PHILIPS

PHILIPS INTERACTIVE MEDIA of America

Technical Note #47

Real-Time Code Loading

Charles Golvin

October 25, 1989

This technical note describes a method for loading an executable program from a real time record and executing that program. This technique maybe used by titles with memory considerations provided that the title engineer is able to divide the title into separately compiled programs. This technique does not provide the ability to load, execute, and return from a code fragment, such as a function or subroutine.

REAL-TIME CODE LOADING

The Problem

Suppose you have a CD-I application that is in part composed of several separate programs that execute at different times, started from a main module. You know that you can load a module from disc into memory using **modload()** or **modloadp()**, but you also know that opening and reading data from non-real time files is not as efficient as doing the same from real-time files. What you really want to do is deliver the module from a real-time file and then execute that program. How to do it?

The Answer

The key to solving this problem is the OS9 **F$VModul** service request. This call takes a pointer to some memory, verifies that the module pointed to is indeed a valid OS9 module. and, if so, enters the module into the system module directory. Once entered into this directory the module may be accessed as if it had been loaded by **modload()**, **modloadp()**, or by using the **load** utility from an OS9 shell.

The one dirficulty with **F$VModul** is that it is a privileged system state service request. This means that either the calling program must be running in system state (not very desirable if you want the system to be able to do anything else) or the program must be able to
load a system state trap handler that contains the call to **F$VModul**. I will describe the latter technique.

The Process

The first step is to create an OS9 system state trap handler that accesses the **F$VModul** service request. This functionality may be added to an existing system state trap handler or a new trap handler may be written to access this service request. Once the trap handler exists, there must be an assembly routine to make the actual trap call. The trap call may be surrounded by a C binding that prepares the necessary arguments for the called assembly routine. The only other code required is the assembly code that performs the installation of the trap handler module.

The only run time requirement for this scheme is that the buffer created to hold the module must be allocated using one of the system memory request functions **_srqmem()** or **srqcmem()** and not **malloc()**.

When the target task finishes executing and exits, the target module's link count is decremented and, if the link count becomes zero (-1 in the case of a sticky module) the module is unloaded from memory. If the target task was chained to, then the memory allocated for the target module buffer is returned to the system. If the target task was forked, then the forking process must use **_srtmem()** to return the allocated memory to the system.

Example

The following piece of pseudo code shows a possible sequence that might be used to load a module, execute it, and return to the original process:

```
#include <module.h>   /* needed for the modhcom structure */
#include <memory.h>                          /* needed for the SYSRAM define */
#include <modes.h>                           /* needed for the S_IREAD define */

                                            /* NUMBER is the size of the module to load,
                                             * rounded up to the nearest sector
                                             */
#define MODULE_SIZE NUMBER

char    *target_buffer,                  /* Buffer to catch incoming code module*/
        *rtf_name;                      /* Name of real time file containing code*/
int     rtf_id;                          /* Id of real time file containing code */
PCB     *play_pcb;                                  /* PCB for real time play */
extern char *srqcmem(),                            /* Colored memory allocator */
            **environ;                              /* Environmental variables */
extern int    os9forkc();                             /* Fork another process */
static char *arg_block[] =              /* Argument block for forking process */
{
        "target_name", NULL
};
        /* Install the trap handler */

install_trap_handler( int trap_number, char *trap_name );

        /* Allocate the target buffer */

if ( (target_buffer = srqcmem( MODULE_SIZE, SYSRAM )) == NULL )
        return( ALLOCATION_ERROR );

        /* Open the real time file */

if ( (rtf_id = open( rtf_name, S_IREAD )) == SYSERR )
        return( OPEN_ERROR );

        /* assume we've already set up the PCB and PCLs, and that the file pointer
         * is positioned to point to the start of the code module we wish to load
         * Now play the code module into the allocated buffer
         */
ss_play( rtf, play_pcb );

        /* assume now that the buffer full signal has been received indicating the
         * complete delivery of the code module we wish to load and that the play
         * of the real time record has been stopped if appropriate
         *
         * do_load() is just a C-binding to the trap handler that assembles the appropriate
         * arguments and then calls the trap
         */
if ( do_load( (struct modhcom *)target_buffer ) == SYSERR )
        return( LOAD_ERROR );

        /* Now fork to the code module that we loaded */

if ( os9exec( os9forkc, arg_block[0], arg_block, environ, 0, 0, 3 ) == SYSERR )
        return( FORK_ERROR );
```

Summary

The requirements for the use of this code loading strategy are:

- A system state trap handler containing a call to **FSVModul** and the means to install said trap handler.
- A C binding that prepares the arguments and provides access to the trap handler.
- An application that knows the size, in sectors, of each module it will start and the offset from the start of the real time file containing the module to the start of the module.
- A buffer, allocated from system memory, of sufficient size to contain the module it wishes to start.

  Note: PIMA has sample code designed to load an executable program from a real-time record and to execute it. To request this program, please contact Developer Services, (310) 444-6519.