TN 100

Responsibility

The text and program examples in this note originates mainly 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
CompuServe: 100142,2464

Prologue

Since this note is the second part of a sequel of two (the first one was published June 9, 1993 as **TSA-009: Balboa Video Manager insights 1**) and is aimed at experienced users of **Balboa**, the reader is well advised to read **TSA 009** prior to this one.

TABLE OF CONTENTS

1. Introduction to Video Synchronisation

Video synchronisation is critical to achieving high quality CD-I titles.
The CD-I system offers a number of ways of addressing these issues and Balboa includes two managers specifically targeted at this area, the video interrupt manager and the video synchronisation manager.
As mentioned earlier (see TSA 009), the video synchronisation manager is a low level manager that the video interrupt manager calls and the discussion here will only address the video interrupt manager level.

1.1. CDRTOS Video Synchronisation Facilities

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

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

1.2. Strategies, Pitfalls and Warnings

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

As mentioned previously in the discussion of ***vs_update()*** (see **TSA 009**), **Balboa** makes extensive use of double buffered **LCTs** so that it can build LCTs without having to worry about explicit scan synchronisation.
The example code in this section is a hardware scroll that also uses double buffered **LCTs** as does the **Balboa** sprite manager.

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

Some applications may have compelling reasons for not wanting to wait for end of field.
For these applications, triple buffering can be useful if they have enough memory. With three buffers, an application can start accessing the third buffer where it would otherwise have to wait for the second buffer to start being displayed. The **DYUV** movie example mentioned in **TSA 009** uses even four output-buffers specifically for this reason.

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

1.3. Some Hidden Nasties

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

One particular problem where video synchronisation may be required is where instructions are being written to the currently active LCT.
The path into the CD-i video memory need only be 16 bits wide and the video processor can steal cycles between the two 16 bit accesses needed to write a 32 bit LCT instruction.
If the video hardware was reading the same instruction as the application was writing then it is theoretically possible that the video hardware could read 16 bits of the previous value and 16 bits of the new value. I have not seen an instance of this being a problem but if an application experiences some form of flashes that are one field in duration then it may be worth looking at scan synchronising LCT accesses.

Another hidden nasty concerns the CDRTOS call *dc_flnk()*. Due to historical differences between the Philips CD-i chips and the Green Book, this function has a lot more work to do than the Green Book description implies. Calls to this function should generally be scan synchronised to avoid it being called during the vertical retrace.

Using the function ***dc_llnk()*** to connect **LCTs** together can reduce this problem. In **Balboa** ***vs_update()*** normally switches **LCTs** using ***dc_flnk()***. By specifying the option ***VSF_LLNK*** in the flags input to ***vs_open()***, ***vs_update()*** will use ***dc_llnk()*** instead. This might be something to try if experiencing obscure scan synchronisation problems near where ***vs_update()*** is called.

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

1.4. The Video Interrupt Manager

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

```
VIDEO_INT
     *VideoInterrupt;  /* video interrupt for timing */
VIDENV
     *VE;              /* video environment */

     VideoInterrupt =
          vi_add
          (
               VE,
               VE->ve_screenstart+2,   /* if display scan
                                          reaches this line,
                                          then */
               pxPicture->pi_plane,
               0,                      /* at every
                                          display scan, */
               0,                      /* continuously */
               cbScroll,               /* call: cbScroll
                                          */
               &PositionPerIKfields,   /* until told
                                          otherwise */
               DISPATCHED
          );
```

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

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

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

```
VS   *vsScreen;

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

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

2. Video Synchronisation Example - Scrolling a Large Image

2.1. Introduction

A good example where video synchronisation is very important is scrolling images around the screen using the hardware facilities. Complete source code for an example program to do this is in Appendix 1. The basic initialisation code is the same as the other examples. This example function will scroll a full screen image horizontally, vertically or diagonally. This chapter is divided into three parts:

- initialisation (paragraph 2.2.)
- scrolling (paragraph 2.3.)
- termination (paragraph 2.4.)

2.2. Initialisation

When *vs_update()* is asked to display a **PICTURE** whose line length is different from the hardware line length of the **CD-i** player the application is running on, it automatically puts a 'load display start address instruction' in the **LCT** for each scan line where that **PICTURE** is to be displayed . This display start address can be obtained from **PICTURE**'s member ***pi_lstart***, which is a pointer to an array of pointers to the beginning of each line to be displayed.
Such **PICTURE**s can be scrolled by replacing these with load display start address instructions pointing to other addresses within the **PICTURE**.

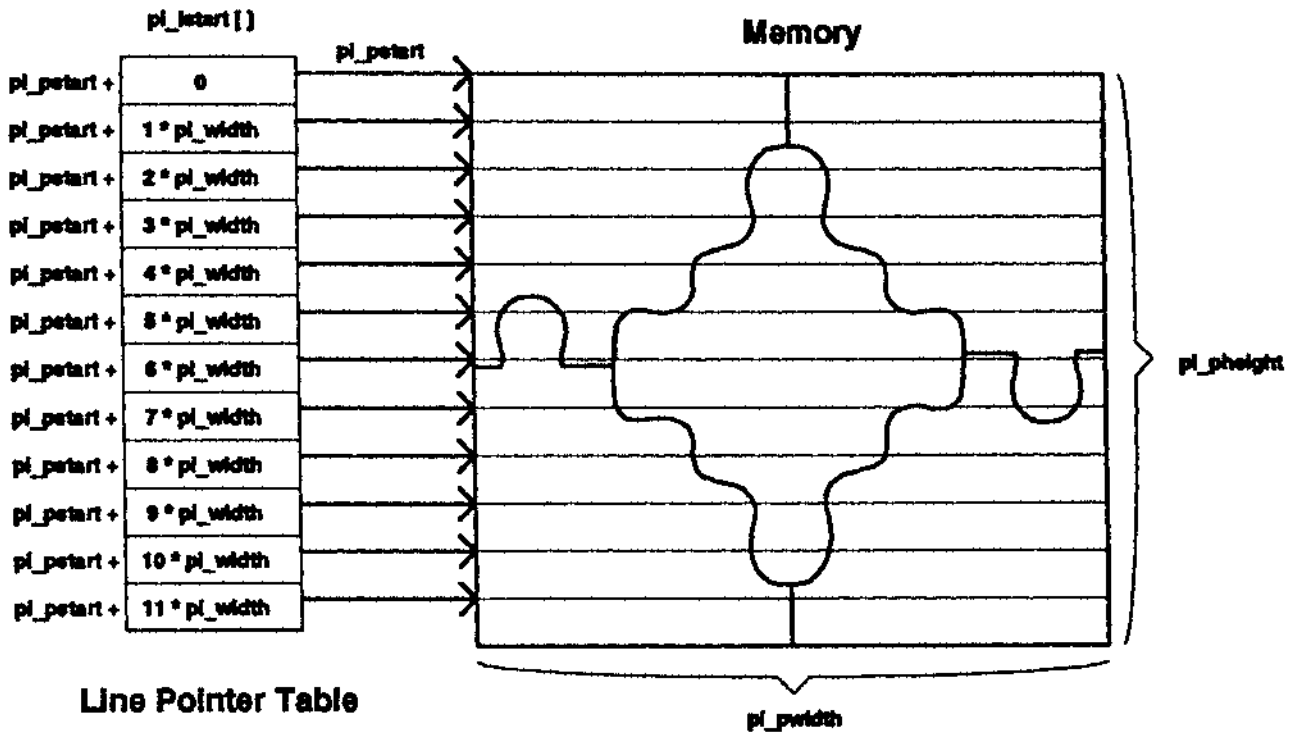


Figure 1 Line start addresses in PICTURE object

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

A smooth scroll requires a very even scroll speed. An integer number of pixels per field interval will certainly give this.
This imposes some restrictions if a scroll is to run at the same apparent speed on both **NTSC** and **PAL** players.
In this example, the interface forces the scroll speed to be a multiple of 75 pixels per second that becomes n*1.5 pixels per field (75/50) on a **PAL** player and n*1.25 pixels per field (75/60) on an **NTSC** player.
Unlike the **Balboa** video effects' library, this example will not work with a split screen and will not work with a high resolution or interlaced display. Also it will only work with clut images. For a **DYUV** image much more work is required, see **PIMA** technical note #34 **"DYUV Panning Algorithms"** for more information on this.

The example function is split into 5 sections, ***cbSetupScroll()***, ***cbScroll()***, ***cbShadow_LCT()***, ***cbOriginal_LCT()*** and ***cbScrollDone()***.

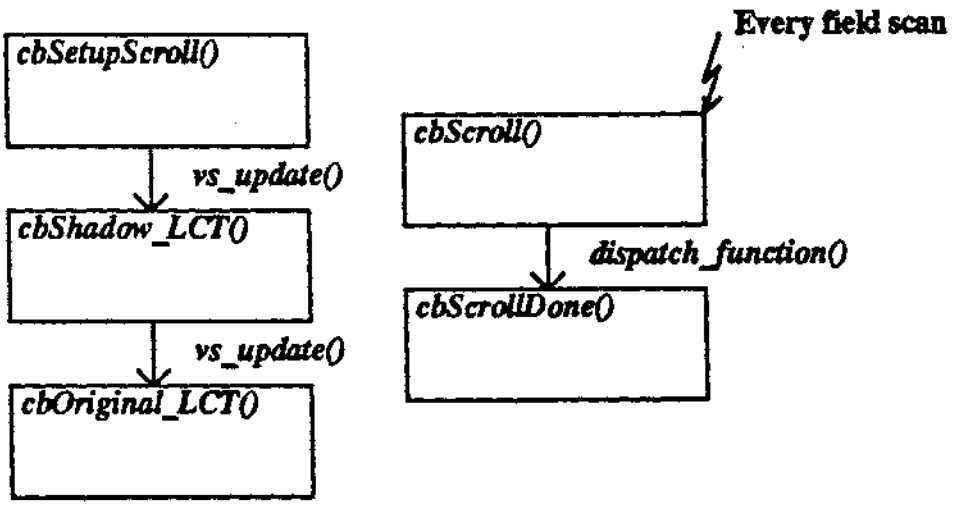


Figure 2 Calling sequence for Scrolling example

The first 3 of these handle the set-up of the effect, including starting a video interrupt using ***vi_add()***. This video interrupt calls ***cbScroll()*** once per field while the effect is running. At the end of the effect, ***cbScroll()*** puts ***cbScrollDone()*** into the **Balboa** dispatcher using ***dispatch_function()***.

2.2.1. Set-up of the scroll environment - *cbSetupScroll()*

The image is scrolled both horizontally and vertically as well as diagonally. The order and distances to travel are stored in an array called: ***Itinerary []***.
Each time a new entry of this array is read and interpreted a new video effect (VFX) is started. If the end of this array is reached it is started again from the beginning only with a higher speed.
The index in array ***Itinerary []*** is the variable ***step***. If ***step*** has stepped through ***Itinerary []***, it is set to zero again and the speed variable ***multiplier*** is incremented, in this example with two. If the latter reaches the value ***MAXIMUM_SPEED*** ***dispatch_quit()*** is called, which in turn dispatches ***cbQuitScroll()***. At this stage the scrolling application is aborted.

2.2.1.1. Timing

The timing and positioning of the scroll is controlled by a number of global variables: (x, y) positions (*PositionPer1Kfields*), (x, y) increments per field (*dX_per_1Kfields, dY_per_1Kfields*) and (x, y) increments, number of fields per video effect (*dX_per_vfx, dY_per_vfx, FieldsPerVFX*, respectively.).
The globally accessible variables are:

