PHILIPS

PHILIPS INTERACTIVE MEDIA

Technical Note #89

A New Method for Video Scan Synchronization in CD-i

Jon Piesing
Philips Research Labs, Redhill

January 4, 1994

This technical note describes three methods of video scan synchronization in CD-i, two that are described in the Green Book and one recently discovered by the author. This is of particular significance to highly interactive action games, since the two Green Book methods fall far short of what is available on other platforms.

Copyright © 1994 Philips Interactive Media .
All rights reserved.
This document is not to be duplicated or distributed without permission from Philips Interactive Media.

Table of Contents

Introduction 1
The Demonstration Program 1
Summary of Current Methods 2
  Signals 2
    Figure 1. Flow of Control: Signals 2
  Events 3
    Figure 2. Flow of Control: Events 3
  Events - The Balboa Version 3
The New Method: Replacing F$EVENT 4
  Basic Principles 4
    Figure 3. New Scan Synchronization Techniques 4
  Interfacing to My Implementation 5
  Writing Code to be Inserted into F$Event 6
Debugging 6
  Triggering Interrupts Yourself 6
  A Trace Buffer 7
Inside the New Method 8
  Interfacing to a System State Trap Handler 8
  Writing a System State Trap Handler 1
  Reading the Original F$Event Vector 2
  Calling F$SSvc 3
  Interfacing between the Trap Handler and Applications 3
  Accessing the System Path Descriptor 4
Appendix 1. Traps and Trap Handlers 5
Appendix 2. Source Code for Demonstration Program "syncdemo.c" 7
Appendix 3. Source Code for System State Trap Handler 14

A New Method For
Video Scan Synchronization in CD-i

INTRODUCTION

Accurate and reliable synchronization to the video scan has always been a problem in CD-i. Action games are particularly sensitive to this, and the current capabilities of CD-i are not nearly as effective as the capabilities of other platforms with which the developers of action games are accustomed. This note summarizes the two methods defined in the Green Book and a new method I recently discovered that seems to offer significantly greater reliability and predictability.

Also available is a demonstration program that uses the three methods to achieve the same results and visually clearly shows the timing and reliability problems of the two current methods. Full source code for this demonstration program is included in the appendices to this document.

The description of the new method is split into two sections—the first aimed at people who simply want to use the method and the second aimed at people who want to understand how it works. The new method can be used by people with only a little knowledge of 68000 assembler. Understanding how the method works requires a significant understanding of 68000 assembler, because much of my implementation is in assembler.

This document includes changes to address issues of reliability and debugging raised in feedback from early users of the new technique. (An earlier version of this document was published by PRL and distributed to a limited audience.)

THE DEMONSTRATION PROGRAM

The demonstration program, "syncdemo" shows a display on the CD-i player divided into two sections. The top 40 lines of the screen are green; the remainder of the screen should alternate between red and blue at the display rate of the player concerned. This switching is initiated by an interrupt instruction in the LCT at line 40 of the display. Ideally, there should be one transition between the green area and the blue/red area and no transitions inside the blue/red area.

When using signals to do the scan synchronization, there is a transition in the blue/red area a number of lines down the screen from the green area. Above this point, the display alternates between red and blue with each field; below this point, the display alternates between blue and red. This transition is evidence of a delay between the interrupt occurring in the LCT and activation of the demonstration program's signal intercept routine. The location of the transition is also subject to variation and can be located considerably further down the screer than is the normal minimum.

When using events to do the scan synchronization, the transition present in the red/blue area is almost absent, thus indicating a much faster response time than the signal method. It does, however, appear sometimes and can be almost as far down the screen as the minimum of the signal based case. Clearly, the event mechanism is subject to the same sort of variation as the signal mechanism.

When using the new method to do the scan synchronization, I have never noticed anything other than a completely clean transition. This indicates that the new method is both very responsive and very reliable.

In order to run the demonstration program, it is necessary to set the owner fields in the headers of both modules to 0.0; otherwise, the demonstration of the new method does not start due to a permission problem. This is done using the Microware "fixmod" utility.

SUMMARY OF CURRENT METHODS

CD-RTOS provides two primitives for video scan synchronization, one built on the CD-RTOS signal mechanism and one built on the CD-RTOS event mechanism.

Signals

Using the CD-RTOS system call dc_ssig()/DC_SSig, programs can specify a signal number to be sent when an interrupt is detected from the video decoder. This facility needs to be re-armed each time it is used. This facility is described in detail on page VII-174 of the Green Book and is shown by items 1, 4 and 5 in the diagram below.

