. This includes reading the player video hardware type and obtaining a path to the CD-I video device. Example call:

```
vs_init(BP_MEM_DONTCARE,1);
```

The first parameter can be any of the allowed memory types as described with bp_allocate() above. The second parameter defines the sizes of some internal buffers. A value of 1 is normally fine for this.

- vs_open()

This call creates the basic video structures to which all others are attached. An example code fragment is :

```
VS *vidscreen;

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

The first parameter is a set of flags which defines various options to be used when creating the video screen. The two shown here are enable double buffering of LCTs and enable the cursor. Double buffering of LCTs is required except in certain specific conditions. The second parameter sets the size of various internal data structures. This number should be increased for complex screen layouts. The last parameter defines the memory bank used for Balboa memory allocations while in vs_open().

- vi_init()

The video effects library uses the video interrupt manager for timing purposes. Typical initialisation code for this could look like :

```
VS *vidscreen;

vi_init(vidscreen->vs_videnv,BP_MEM_DONTCARE,2,4);
```

The two numbers refer to the maximum number of lines which can have video interrupt callbacks attached to them and to the maximum number of video interrupt callbacks. The first of these should not be more than two or three since too many interrupts can saturate the system. No such limit applies to the number of callbacks.

- pi_create()

This function creates a picture, an array of pixels in memory and a structure describing it. A typical call could look like :

```
PICTURE *pic;
pic=pi_create(PLANE_A,D_CLUT7,768,560,0,0);
```

The first parameter is which of the two CD-I video planes contains the picture. The next parameter contains the image coding type and various flags, in this case

the image is clut 7 and the flags are left to take their default values. The third and fourth parameters are the size in high resolution coordinates of the pixel memory. In this case 768 by 560 translates into 384 by 280 in pixels.

Data loaded from a real time file is loaded in integer multiples of 2324 bytes. The fifth parameter to pi_create() specifies a minimum number of bytes to be allocated and so can be used to force the memory size to a multiple of 2324. The last parameter is ignored for all image types other than RGB555 or QHY.

- pi_front()

As described above, the actual display of the CD-I display is generated by parsing a linked list of pictures. This function inserts a picture in that list at the front. Example call :

```
VS *vidscreen;
PICTURE *pic;

pi_front( vidscreen, pic );
```

- vs_show()

The video screen mechanism allows Balboa applications to have multiple independent video configurations if that is appropriate. The function vs_show() causes one particular configuration to become active. Example call:

```
VS *vidscreen;

vs_show( vidscreen );
```

- vs_update()

This function is the highest level interface to the video screen managers. It takes a video screen, analyses everything attached to it and then performs all the CDRTOS calls needed to display it. Example call :

```
VS *vidscreen;

vs_update( vidscreen, 0, DISPLAY_625, 0 , 0 );
```

The first parameter is the video screen to display. The second parameter is used to enable / disable interlaced video output. The third parameter is the hardware

configuration which the application would prefer to run under, other options include DISPLAY_525_TV and DISPLAY_525. The last two parameters are a low level interface into Balboa and are outside the scope of this introduction.

- vfx_cross_fade()

This function will perform a dissolve from one picture on a video screen to another picture on the same video screen. Example call :

```
PICTURE *pica,*picb;
void done();

vfx_cross_fade( pica, picb, 200, 0, done, 0 );
```

The first two parameters are the the source and destination pictures. The next parameter is the duration measured in 256ths of a second. The fourth parameter indicates whether the original picture should remain attached to the video screen or not.

The functions in the Balboa video effects library are asynchronous, the call from the application to the vfx_ function starts the effect. Control returns to the application while the effect progresses. The last two parameters to the vfx_ functions specify a C function to be called when the effect is finished.

- vfx_fade_up()

This function is very similar to vfx_cross_fade() except that it fades up one picture from black rather than working between two pictures. For it to work properly, the pi_icf entry in the picture structure should be set to zero before the picture is displayed. Example code :

```
PICTURE *pica;
void done();

pica->pi_icf = 0;
vfx_fade_up( pica, 200, 63, done, 0 );
```

The parameters are the same as for vfx_cross_fade() except that one picture is specified instead of two and the third parameter defines the final image contribution factor to be arrived at, in this case the maximum value, 63.

- vfx_fade_down()

This function will fade down a picture from normal intensity to black. Example code :