```
typedef struct
{
     int     x;
     int     y;
}COORD;

COORD
     PositionPer1Kfields;     /* Contains x,y position after
                               1000 fields */
int
     dX_per_1Kfields,  /* x inc. per (after) 1000 fields */
     dY_per_1Kfields,  /* y inc. per (after) 1000 fields */
     FieldsPerVFX,     /* number of fields before effect is
                       finished */
     dX_per_vfx,       /* x displacement per effect */
     dY_per_vfx;       /* y displacement per effect */
```

In order to handle fractional pixel increments per field and to avoid floating point arithmetic, the first four of these are stored as binary fractions with 10 fractional bits below the binary point.
Another way of looking at this is to calculate positions and increments as they would appear as a thousand fields have been scanned by the video hardware.
For this reason these variables (integers) are shifted to the left 10 positions

```
#define PER_THOUSAND(x)        (x << 10)
```

approximately multiplying it with 1000. Of course they should be brought back to normal proportions

```
#define PER_ONE(mx)            (mx >> 10)
```

the moment we need these variables to update our real position in the image.

The first action within the main *cbSetupScroll()* function is to translate the scroll speed parameters from pixels per second *(PIXELS_PER_SECOND)* into the increments in pixels per field *(PIXELS_PER_FIELD)*, taking account of the video display type of the player in question.
This is done by macro's:

```
#define PIXELS_PER_SECOND     75    /* base scroll velocity
                                    */
#define FIELDS_PER_SECOND     vm_display_freq
#define PIXELS_PER_FIELD      PIXELS_PER_SECOND /\
                              FIELDS_PER_SECOND
```

By changing the way these are calculated, less restrictive forms of scrolling are possible at the risk of not being smooth.

The second thing *cbSetupScroll()* is to do is to calculate some local variables in order to set up the global ones. The local variables are:

```
void cbSetupScroll ()
{
      static int
            multiplier  = 2;  /* speed multiplier */
      static int
            step = -1;        /* the current test number */
      int
            x_velocity,       /* velocity in x direction */
            y_velocity,       /* velocity in y direction */
            ms_per_vfx,       /* milli-seconds per video
                              effect */
            pixels_per_vfx;   /* vfx :: video effect */
```

The static variables' *multiplier* and *step* have already been explained in paragraph 2.2.

The variable *ms_per_vfx* indicates in milli-seconds how long the video effect (i.e. one complete scroll horizontally, vertically or diagonally) is going to last:

```
/* milli-seconds per video effect */
ms_per_vfx =
      PER_THOUSAND((Itinerary[step]).distance) /
                  ( multiplier * PIXELS_PER_SECOND );
```

NOTE:

Since we are working with macro's and integer arithmetic it is crucial to perform multiplication over division, defying the law of equal precedence, for the sake of accuracy. So mind the order and the brackets.

This then gets converted into the number of video fields *(FieldsPerVFX)* it takes to do the effect, which in turn is then used as a main counter (in *cbScroll()*) to determine when the effect is finished.

```
/* Nr. of displayed fields during 1 video effect */
FieldsPerVFX = PER_ONE(ms_per_vfx *
                              FIELDS_PER_SECOND);
```

The variables *x_velocity* and *y_velocity* are the factors used to multiply *pixels_per_vfx* with, in order to calculate how much the picture is to move in pixels, both in x and y directions

```
/* work out where we will end up in x & y direction per
video effect */

dX_per_vfx = x_velocity * pixels_per_vfx;
dY_per_vfx = y_velocity * pixels_per_vfx;
```

where

```
/* resulting in a displacement of pixels per video effect */

pixels_per_vfx = FieldsPerVFX * PIXELS_PER_FIELD;
```

2.2.1.2. Video Synchronisation

This example uses the **Balboa** video interrupt manager as its basic source of timing information. A video interrupt call-back is set-up using *vi_add()* during *cbSetupScroll()*. This call-back should eventually call the step function *cbScroll()* once per field until the end of the effect when it is removed using *vi_remove()* from within *cbScrollDone()*.