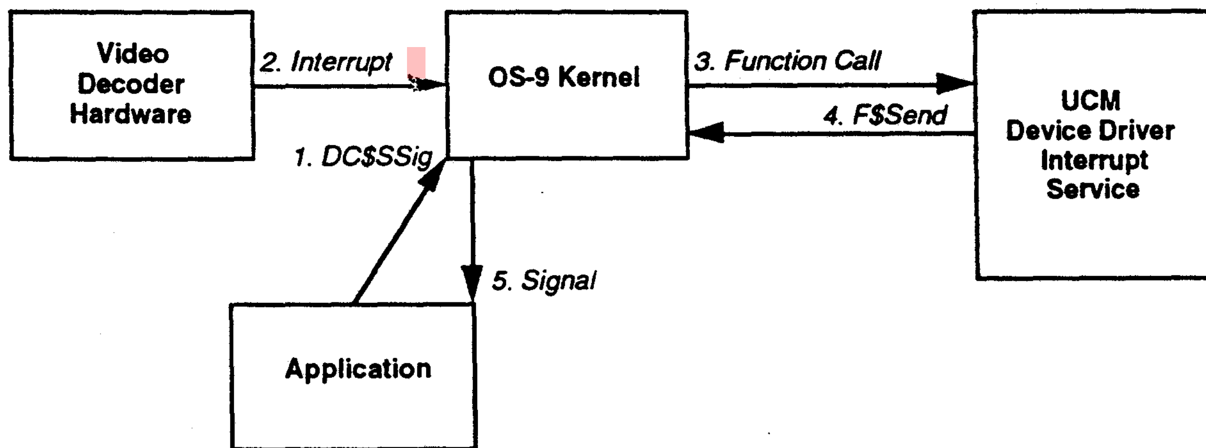

Figure 1. Flow of Control: Signals

Events

The CD-RTOS operating system includes a semaphore mechanism called "events." The CD-RTOS video system creates one of these called "line_event," and each time an interrupt is detected from the video decoder, this event is pulsed temporarily from value 0 to value 1. Processes may attach to "line_event" and wait for the event to be pulsed, at which time they are awakened. The general event mechanism is described in section 4 of the OS-9 Technical Manual. The "line_event" example is described at the bottom of page VII-79 in the Green Book. This is shown by items 1, 4 and 5 in the diagram below.

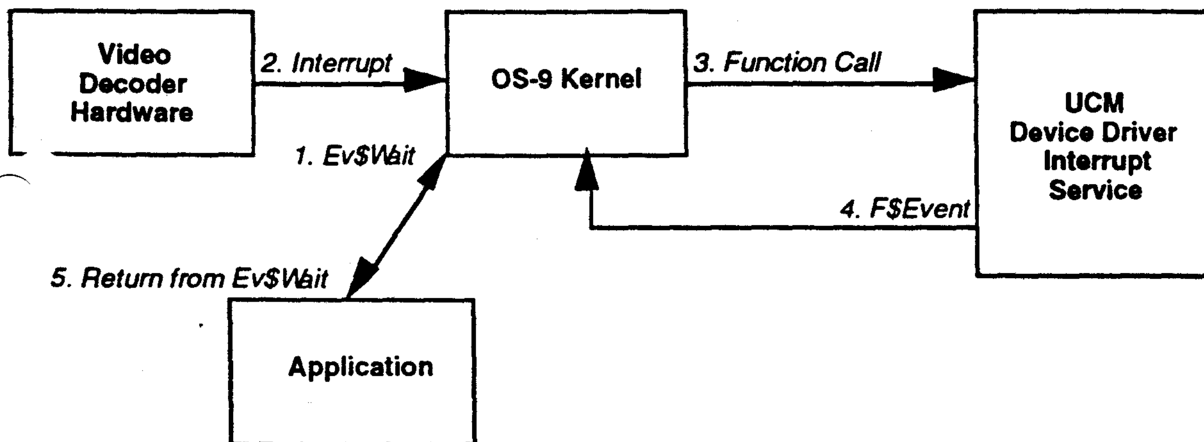

Figure 2. Flow of Control: Events

Events - The Balboa Version

The video synchronization manager in Balboa uses a variant of the event mechanism which should have even lower overhead. It forks a second process ("cdi_vsync"), which is in system state and which waits on the "line_event" in system state. This is more efficient than the basic event mechanism due to the lack of context switching between user state and system state.

This second process waits on the line event and then directly calls the function in the main process with the correct variable space pointer. In complex applications, this has another advantage in that the second process will not be awakened by the signals received by the first.

THE NEW METHOD - REPLACING F$EVENT

Basic Principles

The diagrams on the previous pages show the two methods that can be used by the UCM device driver interrupt handler to inform applications that a video interrupt has occurred. The new method uses a little-known CD-RTOS system call to intercept the **F$Event** call and replace it with some other code. The effect of this is to enable applications to insert their own code to be called during the execution of the video interrupt handler rather than at some indeterminate time later.

The F$SSvc system call is allowed only from within system state. There are many methods for getting into system state in OS-9; my method uses the simplest of these—a system state trap handler. For a discussion of traps and trap handlers, please see Appendix 1.

The new flow of events is shown in the diagram below. The application calls the system state trap handler by a trap (step 1). Once in the system state trap handler, F$SSvc is called to replace **F$Event** (step 2). When an interrupt happens in the video decoder (step 3), the OS-9 interrupt routine is activated and that calls the UCM device driver video interrupt routine (step 4). When that makes the **F$Event** call (step 5) that call now points to code in the system state trap handler. The system state trap handler performs some interfacing and then calls a function within the application (step 6).