```
PICTURE *pica;
void done();

vfx_fade_down( pica, 200, 0, done, 0 );
```

The parameters are the same as for vfx_fade_up().

5. Interactivity

5.1. Managers

For the purposes of this introduction, there are three managers concerned with interactivity, these are the hot-spot manager, the cursor manager and the video screen manager. The hot-spot manager is concerned with the pointing device input and mapping it into regions of the screen which the application has defined to be sensitive to pointing device behaviour. The cursor manager is concerned with the visible hardware cursor, its properties and behaviour. Since the hot-spots are regions of the screen, the video screen manager is also involved in hot-spots.

5.2. Structures and Objects

The basic interactive structure is the hot-spot. This defines a rectangular area of the screen which is sensitive to user input from the pointing device. Hot-spots are arranged in trees. Each tree has a root hot-spot at the top of the tree with children attached to it. There is an explanation of this on pages 13-5 to 13-7 of the Balboa programmer's guide, volume 2.

Each hot-spot has attached to it a cursor callback structure. This is a list of Balboa callbacks together with an event mask for each callback. If any of the conditions specified by the mask have occured with the cursor in that hot-spot then a call to that function will be inserted into the Balboa dispatcher. There are also various special flags which can be put into the mask, these are described on page 13-35 of the Programmer's guide.

Any hot-spot, including the root, may have a cursor structure attached. The cursor structure defines the various properties of the CD-I hardware cursor and also has attached a linked list of patterns. These patterns can very easily be setup to be animated.

Multiple layers of hotspot trees can be defined, to cater for more complex display organisations (e.g. a scrolling background plane in combination with a fixed control box). For the sake of this introduction a simple hot-spot tree will be defined.

6. Functions

- hot_create_root()

To be able to use hot spots at all, there must be a root hot-spot for all others to be attached to. Example code :

```
void hotfunc();
HOTSPOT *root;
CPCALLBACK rootcb = { EV_B0D | DISP_LASTONE , hotfunc, 0 };
root=hot_create_root( 768,560, NULL, &rootcb, NULL, BP_MEM_PLANEA );
```

The first two parameters define the lower right corner of the root hot-spot. Root hot-spots are conventionally positioned with the top left corner at 0,0. The next parameter can be used by applications to support non-rectangular hot-spots, this is outside the scope of this introduction. The fourth parameter is a pointer to the cursor callback structure, in this case that is statically declared, the function hotfunc() will be called when button zero is pressed down.

The fifth parameter defines a cursor to be used, here we specify none so the Balboa default will be used. The last parameter is a memory bank to allocate any memory from.

- hot_create()

To create a normal hot spot (i.e. not a root one), the function hot_create() is used. The parameters are very similar to hot_create_root() except that there are three extra parameters on the front of the list, a parent hot-spot and the coordinates of the top left corner of the hot spot since it is no longer fixed at 0,0. Example code

```
void myfunc();
HOTSPOT *root;
CPCALLBACK hotcb = { EV_B0D | DISP_LASTONE , myfunc, 0 };
hot_create( root, 100,100,200,200, NULL, &hotcb, NULL, BP_MEM_PLANEA );
```

- vs_cp_hstree_install()

Since hot-spots are active areas on the screen of the CD-I system, before it can become active a hot-spot tree must be attached to a video screen. This is done using vs_cp_hstree_install(). Example :

```
VS *vidscreen;
HOTSPOT *root;
vs_cp_hstree_install( vidscreen, root, 0,0, BP_MEM_DONTCARE );
```

The first two parameters are the two objects to be connected. The next two are a coordinate offset between the coordinate space of the video screen and of the hot-spot tree, these are here set to 0,0. The last parameter is a memory bank to use for some memory allocations that need to be done for this function.

- vs_cp_show()

This function enables the cursor manager display. For simple applications it need only be called once, during initialisation. Example code :