```
VIDEO_INT
     *VideoInterrupt;  /* video interrupt for timing */
VIDENV
     *VE;              /* video environment */

     /* set-up the video interrupt we will be using for
     timing */
     VideoInterrupt =
          vi_add
          (
               VE,
               VE->ve_screenstart+2,    /* if display scan
                                        reaches this line,
                                        then */
               pxPicture->pi_plane,
               0,                       /* at every
                                        display scan */
               0,                       /* Continuously
                                        call: */
               NULL,                    /* nothing */
               &PositionPerlKfields,    /* until told
                                        otherwise */
               DISPATCHED
          );
```

This example uses two calls to *vs_update()* to force the current and alternate set of LCTs of the video screen to be the same.
For this reason, the call to *vi_add()* must be in *cbSetupScroll()* before the two *vs_update()* calls so that the video interrupt instruction is copied into both LCTs.
If this parameter of *vi_add()* would have been set to *cbScroll()*, this would have resulted in call-backs to *cbScroll()* after the first call to *vs_update()*, before the rest of the effect has been properly set-up.

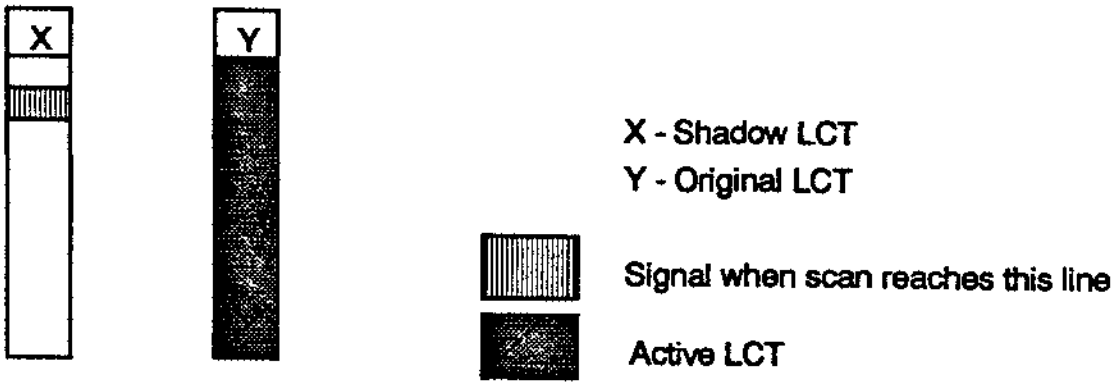
To avoid the effect starting early, a **NULL** pointer is used to defer the actual effect. This then is set to ***cbScroll()*** at the end of ***cbOriginal_LCT()*** when the effect is ready to start, and to **NULL** again when the effect is finished.
As a parameter to ***cbScroll()***, ***PositionPer1Kfields*** is passed.

2.2.1.3. Video Set-up and Initialisation

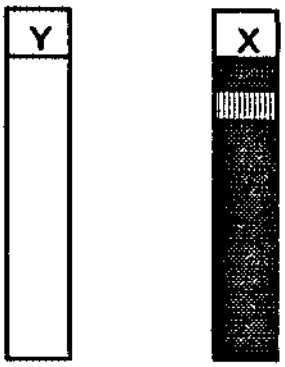
As discussed previously, this example makes use of double buffered LCTs by using the alternate set from a video screen. In order that the only change, as the image scrolls, is the position of the image, the effect must start by making the two sets of LCTs identical. This could be done by reading in one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

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

Before vs_update in cbSetupScroll()



After vs_update in cbSetupScroll()



After the last vs_update in cbShadow_LCT()

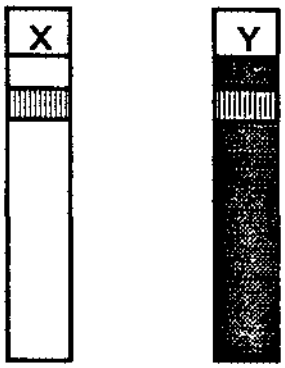


Figure 3 LCT allocation after vs_update ()

```
        .
        .
        /* do the first vs_update to get the first set of LCTs
        correctly set-up */
        vs_update_done
        (
              vsScreen,
              cbShadow_LCT,
              LCT_Shadow,
              DISPATCHED
        );
        vs_update
        (
              vsScreen,
              NON_INTERLACE,
              DISPLAY_625,
              NONE, NONE
        );
        /* After this vs_update() cbScroll is called every
        field, but since the video interrupt call-back is set
        to zero an immediate return is forced.
        'VideoInterrupt' is attached here to one LCT only */
}
```

2.2.2. Set-up of the alternate LCT - *cbShadow_LCT()*

The only thing done in this function is to retrieve the necessary information from the LCTs and FCT. This is done through a call to *FindLctInUse()* (see paragraph 2.2.4.), where the first parameter indicates the line on the screen, so *FindLctInUse()* can figure out which LCT is responsible for this line.

```
void        cbShadow_LCT(context, lct)
int         context;
LCT_INFO    *lct;
{
      /* extract the various ids and things from the first
      vs_update(), these become the alternate set of LCTs.
      */

      FindLctInUse(0, lct);    /* find lct responsible for
                               line 0 on screen */
```

```
        /* now do the second vs_update to get the other set of
        lcts setup */
        vs_update_done
        (
              vsScreen,
              cbOriginal_LCT,
              LCT_Original,
              DISPATCHED
        );
        vs_update
        (
              vsScreen,
              NON_INTERLACE,
              DISPLAY_625,
              NONE, NONE
        );
        /* This vs_update() has made a 1 to 1 copy of the
        Shadow LCT, so now both LCT's are the same. */
}
```

2.2.3. Set-up of the original LCT - *cbOriginal_LCT()*

Also this function, of course, first calls *FindLctInUse()* (see paragraph 2.2.4.)to get the details of the other LCT.

```
void        cbOriginal_LCT(context, lct)
int         context;
LCT_INFO    *lct;
{
/* extract the ids and things from the second vs_update
these are the primary set of LCTs and hence go into element
zero of the three arrays - pLCT[], LCTlines[] and LCTid[] */

      FindLctInUse(0, lct);    /* find lct responsible for
                               line 0 on screen */

      LCT_Current = lct;       /* the currently displayed LCT
                               */

      /* extract the fct id we are going to link to */
      DCP.fct_id =
            (pxPicture->pi_plane == PLANE_B
                               ? VE->ve_fct_b
                               : VE->ve_fct_a);
```

The last two actions to do is to set the current position of our picture in *PositionPer1Kfields*, through the function *SetUpPos ()* (see paragraph 2.2.5.) and to start the actual scrolling by setting the video interrupt call-back routine to *cbScroll()*.

```
        /* get the current starting positions for the
        horizontal  & vertical aspect of the scroll.  */
        SetUpPos (&PositionPer1Kfields);

        /* set the 'running' flag to start the call-backs
        actually doing something */
        VideoInterrupt->vi_cb.function = cbScroll;
}
```

2.2.4. Retrieval of DCP information - *FindLctInUse()*

The array that *FindLctInUse()* stores its information in, contain the LCT details, necessary for the load display start address instructions in function *SetUpDCP()*.

```
typedef struct  tDCP_INFO
{
     int          fct_id;
     LCT_INFO     lct[2];
} DCP_INFO;

/* the inputs for dc_flnk */
DCP_INFO          DCP;          /* the two LCTs and FCT we are
                                   working with */


typedef struct  tLCT_INFO
{
           int   id;
           int   line;
           int   linecount;
}LCT_INFO;

#define NOT_ACTIVE_LCT        1
#define ACTIVE_LCT            0

LCT_INFO
     *LCT_Original = &(DCP.lct[ACTIVE_LCT]),
     *LCT_Shadow   = &(DCP.lct[NOT_ACTIVE_LCT]),
     *LCT_Current;
```

This function (*FindLctInUse()*) can be written in several ways. Two ways are given here, but both solutions find:

- which display segment the scroll is to happen within
- the height in scan lines of that display segment (which is the number of scan lines to write into the **LCT**)
- the line within the **PICTURE** displayed on the first line of the display segment
- the **LCT** id, necessary for any **CDRTOS** call pertaining **LCT**'s.

The first solution is mainly meant to get more insight how **Balboa**'s video manager maintains its structures and how it can be used to obtain the necessary information.
The display segment within which the effect is to run can be found by looping over all the display segments looking for the first one to contain the picture to scroll. To avoid bus traps, if none is found an error is reported to the status manager and the function gives up and returns.