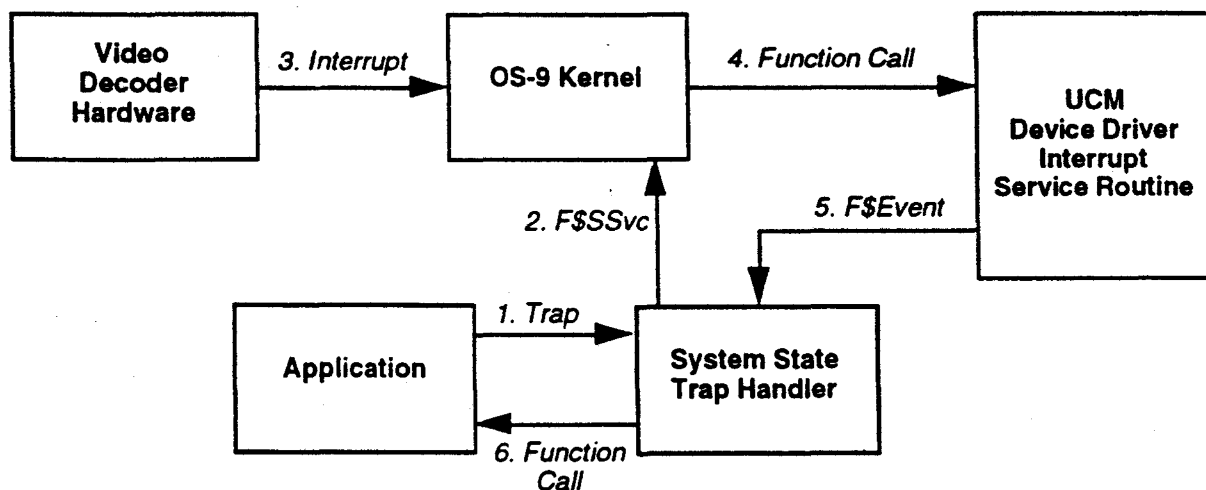

Figure 3. New Scan Synchronization Techniques

In order to avoid breaking other users of the event mechanism, the contents of the default **F$Event** vector are read before replacing it and that address called if the event call does not match that which would be generated by the UCM video interrupt service routine.

Interfacing to My Implementation

The interface to my implementation is three assembler functions that are designed to be called from "C." All three functions return the value **0** if successful or **-1**, if an error occurs; the error number is returned in the global variable "errno."

```
int install_trap( trap_handler_module_name )
char *trap_handler_module_name;
```

This function installs the system state trap handler module whose name is provided as the parameter. The module is normally present in the OS-9 module directory.

```
long init_interceptor( event_id, function )
long event_id;
int (*function)();
```

This function causes the operating system code for **F$Event** to be replaced by code in the system state trap handler installed by **install_trap()**. When the event whose identifier is contained in the **event_id** parameter to this function is pulsed, the application function specified by the function parameter to this function is called.

```
long remove_interceptor();
```

This function restores the **F$Event** code to be the original code from the operating system. This function is not in accordance with the Green Book and should be used only while debugging. Here is some example "C" code that shows how these functions can be used :

```
void asm_lct_switch();
int event_id;

        /* get the event identifier of "line_event" so that the new F$Event code
        knows which event it is to trap */

        event_id=_ev_link("line_event");

        /* install the trap handler called "sys_trap" */

        install_trap( "sys_trap" );

        /* replace F$Event and start our code being called */

        init_interceptor(event_id,asm_lct_switch);
```

The code called by the system state trap handler is executed with interrupts disabled in a very time critical interrupt handler. It should be kept as short as possible and should not do any form of input or output. The use of DC_PWrLCT to write an entire column of LCT instructions in my example is almost certainly too time consuming. The only input available to this code is the global variable space pointer from the main process which is passed in register a3. This enables code written in assembler to reference the global variables of the parent process.

My example does not support writing "C" code to be called in place of **F$Event**. The reason for this is that you obtain strange results if you call CD-RTOS functions from "C'

in system state. In order to avoid such unpredictable and difficult-to-explain errors, only assembler is supported.

Writing Code to be Inserted into F$Event

The biggest problem with writing code to be inserted into **F$Event** is the problem with calling CD-RTOS functions mentioned earlier. The nature of this problem is that, when in system state, the address of the CD-RTOS path descriptor must be in register **a1**, and the system path number (not the user path number) must be in register **d0**. My example program includes a function **find_path_descriptor()** that finds this via a somewhat convoluted route. Here is some example code to be called from inside the **F$Event** replacement, which also shows the usage of **find_path_descriptor()**.

```
int instructions_1[280],instructions_2[280];
int lct_flag = 0;
union pathdesc *find_path_descriptor();
union pathdesc *ucm_path_descriptor;

       ucm_path_descriptor = find_path_descriptor();

#asm
*
* this code is called with the user variables pointer in a3 not in a6 as it would
* normally be. Hence user globals are addressed as global_name(a3).
*

asm_lct_switch:
              lea.l    instructions_1(a3),a0   first lct instruction buffer
              tst.l    lct_flag(a3)
              beq.s    callit
              lea.l    instructions_2(a3),a0   second lct instruction buffer
callit        eori.l   #1,lct_flag(a3)
              move.l   ucm_path_descriptor(a3),a1
              move.w   0(a1),d0         system path id
              move.w   #SS_DC,d1
              move.w   #DC_PWrLCT,d2
              move.l   lcta(a3),d3      lct id
              moveq.l  #0,d4            starting line number
              moveq.l  #0,d5            starting column number
              move.l   #280,d6          number of lines
              move.l   #1,d7            number of columns
              tst.w    PD_CPR(a1)       test are we busy
              bne.s    skipit
              os9      I$SetStt         make the system call
skipit
              rts
#endasm

       The "C" equivalent to the above assembler would be :

       dc_pwrlct(ucm_path,lcta,0,0,280,1,
              (lct_flag)?instructions_1:instructions_2);
       lct_flag ^= 1;
```

In OS-9, the I/O system file managers and device drivers are not re-entrant. If code in an interrupt handler makes a UCM call while the main part of the application is also in a UCM call, the interrupt handler blocks, interrupts are enabled and the player may crash with a system stack overflow. In my example above, the PD_CPR field of the path descriptor is checked to see if the UCM is busy and, if that is the case, no call is made.

DEBUGGING

Debugging code called as part of interrupt service routines is very difficult, much more so than writing such code. In this case, it is not possible to use the Microware "sysdbg" utility in the normal manner because the video interrupt is a higher priority than the serial port interrupt, which "sysdbg" uses to handle its input and output. Here are two techniques that can help.