```
VS *vidscreen;
vs_cp_show(vidscreen);
```

- vs_cp_start()

This function enables the various functions of the cursor and hot spot manager, such as making the cursor follow the movements of the pointing device, cursor pattern cycling and hot spot detection. For simple applications it need only be called once, during initialisation. Example code :

```
VS *vidscreen;
vs_cp_start( vidscreen);
```

7. Retrieving Data from a CD-I Disc

7.1. Managers

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 do this. The Balboa play manager is multi-leveled, simple applications need only concern themselves with the low level play manager and that is what is described here.

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

7.2. Structures and Objects

There are two basic data structures in the Balboa play manager, play events and play buffers. The play buffer describes a group of sectors on a particular channel and where they are to be put in memory. Play events are outside the scope of this introduction, they are explained in chapter 12 of the Programmer's Guide.

7.3. Functions

- pmb_set_block()

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. Example code :

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

For simple applications, a size in the region of 3K to 4K should be adequate.

- pml_init()

This function initialises the Balboa play manager. It performs various initialisations for other Balboa managers. Example :

```
pml_init();
```

- **pml_add_buffer()**

This function adds a buffer into the play buffer list. This list defines where the data from the CD is to be put in memory during the course of playing a real time file. Example code fragment:

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

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

- **pml_play()**

This function starts a real time file play. As the data comes in from the disc it will be loaded into memory as described in the play buffer list created previously. Example code :

```
void play_done();
int file;

pml_play(file, 0, 31, 1, 3, 0, restart, NULL, NULL );
```

The first parameter file is the system path to the file to play. This would be returned from an open() call. The third parameter is a bit mask which defines those channels which are desired for input from the disc. This is a binary bit mask and so if channels 3 and 4 are required then the mask would be (2<<3) | (2<<4) = 0x0c = 12.

The fourth parameter defines a channel number from the disc for direct audio play. This should have not more than one bit set. The channel mapping is the same as the third parameter. The fifth parameter is the number of real time records to play. For this simple application, this can be found from the number of record statements in the input file for the master program.
The seventh and eight parameters are a callback which will be called when the playing of the real time file has finished. This can be when the specified number

of records has been played or at the end of the file.

The second, sixth and last parameters should be zero for novice users.

- pml_cleanup_all()

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(). Example call :

```
pml_cleanup_all();
```

8. Required CDRTOS Functions

- open()

Before a real time file can be played, the file must be opened. The CDRTOS call open() is used to do this. Example call:

```
int file;

file = open( FILENAME, S_IREAD );
```

The first parameter is the name of the file to open, the second is the mode to open it with, in this case read only.

- ss_pause()

It is possible to pause the delivery of data during a real time file play. The example program here uses the CDRTOS call ss_pause() to do this. Example call:

```
int file;

ss_pause(file);
```

- ss_cont()

To restart the playing of a real time file after an ss_pause(), ss_cont() is used. Example call :

```
int file;

ss_cont(file);
```

9. Putting it all together

In the previous chapters each of the individual functions has been described that you need in order to create the very simple example that is annexed. In order to get this tutorial running, you need to compile and link with the Balboa libraries and replace the executable in the disc image as described in appendix B. In case you encounter difficulties call the support group in Eindhoven; refer your queries to:

Andy Maxfield
Tel : 040-737193
Fax: 040-734234
E-mail: andy@cdi-as.ce.philips.nl

```
void runit( argc,argv)
int argc;
char **argv;
{
        char *buffer;

        /* initialise the various managers as described above */

        STATUS_INIT(stderr,ST_MGR_ALL,ST_TYP_ALL,0);
        sgm_init();
        vs_init(BP_MEM_DONTCARE,1);
        vidscreen=vs_open( LCT_DOUBLE|CURSOR_ON,5,BP_MEM_
        PLANEA );
        vi_init(vidscreen->vs_videnv,BP_MEM_DONTCARE,2,4);

        /* create two pictures, one in each video plane ready to do
        video effects between them. */

        pica =pi_create(PLANE_A,D_DYUV,768,480,40*2324,0);
        picb =pi_create(PLANE_B,D_DYUV,768,480,40*2324,0);
        pica1 =pi_create(PLANE_A,D_DYUV,768,480,40*2324,0);
        picb1 =pi_create(PLANE_B,D_DYUV,768,480,40*2324,0);
        /* set the image contribution factor of the first picture to
        zero ready to fade up when it is loaded */

        pica->pi_icf = 0;