```
void        FindLctInUse (lct, picture, screen)
LCT_INFO    *lct;
PICTURE     *picture;
VS          *screen;
{
     DSEG   *ds;
     LCT    *p_LCT;

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

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

```
/* check if we actually found something */
     if( ! ds )
     {
          STATUS(ST_APPL,ST_ERROR,0,"no display segment
                                     found for scroll");
          return;
     }
```

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

```
     /* save the LCT pointer and line */
     if( pxPicture->pi_plane == PLANE_B)
     {
          p_LCT = ds->ds_lct1;
          lct->line  = ds->ds_line1;     /* LCT line
                                         responsible for
                                         screen_line */
     }
     else
     {
          p_LCT = ds->ds_lct0;
          lct->line  = ds->ds_line0;     /* LCT line
                                         responsible for
                                         screen_line */
     }
```

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

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

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

The FCT id for the *dc_flnk()* call can be found from the *ve_fct_a* or *ve_fct_b* entries in the video environment structure.

```
if( picture->pi_plane )
        DCP->fct_id= vsScreen->vs_videnv->ve_fct_b;
else
        DCP->fct_id= vsScreen->vs_videnv->ve_fct_a;
```

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

```
        lct->linecount = UCM2PIXEL(ds->ds_nlines);
        return;
}
```

The alternative way is given below and is more concise.

```
void        FindLctInUse (screen_line, lct)
int         screen_line;
LCT_INFO    *lct;
{
      DSEG  *ds;
      LCT   *p_LCT;

      ds = vs_find_segment (vsScreen, screen_line);

      /* save the LCT pointer and line */
      if( pxPicture->pi_plane == PLANE_B)
      {
            p_LCT = ds->ds_lct1;
            lct->line  = ds->ds_line1;
            /* LCT line responsible for screen_line */
      }
      else
      {
            p_LCT = ds->ds_lct0;
            lct->line  = ds->ds_line0;
            /* LCT line responsible for screen_line */
      }
      lct->linecount = UCH2PIXEL(ds->ds_nlines);

      /* get the LCT id to use for dc_flnk() */
      lct->id = p_LCT->lc_id;
      return;
}
```

2.2.5. Dead reckoning - *SetUpPos ()*

One of the position variables that has not been discussed is x and y position as in *PositionPer1KFields*, the initial x and y position of the image on the screen. For x this is taken from the *pi_viewport* entry in the **PICTURE** structure and then converted from UCM co-ordinates to our binary fraction format using a shift left of 10 places.
The line within the picture displayed (y position) at the top of the display segment can be found with a simple subtraction.

```
void        SetUpPos (pos)
COORD       *pos;
{
      DSEG     *ds;

      ds = vs_find_segment(vsScreen, pos->y);

      /* get the number of lines from the display segment
      and convert this from UCM co-ordinates into pixel ones
      since everything we are working with is normal
      resolution. Since we are working with a system that
      reflects the screen after a 1000 events we need to
      adjust for this.
      */

      pos->x =
            UCM2PIXEL(PER_THOUSAND( pxPicture ->
                                    pi_viewport.ul.x));
      pos->y =
            UCM2PIXEL(PER_THOUSAND( ds->ds_screenline -
                                    pxPicture->pi_line ));

      return;
}
```

2.3. Scrolling

As described just above, a global variable *VideoInterrupt->vi_cb.function* is used to control whether the effect is ready to start. The step function *cbScroll()* is called every field scan, i.e. in PAL every 20 ms, NTSC every 16.67 ms.
The purpose of this function is to update the new position and make the video logic aware of this.

2.3.1. The step function - *cbScroll()*

```
void cbScroll(context, position_after_1Kfields)
int         context;
register
      COORD *position_after_1Kfields;
{
      /* are we finished ? */
      if(FieldsPerVFX == 0)            /* YES */
      {
            /* see paragraph 2.4.*/
      }
      FieldsPerVFX--;
```

The first code in the step function handles the termination of the effect: it checks if we already had *FieldsPerVFX* signals from the video hardware.
This is discussed separately in paragraph 2.4. later in this document. If this variable *FieldsPerVFX* is not zero, it is decremented.
Following that the variables, which control the display and motion of the image are incremented, ready for the next step.

```
      /* apply the increment to the position variables */
      position_after_1Kfields->x += dX_per_1Kfields;
      position_after_1Kfields->y += dY_per_1Kfields;
```

After this a call is made to *SetUpDCP()* (see paragraph 2.3.2.) to effectuate the previously made adjustments.
Of course since we work with variables reflecting the situation after every 1000 fields, an adjustment i.e. shift to the right, need to be made in order to make the correct adjustments for the LCT and FCT (DCP).

```
      SetUpDCP
```

```
        (
                PER_ONE(position_after_lKfields->x),
                PER_ONE(position_after_lKfields->y)
        );
        return;
}
```

2.3.2. The DCP update function - *SetUpDCP()*

In preparation for the *'for"* loop, which will generate the load display start address instructions, the input variables to ***SetUpDCP()***, which define the position of the image must be the converted variables (***xpos***, ***ypos***) from our binary fraction format.
The new **x** position (***xpos***) becomes a variable called ***pixel_x_offset*** and the ***lstart*** is used as an offset into the line start table of the **PICTURE** object, held in the ***pi_lstart*** entry of the **PICTURE** structure.

```
void SetUpDCP (xpos, ypos)
register    int     xpos;
register    int     ypos;
{
     register int
          *lstart =    (int*)(pxPicture->pi_lstart) + ypos;
     register int
           line;
     register int
           pixel_x_offset  = xpos;
```

Now we have all the inputs for the loop to create the instructions.

2.3.2.1. Temporary Workspace

The fastest way to write the load display start address instructions to the **LCT** is to write them all at once using the **CDRTOS** call ***dc_pwrlct()***.
To do this a buffer will be needed to hold the instructions prior to writing. For a full screen image on a **PAL** player, this buffer will need to be at least 280 scan lines * 4 bytes = 1120 bytes.

In function *SetUpDcp()* it is defined as follows:

```
long  lct_column[PAL_HEIGHT]; /* Take the biggest
                                 picture with respect
                                 to height */
```

The loop to generate the instructions uses the *cp_dadr()* macro from the CDRTOS header file **ucm.h** to translate an address into the load display start address LCT instruction for that address.

```
/* loop for each line we have to write, building the
LCT   instructions */
for (line = 0; line < LCT_Current->linecount; line++)
      lct_column[line] =
            cp_dadr( lstart[line] + pixel_x_offset );
```

2.3.2.2. Updating the video hardware

The CDRTOS call *dc_pwrlct()* is used to write the instructions just generated into the LCT. Before we can do this though, we need to switch the LCT, otherwise we would write in the LCT being displayed; this would result in glitches on the screen if the video hardware would collide with the CPU, writing the new LCT instructions.

```
/* switch our flag to a new LCT */
LCT_Current =
      (LCT_Current == LCT_Original  ? LCT_Shadow
                                    : LCT_Original);
```

For normal resolution LCTs, *dc_pwrlct()* is a lot faster than the more conventional *dc_wrlct()*. Most of the parameters required for this call were found by *FindLctInUse()* in paragraph 2.2.4..

For the column number parameter, *DSEG_INS_DADR* must be used since that is the column number which *vs_update()* will have written its load display start address instructions to.

```
/* now write the instructions we just built */
dc_pwrlct
(
      vm_vidpath,              /* path to video device
                               */
      LCT_Current->id,         /* LCT id */
      LCT_Current->line,       /* Lct line number to
                               begin write */
      DSEG_INS_DADR,           /* LCT column number to
                               begin write */
      LCT_Current->linecount,  /* physical nr of lines
                               to write */
      1,                       /* physical nr of
                               columns to write */
      lct_column               /* pointer to buffer of
                               data to write */
);
```

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

```
/* switch (DCP.lct) via dc_flnk and show new situation
*/
dc_flnk
(
      vm_vidpath,        /* path to video device */
      DCP.fct_id,        /* FCT id */
      LCT_Current->id,   /* LCT id */
      0                  /* Lct line number to link to
                         */
);
return;
}
```

2.4. Terminating the Video Effect

To terminate correctly we have to check that if we give the control back to **Balboa**, everything should be such a state that **Balboa** can continue without problems.
One of the things to check is to see if we end up with the same **LCT** as with which we started. Also since on the screen we see at the end something different than with which we started we need to reflect this in **Balboas'** data structures.