Triggering Interrupts Yourself

To use the "sysdbg" utility, it is necessary that the code concerned be called by a normal system call from a user state application and not from an interrupt handler. This can be done by removing all LCT and FCT video interrupt instructions and then making the same system call as the video device interrupt service routine, but from within the application. This uses the **_ev_pulse()** call. For example:

```
int event_id;

        /* write our interrupt signal - commented out for testing !!! */

        /*      FERROR(dc_wrli(vm_vidpath,lcta,START_LINE,2,cp_sig()),
                "error writing lct interrupt instruction");*/

        /* make the call which the interrupt handler would have made */

        _ev_pulse(event_id,1);
```

A Trace Buffer

One way of debugging code within interrupt service routines is a trace buffer. Since problems in such code often causes the CD-i player to crash, memory in the normal player memory map cannot be used because it will be cleared as part of the boot process. On a CD-i 605 player, the quantity of system RAM in the system visible to OS-9 can be reduced from 4M using the player shell. If the RAM is reduced from 4M to 3.5M, the memory from address $980000 to $a00000 is available as a trace buffer. Here is some example code showing how I have used this technique :

```
        /* initialize a base pointer in 'C' */

        *((int*)0x980000) = 0x980010;

asm_lct_switch:
                move.l  $980000,a2
                move.l  #1,(a2)+
                lea.l   instructions_1(a3),a0    first lct instruction buffer
...................
                move.l  #2,(a2)+
                move.l  (a2),$980000
                rts
```

The memory above the address $980000 can be examined using a debugger to see what is happening. Using this technique with the memory set on 4M crashes the system.

INSIDE THE NEW METHOD

This section of the document describes my example implementation in considerable detail. It is targeted at experienced assembler programmers with OS-9 knowledge. Understanding of this section is not needed for use of the example implementation described in the previous section.

Interfacing to a System State Trap Handler

Trap handlers are installed using the OS-9 call **F$TLink**. There is no "C" binding for this call and, hence, it must be done in assembler. An example of this call looks like :

```
TrapNum            equ     7                  ( arbitrary )
ItCancel           equ     0
ItIntercept        equ     1
*************************************************************
* install_trap - routine to install trap handler
* parameters - (d0) - address of name of module to use
* returns 0 for okay or -1 with error code in errno
* call from C as install_trap( char *name );
*************************************************************
install_trap:
       movem.l d1/a0-a2,-(a7)
       move.l  d0,a0                          the name of the module
       moveq.l #TrapNum,d0                    trap number to use
       moveq.l #0,d1                          no optional memory override
       os9     F$TLink                        install it
       bcs.s   error                          branch to error handler
*
* normal return code - no error has occurred
*
noerror
       moveq.l #0,d0                          return zero
*
* common return code between error and non error conditions
*
return_value
       movem.l  (a7)+,d1/a0-a2
       rts
*
* an error occurred so save the error number and setup for error condition
*
error
       move.l  #-1,d0                         return code
       ext.l   d1
       move.l  d1,errno(a6)                   save the error number
       bra.s   return_value
```

Calling code in a trap handler also needs to be done in assembler. Here is an example :

```
init_interceptor:
       movem.l d1/a0-a2,-(a7)
       tcall   TrapNum,ItIntercept            try a memory return
       bcc.s   noerror                        no error so just return
       bcs.s   error                          use the error reporting code from above
```

Writing a System State Trap Handler

A module is turned into a system state trap handler by setting the type entry in the main psect statement to TrapLib and setting the SupStat bit in the module revision word. The last item in the psect statement should be the label of the entry point for the main trap handler. The first statements following the psect should be offsets of the initialization and termination entry points. My version of this looks like the following:

```
Type    set     (TrapLib<<8)+Objct
Revs    set     (ReEnt+SupStat)<<8

        psect event_trap_interceptor,Type,Revs,0,0,TrapEnt
        dc.l   TrapInit         initialization entry point
        dc.l   TrapTerm         termination entry point
```

The termination routine is simple and may never be called. The initialization routine is called each time an OS-9 process attaches to the trap handler. One useful input to this routine is the first entry on the stack, which contains the contents of the calling processes a6 register; that is, the address of its variable space. The next long word on the stack is NULL and should be removed before the initialization routine exits. Here is an example initialization routine that does nothing :

```
TrapInit:
        move.l (a7)+,a6                         restore the user's a6 register
        addq.l #4,a7                            discard the NULL on the stack
        rts
```

As mentioned previously, the operation identifier supplied with the tcall pseudo instruction is passed to the trap handler entry point on the stack. Trap handlers typically include some tests or a jump table to translate this number into one of many pieces of code to call in the trap handler. In the case of the trap handler I used for the scan synchronization, this test code is very simple :

```
ItCancel        equu    0               remove our code from F$Event
ItIntercept     equ     1               replace F$Event with our code
maxtrap         equ     2               we only have two calls implemented here

TrapEnt:
        movem.l d0-d1,-(a7)             save registers
stacked set 2*4                         number of bytes we have put on the stack
*
* now check if the trap code is in range
*
        move.w stacked+4(a7),d0         get the operation id (in this case 0 or 1)
        cmpi.w #maxtrap,d0              are we in range ?
        bge.s   badtrap                 no so branch to an error handler
*
* work out the address of the routine to call using a series of tests
*
        lea.l   cancel_intercept(pc),a0         remove our code from F$Event
        cmp.w  #ItCancel,d0
        beq.s  TrapDespatch
        lea.l  intercept(pc),a0                 replace F$Event with our code
```

```
*
* restore registers and jump to the code we decided upon
*
TrapDespatch
        movem.l (a7)+,d0-d1                     restore the registers with the user
        jmp     (a0)                            call the routine
*
* bad trap number error - drops through into main error return routine
*
badtrap:
        movem.l (a7)+,d0-d1/a0
        move.l #E$IllFnc,d1
*
* Return With Error
*
ReturnError:
        ori     #1,ccr
*
*  common return routine
*
ReturnOkay:
        addq.l  #8,a7                           discard junk on stack
        rts
```