        /* setup ready to display everything */

        pi_front( vidscreen, pica );
        vs_update(vidscreen,0,DISPLAY_525_TV,0,0);
        vs_show( vidscreen );

        /* create and enable the hot spots */

        root=hot_create_root( 768,560, NULL, &rootcb,
                                        NULL, BP_MEM_PLANEA );
        vs_cp_hstree_install( vidscreen, root, 0,0, BP_MEM_DONTCARE );
        vs_cp_show(vidscreen);
```

```
        vs_cp_start(vidscreen);

        /* initialise the play manager */

        buffer = (char*) bp_allocate( 2*4096, BP_MEM_PLANEA );
        pmb_set_block(buffer,2*4096);
        pml_init();

        /* open the file */

        file = open(FILENAME,S_IREAD);

        /* run the stuff */

        start_play();
}
```

```
void start_play()
{
        /* setup two play manager buffers, one for each picture */

        pml_add_buffer( 0,VIDEO_TYPE, 40,pica->pi_pstart,
                                picture_loaded,0, DISPATCHED);
        pml_add_buffer( 0,VIDEO_TYPE, 40,picb->pi_pstart,
                                picture_loaded,0, DISPATCHED);
        pml_add_buffer( 0,VIDEO_TYPE, 40,pica1->pi_pstart,
                                picture_loaded,0, DISPATCHED);
        pml_add_buffer( 0,VIDEO_TYPE, 40,picb1->pi_pstart,
                                picture_loaded,0, DISPATCHED);

        /* initialise the variables which define where we are in the
        real time file */

        count = flag = 0;

        /* set the image contribution factor of the first picture to
        zero ready to fade up when it is loaded */

        pica->pi_icf = 0;

        /* start playing again back at the start */

        pml_play( file, 0, 31, 1, 3, 0, restart, NULL, NULL );
}
```

```
void picture_loaded( context, buf)
int context;
PML_BUFFER *buf;
{

        PICTURE *from;

        if( count == 0 )
        {
                /* if this is the first image then do a fade up */

                vfx_fade_up( pica, 300, 63, fxdone, 0 );
        }
        else
        {
                /* otherwise do a dissolve */
                switch( (count-1)%4)
                {
                case 0:
                                from = pica;
                                to   = picb;
                                break;
                case 1:
                                from = picb;
                                to   = pica1;
                                break;
                case 2:
                                from = pica1;
                                to   = picb1;
                                break;
                case 3:
                                from = picb1;
                                to   = pica;
                                break;
                }
                vfx_cross_fade( from, to, 300, 0, fxdone, 0 );
        }
    count++;
}
```

```
void restart()
{
        /* stop the play */

        ss_abort(file);

        /* clean up the play manager */

        pml_cleanup_all();

        /* fade the current picture down to black */

        vfx_fade_down( to, 300, 0, start_play1, 0 );
}

void start_play1()
{
        /* Re-initialise the picture objects. */
        pica->pi_icf = 0;
        to  ->pi_icf = ICF_MAX;
        pi_front( vidscreen, pica );
        vs_update(vidscreen,0,DISPLAY_525_TV,0,0);
        start_play();
}

void fxdone()
{
}

int flag2 = 0;

/* use the hot spot to stop and start the play */
```

```
void hotfunc()
{
        if( flag2 == 0 )
        {
                ss_pause(file);
        }
        else
        {
                ss_cont(file);
        }
        flag2 ^= 1;
}
```

Appendix B

The following utility can be used to replace an application within a CD-I image with a new application. The old version of this utility was referred to as cdimage. The new name is CDEDIT, which can be used as follows:-

cdedit [<opts>] <CD-I image file name> [<opts>]
Options:

-o	Use old image format (512 bytes header)
-e	Image does not include the initial pause
-s	Assume image is not scrambled
-?/	Give this help

N.B. Of course the application can also be changed by rebuilding the image provided by OptImage. The master script is called build.cd