2.4.1. Deactivating the interrupts

The global variable ***FieldsPerVFX*** was set-up before the scroll started to be the number of field periods for which the effect was to run. This variable is tested whether the end of the effect has been reached.

```
/* are we finished ? */
if(FieldsPerVFX == 0)          /* YES */
{
      /* end of scroll so clear 'running' flag and
      call the termination routine */

      VideoInterrupt->vi_cb.function = NULL;
      dispatch_function
      (
            0,
            cbScrollDone,
            cbSetupScroll
      );
```

If it is zero the last signal has been received and in order to stop the signals from coming (i.e. calling ***cbScroll()***), the video interrupt call-back routine is set to **NULL**.
At the same time we dispatch ***cbScrollDone*** with parameter ***cbSetupScroll***.

Normally we could end here the scroll operation, if it was not for a nasty side effect for which we need to check. The problem is that when we took over the **LCT's** from **Balboa**, **Balboa** at this point still expects that the very first **LCT** we started with to be the current one. If, however ***FieldsPerVFX*** happens to be an odd number we will end up with the alternate **LCT**.
We need to check this and eventually switch back to the old **LCT**.

```
        /* are we using the right (DCP.lct) or not ? */
        if( LCT_Current == LCT_Shadow )/*The wrong one*/
        {
              /* we want to switch to the correct set of
              (DCP.lct) without applying any further
              position increment.   */
              SetUpDCP
              (
                    PER_ONE(position_after_IKfields->x),
                    PER_ONE(position_after_IKfields->y)
              );
        }
        return;
}
```

2.4.2. Non time critical adjustments - *cbScrollDone()*

The function ***cbScrollDone()*** performs the non time critical aspects of the termination of the effect.
There are two parts to this:

- adjusting the **PICTURE** structure to match the section of the image, which is now being displayed.
- kicking off the next video effect (scroll).

What is needed is to move the viewport of the **PICTURE** the same distance as the image moved during the scroll.
This distance is computed during the set-up for the effect and held in two global variables, ***dX_per_vfx*** and ***dY_per_vfx***.
The division is by 2 since the inputs to ***pi_viewport()*** are in **UCM** co-ordinates not pixel ones.
The last parameter to ***pi_viewport()*** defines where on the screen the top of the viewport should appear. In this case we have scrolled the image within that portion of the screen it contributed to before and so we need the top line of the viewport to be displayed in the same place as it was before.

```
void cbScrollDone (context, nxtfun)
int  context;
void (*nxtfun) ();
{
     /* set the viewport of the picture to match where we
     now   are on the screen */

     pi_viewport
     (
          pxPicture,
          pxPicture->pi_viewport.ul.x +
                              PIXEL2UCN(dX_per_vfx),
          pxPicture->pi_viewport.ul.y +
                              PIXEL2UCN(dY_per_vfx),
          pxPicture->pi_viewport.lr.x +
                              PIXEL2UCN(dX_per_vfx),
          pxPicture->pi_viewport.lr.y +
                              PIXEL2UCN(dY_per_vfx),
          pxPicture->pi_viewport.ul.y +
                              pxPicture->pi_line
     );
```

The very last action of the effect is to generate the call-back to the application that the effect is finished and that eventually a new effect can start. The address of this function was passed as a parameter to *cbScrollDone()*. Also the video interrupt is removed.

```
     /* remove video interrupts */
     vi_remove(VE, VideoInterrupt);

     /* generate the call-back to the rest of the
     application */
     dispatch_function(MA, nxtfun, MA);
}
```

2.4.3. The final call-back - cbQuitScroll()

If in *cbSetupScroll() multiplier* has reached the value *MAXIMUM_SPEED*, *dipatch_quit()* is called.
Since during initialisation in *StartScroll()* the dispatcher is told to dispatch *cbQuitScroll()* if *dispatch_quit()* is called the call-back below will be dispatched and terminate the complete application.

```
void cbQuitScroll(context, pm)
int  context;
void *pm;
{
     vs_finish();
}
```

2.5. The Demonstration Program

The example program listed in Appendix 1 shows vertical, horizontal and diagonal scrolls at a variety of speeds.
In an overview is given of all the call-backs and how they are activated.