Reading the Original F$Event Vector

In order that the main OS-9 event mechanism can continue to work, my example reads the original contents of the **F$Event** vector and calls those contents if the event call does not match a video synchronization one. To read the original value of the **F$Event** vector, we need to get the address of the OS-9 vector table. This can be done using **F$SetSys**. In fact there are two such tables, one for calls made in user state and one for calls made in system state. The addresses of these are at locations **D_SysDis** and **D_UsrDis** in the system global variables table as accessed by **F$SetSys**. Here is some example code to read these two vector tables :

```
*
* user F$SetSys to obtain the base addresses of the system vector tables
*
        move.w  #D_SysDis,d0            system table
        move.l  #$80000004,d1           examine, long word
        os9     F$SetSys
        bcs.s   initdone
        move.l  d2,sys_table(a6)
*
* user F$SetSys to obtain the base addresses of the user vector tables
*
        move.w  #D_UsrDis,d0            user state table
        move.l  #$80000004,d1           examine, long word
        os9     F$SetSys
        bcs.s   initdone
        move.l  d2,user_table(a6)
initdone
```

Calling F$SSvc

The calling interface to **F$SSvc** is very simple and is described in the OS-9 manual. The main input to the call is in the form of a table. The table format allows the user state code and the system state code to be replaced separately. If you are doing this, you must be very careful to replace the system state code second or strange crashes can result.

The value that is contained in register **a3** when this call is made is passed to the code being inserted. In this case, we copy register **a6** to register **a3** ; this makes the variable space of the system state trap handler accessible to our replacement for **F$Event**. Here is some example source code :

```
*
* replace it
*
        lea.l    request_tables(pc),a1
        move.l   a6,a3                       variables for the inserted code
        OS-9      F$SSvc
*
* check for errors
*
        bcs.s    ReturnError
        bcc.s    ReturnOkay

request_tables:
        dc.w     F$Event
        dc.w     uevent-*-2
        dc.w     F$Event+SysTrap             this MUST follow the one without the SysTrap!
        dc.w     sevent-*-2
        dc.w     -1
```

Interfacing between the Trap Handler and Applications

Replacing the code for **F$Event** is only part of the task. There are other functions in the OS-9 system that use events, and they need to continue to work. In my example, the application passes the address of the routine to be called and the event id of the "line_event" to the system state trap handler. The code inserted into the interrupt service routine checks this event id against the event id passed to it and checks that the event code is **Ev$Pulse**. Having done this, it loads the parent's variable space address into register **a3** and calls the parent code specified.

```
*************************************************************************
*
* the event interceptor
*
*************************************************************************
uevent:
        move.l  user_event(a3),-(a7)    the original os vector
        bra.s   do_event
sevent:
        move.l  sys_event(a3),-(a7)     the original os vector
do_event
```

```
eck, is this an Ev$Pulse of the correct event ?

    cmp.l   event_id(a3),d0
    bne.s   not_ours
    cmp.w   #$8009,d1       Ev$Pulse, activate all processes
    beq.s   ours
    cmp.w   #9,d1           Ev$Pulse
    bne.s   not_ours

r function so call the user function

:
    movem.l d2-d7/a0-a6,-(a7)       save registers
    move.l  user_func(a3),a0
    move.l  user_vars(a3),a3
    jsr     (a0)                    call it
    movem.l (a7)+,d2-d7/a0-a6       restore registers

d we are finished. discard the original os vector from the stack.
d then drop through to ...

    adda.l  #4,a7

 event call is not our call. We can just return because the
iginal os vector is still on the stack from when we started

ours:
    rts
```

ssing the System Path Descriptor

iously, I have mentioned the problems with needing the address of the system path riptor in order to call some CD-RTOS functions from within system state. Below is a unction that finds the path descriptor address for a path to the CD-RTOS video ce.

uses the **attach()** call to attach to the UCM device. The **attach()** call returns a cetbl, which is a pointer to a structure described in **<sysio.h>**. From the Devicetbl, can get at the static storage for the device driver, a sysioStatic structure also in **io.h>**. From that structure, there is a list of path descriptors that we search for the e system path number is the same as the system path number of our UCM path. The user path number to system path number translation is performed by g the user path number as an index into the **_path** array in the process descriptor h can be read by **_get_process_desc()**.

```
union pathdesc *ucm_path_descriptor;
int vm_vidpath;                                  /* path to CD-RTOS video device */

void find_path_descriptor()
{
        Devicetbl attach();
        Devicetbl dt;
        union pathdesc *pd;
        procid pbuf;

        /* get a copy of our process descriptor */

        _get_process_desc(getpid(),sizeof(pbuf),&pbuf);

        /* attach to the ucm video device to get the device table entry */

        dt=attach(video_name,0);

        /* find the head of the path descriptor linked list */

        pd = ((sysioStatic*)(dt->V_stat))->v_paths;
        while( pd )
        {
                /* is this the path descr:ptor we are interested in ? */

                if( pd->path.pd_pd == pbuf._path[vm_vidpath] ) break;

                /* no, so on to the next one in the linked list */

                pd = pd->path.pd_paths;
        }
        /* check we found something */

        if( !pd )
        {
                fprintf(stderr,"no path descriptor found for ucm path\n");
                exit(-1);
        }
        return pd;
}
```

APPENDIX 1. TRAPS AND TRAP HANDLERS

One of the instructions in 68K family processors is called "trap." The four least significant bits of this instruction encode a vector number from 0 to 15. When one of these instructions is executed, an exception occurs and control is transfered to the address in the vector corresponding to the encoded trap number.

In OS-9, system calls are performed as calls to trap vector 0. Two other vectors are reserved for the system, leaving thirteen available to programs for their own use. In OS-9, these vectors are used in OS-9 by modules called trap handlers. The act of installing one of these modules connects that module to one of the thirteen remaining trap vectors. There is a partial explanation of this in section 5 of the OS-9 technical manual, "User Trap Handlers."

Trap handler modules have their own variable space just like normal executable programs; the difference is in how the code in these modules is called. The OS-9 assembler includes a pseudo instruction "tcall," that generates a trap instruction to a particular vector, followed by an operation identifier. When a trap instruction is generated, OS-9 calls the trap handler module (if any) appropriate for the process in which the trap instruction occurred and the trap number. The operation identifier is passed to the trap handler module on the processor stack, together with various other items.

In addition to the user state trap handlers described in the OS-9 technical manual, it is possible to create system state trap handlers. These are briefly mentioned in the OS-9 technical manual, but they are not described in any detail. The differences between the user state and system state trap handlers are determined mostly by running a system state trap handler under the "sysdbg" utility and looking at the register and stack contents.

APPENDIX 2. SOURCE CODE FOR DEMONSTRATION PROGRAM "SYNCDEMO.C"

```
/*
	demonstration program for video sync timings

	Jon Piesing PRL Redhill
*/

#include <types.h>
#include <Machine/reg.h>
#include <sysio.h>
#include <path.h>
#include <procid.h>
#include <stdio.h>
#include <csd.h>
#include <signal.h>
#include <ucm.h>
#include <errno.h>
#include <memory.h>
#include <modes.h>

/* error reporting macros */

#ifdef DEBUG
#define ERROR(A,B) {fprintf(stderr,(A));errno=(B);return -1;}
#define FERROR(A,B) if( (int)(A)==-1) {fprintf(stderr,(B));exit(errno);};
#define FERROR0(A,B) if( (int)(A)==NULL) {fprintf(stderr,(B));exit(errno);};
#else
#define ERROR(A,B) {errno=(B);return -1;}
#define FERROR(A,B) if( (int)(A)==-1) {exit(errno);};
#define FERROR0(A,B) if( (int)(A)==NULL) {exit(errno);};
#endif

/* signal numbers */

#define VIDEO_SIGNAL 256

/* define the line where the action happens */

#define START_LINE 40

/* global variables */

int fcta,fctb,lcta,vm_vidpath;
int instructions_1[280],instructions_2[280];
int event_id,intercepting=0;
char *video_name;
union pathdesc *ucm_path_descriptor;
int lct_flag = 0;

/* function prototypes */

char *srqcmem();
void setup(),signal_handler(),lctswitch(),remove_interceptor();
union pathdesc *find_path_descriptor();
void asm_lct_switch(),print_usage();

/* main */

main(argc,argv)
int argc;
char **argv;
{
```

```
int x;

/* setup */

setup();

/* do we have a parameter to tell us which mode to be in ? */

if( argc == 0 )     print_usage();

/* decide whether to use signals or events */

if( strcmp(argv[1],"-sig") == 0 )
{
        /* say what we are doing */

        printf("Demonstration of scan sync using signals\n");

        /* start it */

        dc_ssig(vm_vidpath,VIDEO_SIGNAL,0);

        /* sleep forever */

        while(0==0) sleep(100);
}
else if ( strcmp(argv[1],"-int") == 0 )
{
        /* say what we are doing */

        printf("Demonstration of scan sync by replacing F$Event\n");

        /* sort out the ugly os9 bits */

        ucm_path_descriptor = find_path_descriptor();

        /* get the event number */

        FERROR((event_id=_ev_link("line_event")),
               "error linking to line_event");

        /* install the trap handler */

        FERROR(install_trap( "sys_trap" ),
               "error installing system state trap handler");

        /* run our installer */

        FERROR(init_interceptor(event_id,asm_lct_switch),
               "error installing system call interceptor");

        /* remember we are intercepting */

        intercepting = 1;

        /* sleep forever */

        while(0==0) sleep(100);
}
else if( strcmp(argv[1],"-event") == 0 )
{
        /* say what we are doing */

        printf("Demonstration of scan sync using events\n");
```

```
        /* events */

        FERROR((event_id=_ev_link("line_event")),
              "error linking to line_event");

        /* loop forever */

        while(0==0)
        {
              FERROR(_ev_wait(event_id,1,2),
                    "error waiting for line_event");
              lctswitch();
        }
    }
    else
    {
        fprintf(stderr,"Unknown option %s\n",argv[1]);
        print_usage();
    }
}
  id print_usage()

    fprintf(stderr,"Synatx: syncdemo <sync_type>\n");
    fprintf(stderr,"Function: demonstration of different video sync options\n");
    fprintf(stderr,"Options:\n");