Appendix 1 Listing of Scroll Program

```
/*
 Function    :    c:\balboa_1.3\scroll\scroll.c
 Author      :    Jan Rolff
 Date        :    17-Jun-1993
 Purpose     :    To obtain insight in the Video Manager
 Description:    This programs scrolls a CLUT picture in
                 horisontal, vertical and diagonal
                 direction.
                 It is based on a program written by Jon
                 Piesing.

History

Date              By                Reason
----------------+-----------------+-------------------
   Mar-1993       Jon Piesing       Creation
17-Jun-1993       Jan Rolff         Brush up
*/


/* $dcif$ Include files */

/* NON-BALBOA include files */
#include    <errno.h>
#include    <modes.h>
#include    <stdio.h>
#include    <signal.h>
#include    <ucm.h>

/* BALBOA include files */
#define     BP_DEBUG

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

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

/* define the various parameters for this program */
$define IMAGE_FILE      "RTF/clut8.scroll"

$define IMAGE_WIDTH     768   /* double screen width image
                                 in pixels */
$define IMAGE_HEIGHT    576   /* double screen height image
                                 in scan lines */
$define IMAGE_BYTES     IMAGE_WIDTH * IMAGE_HEIGHT

$define PAL_WIDTH       384   /* PAL picture's width in
                                 pixels */
$define PAL_HEIGHT      280   /* PAL picture's height in
                                 scan  lines */
$define PAL_BYTESPAL_   WIDTH * PAL_HEIGHT

$define MAX_INTERNAL_BUF      1     /* Max # of internal
                                       buffers */
$define MAX_DSEG              5     /* Max # of DSEG's for
                                       picture build up  */
$define MAX_TCMAP             0     /* Max # of TCMAPs for
                                       clut manager */
$define MAX_INTERRUPT_LINES   2     /* Max # of scan line
                                       interrupts */
$define MAX_INTERRUPT_FUNCS   4     /* Max # of video
                                       interrupt callbacks */
$define ARBITRARY_SIZE        4096  /* any conveniently big
                                       size */

/* FORM2 Constants and Macro's */
$define F2_SECTOR_BYTES       (2324)      /* form 2 sector
                                             size */
$define F2_SECTOR_COUNT(p)    (((F2_SECTOR_BYTES - ((p) %\
                              F2_SECTOR_BYTES))\ + (p)) /\
                              F2_SECTOR_BYTES)
$define F2_TOTAL_BYTES(s)     (F2_SECTOR_COUNT((s)) *\
                              F2_SECTOR_BYTES)

$define NA              0           /* Not Applicable */
$define NONE            0
$define PIXEL2UCM(x)    (x<<1)      /* Convert to UCM  */
$define UCM2PIXEL(x)    (x>>1)      /* Convert to pixels */
$define CHANNEL(c)      c           /* Trivial, for
                                       readability only */
```

```
/* Scroll specific #define's */
#define TOTAL_STEPS(n) ELEMENTS(n) /* number of tests for
                                   each speed */
#define MAXIMUM_SPEED  32          /* maximum speed
                                   multiplier */
#define ELEMENTS(n)    (sizeof(n)  / sizeof (*n))
#define PIXELS_PER_SECOND    75    /* base scroll velocity
                                   */
#define PER_THOUSAND(s)      (s << 10)
#define PER_ONE(ms)          (ms >>10)
#define FIELDS_PER_SECOND    vm_display_freq
#define PIXELS_PER_FIELD     PIXELS_PER_SECOND /\
                             FIELDS_PER_SECOND
#define NOT_ACTIVE_LCT       1
#define ACTIVE_LCT           0

/*                           x,  y    */
#define NORTH                0, -1
#define NORTH_EAST           1, -1
#define EAST                 1,  0
#define SOUTH_EAST           1,  1
#define SOUTH                0,  1
#define SOUTH_WEST           -1, 1
#define WEST                 -1, 0
#define NORTH_WEST           -1, -1
```

```
typedef struct tITINERARY
{
     POINT   direction;
     int     distance;
}ITINERARY;

typedef struct  tLCT_INFO
{
          int   id;
          int   line;
          int   linecount;
}LCT_INFO;

typedef struct  tDCP_INFO
{
     int          fct_id;
     LCT_INFO     lct[2];
} DCP_INFO;

typedef struct
{
     int     x;
     int     y;
}COORD;

/* $dcff$ Declaration of Forward functions */
void
     SetUpPos(),
     SetUpDCP(),
     InitPictMngr(),
     InitVidMngr(),
     InitPlayMngr(),
     InitSigMngr(),
     StartScroll(),
     cbQuitScroll(),
     cbEndOfPlay(),
     dispatch_quit(),  /* Not fwd declared in dispatch.h */
     cbSetupScroll(),
     cbScroll(),
     cbScrollDone(),
     cbShadow_LCT(),
     cbOriginal_LCT(),
     FindLctInUse();
```

```
/* $dcgd$ Declaration of Global data */

/* $dfgd$ Definition  of Global data */

/* define the directions and the distances (durations) for
each of the tests */

ITINERARY   Itinerary[] =
{
     {{ EAST },          PAL_WIDTH },
     {{ SOUTH },         PAL_HEIGHT},
     {{ NORTH_WEST },    PAL_HEIGHT},
     {{ SOUTH },         PAL_HEIGHT},
     {{ NORTH_EAST },    PAL_HEIGHT},
     {{ WEST},           PAL_WIDTH },
     {{ SOUTH },         PAL_HEIGHT},
     {{ EAST},           PAL_WIDTH },
     {{ NORTH },         PAL_HEIGHT},
     {{ WEST},           PAL_WIDTH },
     {{ SOUTH_EAST },    PAL_HEIGHT},
     {{ NORTH },         PAL_HEIGHT},
     {{ SOUTH_WEST },    PAL_HEIGHT},
     {{ EAST},           PAL_WIDTH },
     {{ NORTH },         PAL_HEIGHT},
     {{ WEST},           PAL_WIDTH },
};

PICTURE
     *pxPicture;         /* the full screen picture    */
VS
     *vsScreen;          /* the video screen   */
char
     PLTE_Buffer[F2_SECTOR_BYTES]; /* a buffer to hold the
                                      clut data */
VIDEO_INT
     *VideoInterrupt;    /* video interrupt for timing */
VIDENV
     *VE;                /* video environment */
```

```
/* the inputs for dc_flnk */
DCP_INFO
     DCP;                  /* the two LCTs and PCT we are
                           working with */
LCT_INFO
     *LCT_Original = &(DCP.lct[ACTIVE_LCT]),
     *LCT_Shadow   = &(DCP.lct[NOT_ACTIVE_LCT]),
     *LCT_Current;

/* timing and control variables */
int
     dX_per_1Kfields,  /* x inc. per (after) 1000 fields */
     dY_per_1Kfields,  /* y inc. per (after) 1000 fields */
     FieldsPerVFX,     /* number of fields before effect is
                       finished */
     dX_per_vfx,       /* x displacement per effect */
     dY_per_vfx;       /* y displacement per effect */
COORD
     PositionPer1Kfields;    /* Contains x,y position after
                             1000 fields */

/* $dfld$ Definition  of Local data */

/* $dfgf$ Definition  of Global functions */

int main()
{
     dispatch_loop (StartScroll, NA, NA );
}
```

```
void StartScroll()
{
      static int          fd;    /* the OS9 file path as
                                    returned by open() */
      static CALLBACK     cb =
            {
                  cbSetupScroll,     /* The first cb to
                                        execute after assets are
                                        read in from disc */
                  &fd                /* Pointer to file to
                                        open/close */
            };

      dispatch_atquit(cbQuitScroll, MA);
      STATUS_INIT          /* status manager initialisation */
      (
            stderr,
            ST_MGR_ALL,
            ST_TYP_ALL,
            0
      );
      InitSigMngr ();    /* signal manager initialisation */
      InitVidMngr ();    /* video manager initialisations */
      InitPictMngr ();   /* picture manager initialisation */
      InitPlayMngr ();   /* play manager initialisations */

      /* now go and do the play */
      fd = open (IMAGE_FILE, S_IREAD);
      pml_play
      (
            fd,    /* file path */
            0,     /* position */
            1,     /* Potentially active mask: channel 0 & 1
                      */
            0,     /* direct_audio_mask */
            1,     /* nr. of EOR to mark End of Play */
            0,     /* channels to be swiched between
                      active/de-active */
            cbEndOfPlay, &cb,        /* Dispatch function
                                        when  End of Play */
            NULL
       );
}
```

```
void InitSigMngr()
{
      sgm_init();       /* initialise signal manager, needed
                        for play manager */
      sgm_enable
      (
            SIGQUIT,
            dispatch_quit,
            SIGQUIT,
            DISPATCHED
      );
      sgm_enable
      (
            SIGINT,
            dispatch_quit,
            SIGINT,
            IMMEDIATE
      );
      return;
}

void InitVidMngr()
{
      vs_init (BP_MEM_DONTCARE, MAX_INTERNAL_BUF);
      vsScreen =
            vs_open
            (
                  LCT_DOUBLE|CURSOR_ON,
                  MAX_DSEG,
                  BP_MEM_DONTCARE
            );

      VE = vsScreen->vs_videnv;
      cl_init (vsScreen, BP_MEM_DONTCARE, MAX_TCMAP );
      vi_init
      (
            VE,
            BP_MEM_DONTCARE,
            MAX_INTERRUPT_LINES,
            MAX_INTERRUPT_FUNCS
      );
      return;
}
```

```
void InitPictMngr()
{
    pxPicture =
        pi_create
        (
            PLANE_A,
            D_CLUT8,
            PIXEL2UCN(IMAGE_WIDTH),
            PIXEL2UCN(IMAGE_HEIGHT),
            F2_TOTAL_BYTES(IMAGE_BYTES),
            NA
        );
    return;
}
```

```
void InitPlayMngr()
{
     pmb_set_block
     (
          (char*)
          bp_allocate
          (
               ARBITRARY_SIZE,
               BP_MEM_DONTCARE
          ),
          ARBITRARY_SIZE
     );
     pml_init();

/* allocate the memory for the input buffers */
     pml_add_buffer          /* the picture */
     (
          CHANNEL(0),
          VIDEO_TYPE,
          F2_SECTOR_COUNT(IMAGE_BYTES),
          pxPicture->pi_pstart,
          NA, NA,
          DISPATCHED
     );
     pml_add_buffer          /* and its CLUT table */
     (
          CHANNEL(0),
          DATA_TYPE,
          F2_SECTOR_COUNT(F2_SECTOR_BYTES),
          PLTE_Buffer,
          NA, NA,
          DISPATCHED
     );
     return;
}
```

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

void     FindLctInUse (screen_line, lct)
int             screen_line;
LCT_INFO    *lct;
{
     DSEG  *ds;
     LCT   *p_LCT;

     ds = vs_find_segment (vsScreen, screen_line);

     /* save the LCT pointer and line */
     if( pxPicture->pi_plane == PLANE_B)
     {
          p_LCT = ds->ds_lct1;
          lct->line  = ds->ds_line1;    /* LCT line
                                           responsible for
                                           screen_line */
     }
     else
     {
          p_LCT = ds->ds_lct0;
          lct->line  = ds->ds_line0;    /* LCT line
                                           responsible for
                                           screen_line */
     }
     lct->linecount = UCM2PIXEL(ds->ds_nlines);

     /* get the LCT id to use for dc_flnk() */
     lct->id = p_LCT->lc_id;
     return;
}
```

```
void SetUpPos (pos)
COORD       *pos;
{
     DSEG     *ds;

     ds = vs_find_segment(vsScreen, pos->y);