    fprintf(stderr,"    -sig        use signals for video sync\n");
    fprintf(stderr,"    -event      use events for video sync\n");
    fprintf(stderr,"    -int        insert our code into F$Event using F$SSvc\n");
    exit(256);
}
void setup()
{
    char *video1,*video2;
    int ibuf[7],x;

    FERROR0(video_name=csd_devname(DT_VIDEO,1),"error obtaining video device name");

    /* open the path */

    FERROR((vm_vidpath=open(video_name,S_IREAD)),"error opening ucm video device");

    /* create two fcts */

    FERROR0((fcta=dc_crfct(vm_vidpath,PA,8,0)),"error creating plane A fct");
    FERROR0((fctb=dc_crfct(vm_vidpath,PB,8,0)),"error creating plane B fct");

    /* create two lcts */

    FERROR0((lcta=dc_crlct(vm_vidpath,PA,560,0)),"error creating plane A lct");

    /* get memory for plane A and clear to zero */

    FERROR0((video1=srqcmem(384*280,VIDEO1)),"error getting video ram");
    memset(video1,0,384*280);

    /* get memory for plane A and clear to 1 */

    FERROR0((video2=srqcmem(384*280,VIDEO1)),"error getting video ram");
    memset(video2,1,384*280);

    /* set the top 40 lines of each to 2 so we have a visible scan line */

    memset(video1,2,384*START_LINE);
```

```
	memset(video2,2,384*START_LINE);

	/* initialise the two instructions arrays */

	for(x=0;x<280;x++)
	{
		instructions_1[x]=cp_dadr((int)video1);
		instructions_2[x]=cp_dadr((int)video2);
		video1 += 384;
		video2 += 384;
	}
	/* setup for mostly transparent in plane A, plane B off */

	ibuf[0]=cp_icm(ICM_CLUT7,ICM_OFF,NM_1,EV_ON,CS_A);
	ibuf[1]=cp_tci(MIX_OFF,TR_OFF,TR_ON);
	ibuf[2]=cp_dprm(RMS_NORMAL,PRF_X2,BP_NORMAL);
	ibuf[3]=cp_cbnk(0); /* clut bank */
	ibuf[4]=cp_clut(0,235,16,16); /* red */
	ibuf[5]=cp_clut(1,16,16,235); /* blue */
	ibuf[6]=cp_clut(2,16,235,16); /* green */
	PERROR(dc_wrfct(vm_vidpath,fcta,0,7,ibuf),"error writing to FCT");

	/* write our interrupt signal */

	PERROR(dc_wrli(vm_vidpath,lcta,START_LINE,2,cp_sig()),
		"error writing lct interrupt instruction");

	/* link lcts to fcts */

	PERROR(dc_flnk(vm_vidpath,fcta,lcta,0),"error in dc_flnk");

	/* activate both fcts */

	PERROR(dc_exec(vm_vidpath,fcta,fctb),"error in dc_exec");

	/* setup signal intercept handler */

	intercept(signal_handler);
}

void lctswitch()
{
	/* switch lcts */

	dc_pwrlct(vm_vidpath,lcta,0,0,280,1,
		(lct_flag)?instructions_1:instructions_2);
	lct_flag ^= 1;
}

void signal_handler(signum)
short signum;
{
	if( signum == VIDEO_SIGNAL )
	{
		/* re-arm the signal and switch display buffers */

		dc_ssig(vm_vidpath,VIDEO_SIGNAL,0);
		lctswitch();
	}
	else if( (signum==2)||(signum==3) )
	{
		if( intercepting ) remove_interceptor();
		exit();
	}
```

```
*
return_value
       movem.l  (a7)+,d1/a0-a2
       rts
*
* an error occurred so save the error number and setup for error condition
*
error
       move.l  #-1,d0                              return code
errnum
       ext.l   d1
       move.l  d1,errno(a6) save the error number
       bra.s   return_value

**************************************************************************
*
* init_interceptor - install the F$Event interceptor
*
* parameters d0 - event id
*            d1 - function to call
* returns    0 or -1 and error code in errno
*
* i.e. call from C as result=intercept( address,size );
*
**************************************************************************
init_interceptor:
       movem.l d1/a0-a2,-(a7)
       tcall  TrapNum,ItIntercept                  do it
       bcc.s  noerror                              no error so just return
       bcs.s  error
**************************************************************************
*
* remove_interceptor - restore F$Event to original value
*
* parameters - none
* returns    - 0 or -1 and error code in errno
*
**************************************************************************
remove_interceptor:
       movem.l d1/a0-a2,-(a7)
       tcall  TrapNum,ItCancel
       bcc.s  noerror                              no error so just return
       bcs.s  error
*****************************************************************************
*
* the assembler to call in the interrupt routine. This has to involve
* assembler because we are calling it in system state and we need
* the path descriptor pointer in a1 which cannot be done easily from a C
* call.
*
*     inputs - a3 - pointer to this program's variables
*     outputs      -
*
*****************************************************************************
asm_lct_switch:
              lea.l    instructions_1(a3),a0    first lct instruction buffer
              tst.l    lct_flag(a3)
              beq.s    callit
              lea.l    instructions_2(a3),a0    second lct instruction buffer
callit        eori.l         #1,lct_flag(a3)
              move.l   ucm_path_descriptor(a3),a1
              move.w         0(a1),d0           system path id
              move.w   #SS_DC,d1
              move.w   #DC_PWrLCT,d2
```

APPENDIX 3. SOURCE CODE FOR SYSTEM STATE TRAP HANDLER "SYS_TRAP.A"

```
      nam trap handler for intercepting F$event
      ttl memory allocating trap handler

      use <oskdefs.d>
*
* some symbols
*
ItCancel             equ    0
ItIntercept          equ    1
maxtrap              equ    2
*
* headers for the OS-9 module
*

Type   set    (TrapLib<<8)+Objct
Revs   set    (ReEnt+SupStat)<<8

      psect event_trap_interceptor,Type,Revs,0,0,TrapEnt
      dc.l   TrapInit                          initialization entry point
      dc.l   TrapTerm                          termination entry point

*****************************************************************************
*
* The initialisation routine - read system vectors and save user variable
*                               space pointer
*
* input :       (a7) caller's a6
*              4(a7) 0
*              8(a7) return address
*
*****************************************************************************
TrapInit:
*
* user F$SetSys to obtain the base addresses of the system vector tables
*
      move.w  #D_SysDis,d0                     system table
      move.l  #$80000004,d1                    examine, long word
      os9     F$SetSys
      bcs.s   initdone
      move.l  d2,sys_table(a6)
*
* user F$SetSys to obtain the base addresses of the user vector tables
*
      move.w  #D_UsrDis,d0                     user state table
      move.l  #$80000004,d1                    examine, long word
      os9     F$SetSys
      bcs.s   initdone
      move.l  d2,user_table(a6)
*
* save the user's a6 value
*
      move.l        (a7),user_vars(a6)
*
* done
*
initdone
      move.l  (a7)+,a6                         restore user a6 register
      addq.l  #4,a7                            discard the 0 long word
      rts                                      return

*****************************************************************************
```

```
*
* The exit routine - turn off interception
*
****************************************************************************
TrapTerm:
      bsr    cancel_intercept
      rts
****************************************************************************
*
* Now the main trap entry point
*
* input :        (a7) caller's a6
*             4(a7) trap code
*             6(a7) trap vector
*             8(a7) return address
*             d0/d1 parameters
* output: answers in d0/d1
*
****************************************************************************
TrapEnt:
*
* initial entry point - save registers
*
      movem.l d0-d1,-(a7)                      save registers
stacked set 2*4                                number of bytes we have put on the stack
      move.w stacked+4(a7),d0                  get the trap code ( in this case 0 or 1 )
*
* now check if the trap code is in range
*
      cmpi.w #maxtrap,d0
      bge.s  badtrap                           error
*
*     work out the address of the routine to call
*
      lea.l  cancel_intercept(pc),a0
      cmp.w  #ItCancel,d0
      beq.s  TrapDespatch
      lea.l  intercept(pc),a0
*
* very simple despatcher
*
TrapDespatch
      movem.l (a7)+,d0-d1                      restore the registers with the user
parameters
      jmp    (a0)                              call the routine
*
* bad trap number error
*
badtrap:
      movem.l (a7)+,d0-d1/a0
      move.l  #E$IllFnc,d1                     drop through into error return routine
*
* Return With Error
*
ReturnError:
      ori    #1,ccr
*
*  common return routine
*
ReturnOkay:
      addq.l #8,a7                             clear the junk from the stack
      rts

****************************************************************************
```

```
*
* install F$Event interceptor
* --------------------------
*
* parameters  - d0 - event id to look for
*             - d1 - user function to call
*
* returns     - if error then carry set and error number in d1.w
*             otherwise carry clear.
*
**************************************************************************
intercept:
       movem.l        d2-d7/a0-a6,-(a7)
*
* save our input variables
*
       move.l   d0,event_id(a6)
       move.l   d1,user_func(a6)
*
* read the current values of the system vectors
*
       move.l   user_table(a6),a2             user despatch table
       move.l   sys_table(a6),a1              system despatch table
       move.l   #F$Event,d0                   call number
       add.l    d0,d0                         multiply by 4
       add.l    d0,d0
       move.l   0(a1,d0.l),sys_event(a6)
       move.l   0(a2,d0.l),user_event(a6)
*
* replace it
*
       lea.l    request_tables(pc),a1
       move.l   a6,a3
       OS-9     F$SSvc
       movem.l        (a7)+,d2-d7/a0-a6       restore registers
*
* check for errors
*
       bcs.s    ReturnError
       bcc.s    ReturnOkay




**************************************************************************
**************************************************************************
*
* IT IS VITAL THAT THE ENTRY IN THIS TABLE WITH THE SysTrap ADDED COMES
* AFTER THE CORRESPONDING ENTRY WITHOUT THE SysTrap. IF THIS IS CHANGED
* THEN THE SYSTEM STARTS CRASHING FOR NO OBVIOUS REASON.
*
**************************************************************************
**************************************************************************

request_tables:
       dc.w   F$Event
       dc.w   uevent-*-2
       dc.w   F$Event+SysTrap
       dc.w   sevent-*-2
       dc.w   -1
**************************************************************************
*
* the event interceptor
```

```
*
****************************************************************************
uevent:
        move.l          user_event(a3),-(a7)        the original os vector
        bra.s   do_event
sevent:
        move.l  sys_event(a3),-(a7)                 the original os vector
do_event
*
* check, is this an Ev$Pulse of the correct event ?
*
        cmp.l   event_id(a3),d0
        bne.s   not_ours
        cmp.w   #$8009,d1                           Ev$Pulse, activate all processes
        beq.s   ours
        cmp.w   #9,d1                               Ev$Pulse
        bne.s   not_ours
*
* our function so call the user function
*
ours:
        movem.l         d2-d7/a0-a6,-(a7)           save registers
        move.l  user_func(a3),a0
        move.l  user_vars(a3),a3
        jsr     (a0)                                call it
        movem.l         (a7)+,d2-d7/a0-a6           restore registers
*
* and we are finished. discard the original os vector from the stack.
* drop through to ...
*
        adda.l  #4,a7
*
* the event call is not our call. We can just return because the
* original os vector is still on the stack from when we started
*
not_ours:
        rts

****************************************************************************
*
* remove F$Event interceptor
*
* THIS IS NOT CLEAN OR GREEN
* THIS CODE IS SPECIFIC TO DEVELOPMENT PLAYERS SO THAT THE DEMO CLEANS
* UP AFTER ITSELF.
* IT SHOULD NOT BE INCLUDED IN FINAL APPLICATIONS.
*
* parameters                none
* output                    none
*
****************************************************************************
cancel_intercept:
        movem.l         d0/a0-a2,-(a7)
        move.l  user_table(a6),a2                   user despatch table
        move.l  sys_table(a6),a1                    system despatch table
        move.l  #F$Event,d0                         call number
        add.l   d0,d0                               multiply by 4
        add.l   d0,d0
        move.l  sys_event(a6),0(a1,d0.l)            ! not clean/green
        move.l  user_event(a6),0(a2,d0.l)           ! not clean/green
        movem.l         (a7)+,d0/a0-a2
        bra     ReturnOkay

****************************************************************************
```