     /* get the number of lines from the display segment
     and convert this from UCM coordinates into pixel ones
     since everything we are working with is normal
     resolution. Since we are working with a system that
     reflects the screen after a 1000 events we need to
     adjust for this.
     */

     pos->x =
          UCM2PIXEL(PER_THOUSAND( pxPicture ->
                                  pi_viewport.ul.x));
     pos->y =
          UCM2PIXEL(PER_THOUSAND( ds->ds_screenline -
                                  pxPicture->pi_line ));

     return;
}
```

```
void SetUpDCP (xpos, ypos)
register    int     xpos;
register    int     ypos;
{
     register int
          *lstart =    (int*)(pxPicture->pi_lstart) + ypos;
     register int
           line;
     register int
           pixel_x_offset  = xpos;

     long  lct_column[PAL_HEIGHT]; /* Take the biggest
                                      picture with respect
                                      to height */

     /* switch our flag to a new LCT */
     LCT_Current =
           (LCT_Current == LCT_Original  ? LCT_Shadow
                                         : LCT_Original);

     /* loop for each line we have to write, building the
     LCT instructions */
     for (line = 0; line < LCT_Current->linecount; line++)
           lct_column[line] =
                 cp_dadr( lstart[line] + pixel_x_offset );

     /* now write the instructions we just built */
     dc_pwrlct
     (
           vm_vidpath,              /* path to video device
                                       */
           LCT_Current->id,         /* LCT id */
           LCT_Current->line,       /* Lct line number to
                                       begin write */
           DSEG_INS_DADR,           /* LCT column number to
                                       begin write */
           LCT_Current->linecount, /* physical nr of lines
                                      to write */
           1,                       /* physical nr of
                                       columns to write */
           lct_column               /* pointer to buffer of
                                       data to write */
     );
```

```
    /* switch (DCP.lct) via dc_flnk and show new situation
    */
    dc_flnk
    (
        vm_vidpath,         /* path to video device */
        DCP.fct_id,         /* FCT id */
        LCT_Current->id,    /* LCT id */
        0                   /* Lct line number to link to
                            */
    );
    return;
}

void cbEndOfPlay (context, cb)
int         context;
CALLBACK    *cb;
{
    close (cb->parameter);      /* close rtf file */

    /* now we have loaded our image data and clut table,
    display it  together with the clut */
    pi_front( vsScreen, pxPicture );
    cl_exec ( vsScreen, PLANE_A, (CLUT*)PLTE_Buffer );

    vs_update
    (
        vsScreen,
        NON_INTERLACE,
        DISPLAY_625,
        NA, NA
    );
    vs_show( vsScreen );
    dispatch_function (NA, cb->function, NA);
}
```

```
void cbSetupScroll ()
{
      static int
            multiplier  = 2;  /* speed multiplier */
      static int
            step = -1;        /* the current test number */
      int
            x_velocity,       /* velocity in x direction */
            y_velocity,       /* velocity in y direction */
            ms_per_vfx,       /* milli-seconds per video
                              effect */
            pixels_per_vfx;   /* vfx :: video effect */

      /* stop when we reach the maximum velocity */
      step++;
      if (step >= TOTAL_STEPS(Itinerary))
      {
            multiplier += 2;   /* increase the speed */
            if( multiplier >= MAXIMUM_SPEED)
                  dispatch_quit(0);
            step = 0;
      }

      /* now work out how many steps we can do so that we
      don't run off the end. The base scroll velocity is 75
      pixels/second regardless of PAL or NTSC.
      For PAL player,  the basic increment is:
            75/50 = 1.5       PIXELS_PER_FIELD.
      For NTSC player, the basic increment is:
            75/60 = 1.25 PIXELS_PER_FIELD.
      */

      /* milli-seconds per video effect */
      ms_per_vfx =
            PER_THOUSAND((Itinerary[step]).distance) /
                        ( multiplier * PIXELS_PER_SECOND );


      /* Nr. of displayed fields during 1 video effect */
      FieldsPerVFX = PER_ONE(ms_per_vfx *
                                    FIELDS_PER_SECOND);
```

```
/* resulting in a displacement of pixles per video
effect */
pixels_per_vfx = FieldsPerVFX * PIXELS_PER_FIELD;

/* work out where we will end up in x & y direction
per   video effect */
x_velocity = multiplier *
            (Itinerary[step]).direction.x;
y_velocity = multiplier *
            (Itinerary[step]).direction.y;

dX_per_vfx = x_velocity * pixels_per_vfx;
dY_per_vfx = y_velocity * pixels_per_vfx;

/* translate this to x & y offsets per 1000 fields */
dX_per_1Kfields =
      PER_THOUSAND(dX_per_vfx) / FieldsPerVFX;
dY_per_1Kfields =
      PER_THOUSAND(dY_per_vfx) / FieldsPerVFX;

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

tsleep (256/FIELDS_PER_SECOND);

/* setup the video interrupt we will be using for
timing */
VideoInterrupt =
      vi_add
      (
            VE,
            VE->ve_screenstart+2,   /* if display scan
                                    reaches this line,
                                    then */
            pxPicture->pi_plane,
            0,                      /* at every
                                    display scan */
            0,                      /* Continuously
                                    call: */
            NULL,                   /* nothing */
            &PositionPer1Kfields,   /* until told
                                    otherwise */
            DISPATCHED
      );
```

```
        /* do the first vs_update to get the first set of LCTs
        correctly setup */
        vs_update_done
        (
              vsScreen,
              cbShadow_LCT,
              LCT_Shadow,
              DISPATCHED
        );
        vs_update
        (
              vsScreen,
              NON_INTERLACE,
              DISPLAY_625,
              NONE, NONE
        );
        /* After this vs_update() cbScroll is called every
        field, but since the video interrupt callback is set
        to zero an immediate return is forced.
        'VideoInterrupt' is attached here to one LCT only */
}
```

```
void        cbShadow_LCT(context, lct)
int         context;
LCT_INFO    *lct;
{
     /* extract the various ids and things from the first
     vs_update(), these become the alternate set of LCTs.
     */

     FindLctInUse(0, lct);    /* find lct responsible for
                               line 0 on screen */

     /* now do the second vs_update to get the other set of
     lcts setup */
     vs_update_done
     (
          vsScreen,
          cbOriginal_LCT,
          LCT_Original,
          DISPATCHED
     );
     vs_update
     (
          vsScreen,
          NON_INTERLACE,
          DISPLAY_625,
          NONE, NONE
     );
     /* This vs_update() has made a 1 to 1 copy of the
     Shadow LCT, so now both LCT's are the same. */
}
```

```
void        cbOriginal_LCT(context, lct)
int         context;
LCT_INFO    *lct;
{
/* extract the ids and things from the second vs_update
these are the primary set of LCTs and hence go into element
zero of the three arrays - pLCT[], LCTlines[] and LCTid[] */

     FindLctInUse(0, lct);    /* find lct responsible for
                               line 0 on screen */
     LCT_Current = lct;       /* the currently displayed LCT
                               */

     /* extract the fct id we are going to link to */
     DCP.fct_id =
          (pxPicture->pi_plane == PLANE_B
                               ? VE->ve_fct_b
                               : VE->ve_fct_a);

     /* get the current starting positions for the
     horizontal  & vertical aspect of the scroll.  */
     SetUpPos (&PositionPerlKfields);

     /* set the 'running' flag to start the callbacks
     actually doing something */
     VideoInterrupt->vi_cb.function = cbScroll;
}
```

```
void cbScroll(context, position_after_1Kfields)
int         context;
register
     COORD *position_after_1Kfields;
{
     /* are we finished ? */
     if(FieldsPerVFX == 0)          /* YES */
     {
          /* end of scroll so clear 'running' flag and
          call the termination routine */

          VideoInterrupt->vi_cb.function = NULL;
          dispatch_function
          (
               0,
               cbScrollDone,
               cbSetupScroll
          );

          /* are we using the right (DCP.lct) or not ? */
          if( LCT_Current == LCT_Shadow )/*The wrong one*/
          {
               /* we want to switch to the correct set of
               (DCP.lct) without applying any further
               position increment.   */
               SetUpDCP
               (
                    PER_ONE(position_after_1Kfields->x),
                    PER_ONE(position_after_1Kfields->y)
               );
          }
          return;
     }
     FieldsPerVFX--;

     /* apply the increment to the position variables */
     position_after_1Kfields->x += dX_per_1Kfields;
     position_after_1Kfields->y += dY_per_1Kfields;

     SetUpDCP
     (
          PER_ONE(position_after_1Kfields->x),
          PER_ONE(position_after_1Kfields->y)
     );
     return;
}
```

```
void cbScrollDone (context, nxtfun)
int  context;
void (*nxtfun) ();
{
     /* set the viewport of the picture to match where we
     now are on the screen */

     pi_viewport
     (
          pxPicture,
          pxPicture->pi_viewport.ul.x +
                              PIXEL2UCN(dX_per_vfx),
          pxPicture->pi_viewport.ul.y +
                              PIXEL2UCN(dY_per_vfx),
          pxPicture->pi_viewport.lr.x +
                              PIXEL2UCN(dX_per_vfx),
          pxPicture->pi_viewport.lr.y +
                              PIXEL2UCN(dY_per_vfx),
          pxPicture->pi_viewport.ul.y +
                              pxPicture->pi_line
     );

     /* remove video interrupts */
     vi_remove(VE, VideoInterrupt);

     /* generate the callback to the rest of the
     application */
     dispatch_function(NA, nxtfun, NA);
}

void cbQuitScroll(context, pm)
int  context;
void *pm;
{
     vs_finish();
}

/* $dflf$ Definition  of Local functions */
```

Appendix 2 Overview of callbacks

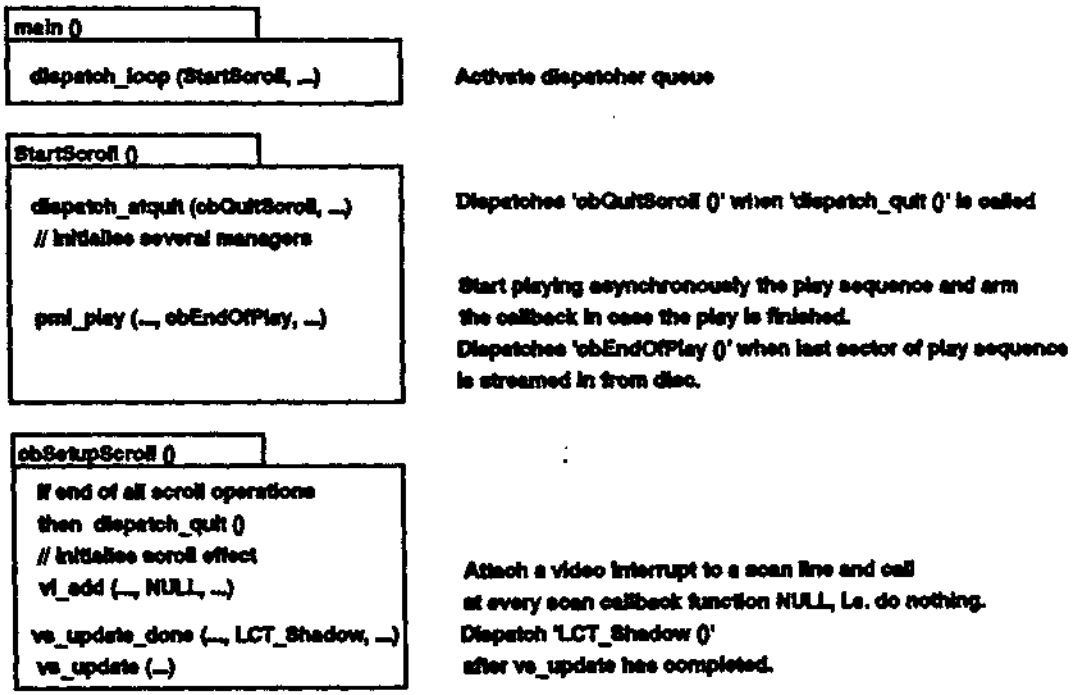


Figure 4 Application invoked Callbacks

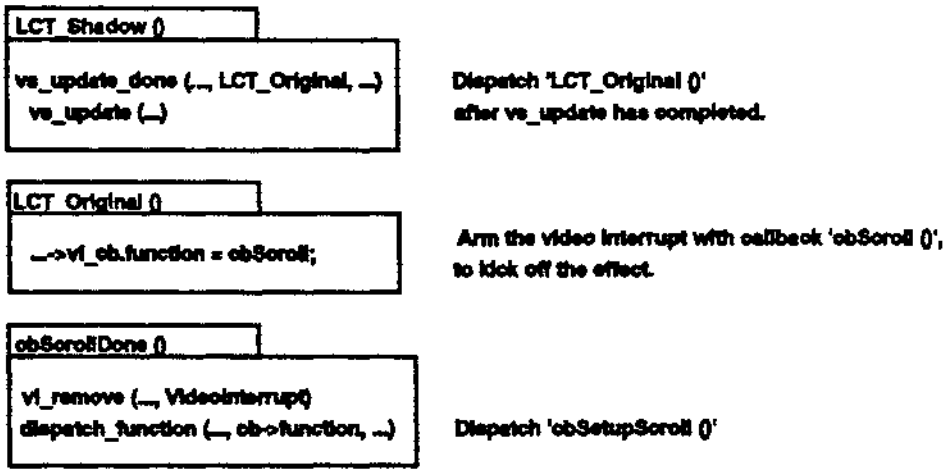


Figure 4 Continued: Application invoked Callbacks

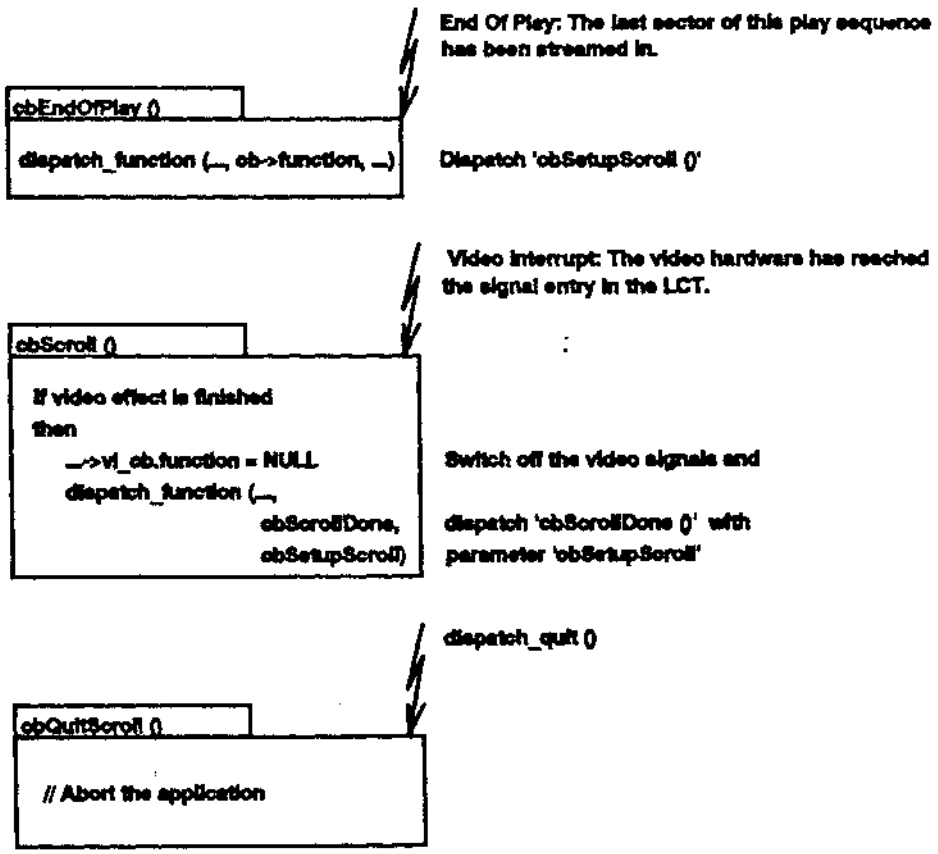


Figure 5 Events and their related Callbacks

I N D E X