PHILIPS

PHILIPS INTERACTIVE MEDIA of America

Technical Note #80

Using Subroutine Modules

Charles Golvin

August 27, 1992

This technical note describes the use of OS-9 subroutine modules in order to dynamically load executable code on demand.

Copyright © 1992 Philips Interactive Media of America.
All rights reserved.
This document is not to be duplicated or distributed without written permission from Philips Interactive Media of America.

TABLE OF CONTENTS

Using Subroutine Modules

Overview

One of the constants in a base-case CD-I player is the one megabyte of memory that is in the system. This memory is reduced by system overhead to a Green Book specified minimum of 480 kilobytes of available memory per plane at the time the application program is loaded into RAM by the player shell. Once loaded, the application program remains in memory until it exits or chains to another process, further diminishing the amount of available memory for other assets. In contrast, the amount of available information on a CD-I disc exceeds six hundred megabytes. Thus, it would be a clear advantage to be able to extend the features of an application program by dynamically loading a specialized section of the application program into RAM. This piece of code, when no longer needed, could then be released from memory, thus freeing up room for video, audio, and other presentation assets. This capability, under certain restrictions, is provided by the use of OS-9 subroutine modules. This note describes how to create and employ subroutine modules and the restrictions associated with doing so. In addition, a simple example is included in the appendices.

Subroutine Modules vs. Program Modules

A program module is a complete program, including its own local data and stack memory description, initialized data and references, command line argument processing routine (i.e., main), and initial entry point. The module contains all the information necessary for the operating system to load, initialize, and execute the program to completion. Calls between functions that are not in the same C source file are resolved by the linker, and these calls are made by function name at the C source level.

A subroutine module is essentially a collection of executable functions that have no context of their own. The module contains no local data or stack requirement, may not contain any global data, has no initialized data or references, receives no command line arguments, and has no initial entry point. It is not possible for the operating system to load and execute the code in such a module—another process must do all the work necessary to execute the function(s) contained within the module. The functions within a subroutine module may call one another by name at the C source level, but it is not possible for functions outside the subroutine module to call its functions in this manner (nor vice versa).

The structure of a subroutine module is quite simple. Following the module header is a list of pairs of values, one pair for each "declared" function, each of which is interpreted as a byte offset from the beginning of the module. The first value of each pair is the offset to the name string of this function, and the second value of the pair is the offset to the entry point for the function itself. The pair of null values that come at the end of this list serves as a terminator, indicating that no more functions are declared.

Note: These offsets may be either words or longwords, as determined by the definitions in the substart module.

Creating Subroutine Modules

It is only slightly more difficult to create subroutine modules than it is to create program modules, but once you have created one module, subsequent instances are easy. The first one is only different because you must create the code to act in place of **cstart**. So, first a brief digression about this critical piece of code.

There is a critical piece of assembly code that is compiled into every OS-9 C program. The file containing these instructions is generally called **cstart**, and this code is executed by the operating system prior to reaching the function called **main**. This code sets up many of the system variables required for a program to run, allows for the provision of environment variables to the program, and prepares the initial argument list that the program receives (i.e., **argv** and **argc**). In addition, the standard Microware-supplied **cstart** file contains stack checking functions and CIO initialization. For example, if you've ever run a program under the OS-9 source level debugger, you've seen that the first executable instruction is at the symbol **_cstart**. A single **cstart** file is typically sufficiently generic to work for any C program, because all the information is contained within the program module.

In contrast, each subroutine module, in order to allow it to be called from arbitrary external programs, generally possesses its own unique start-up file (which I will call "substart" from now on). In structure these various substart files are analogous to each other, only the information for the particular functions in the module changes between instances. Because of the structure of the subroutine module described above, it is necessary to "declare" each function that may be called directly from outside the subroutine module in the module's substart file. By simply providing some keywords embedded in comments in a C file, it is possible to automate the creation of a specialized substart file (see the appendices for examples of substart files and the corresponding C files).

To create a substart for a subroutine module, first write the subroutine module code. In doing so, you must decide which of the functions in the module will be callable from outside the module. The names of these functions must be made known to the substart module via a table which is described in a "psect" directive. This is easily demonstrated by example—the following is the substart code for a subroutine module that makes three functions known to the outside world: **summarize_proust**, **ex_parrot**, and **lumberjack**. In this example, the table is called "SubTable," and its construct clearly matches the structure of the offset table described in the previous section.

Note Because this module is much smaller than 64 Kbytes, word offsets rather than longword offsets are used for the offset table.

```
            use  defsfile

Edition     equ       1
Typ_Lang    equ       (Sbrtn<<8)+Objct
Attr_Rev    equ       (ReEnt<<8)+0

psect   subs_a,Typ_Lang,Attr_Rev,Edition,0,SubTable

* Resolve the references generated from Cstart - originate from -$8000
            org  -32768
_sttop      do.l  1    ; Stack top
_mtop:      do.l  1    ; current non-stack memory top
_stbot:     do.l  1    ; current stack bottom limit
errno:      do.l  1    ; global error holder
_totmem:    do.l  1    ; total data memory used
_sbsize:    do.l  1    ; top of process memory (sbrk)
_fcbs:      do.l  1    ; file control block ptr (for CIO)
environ:    do.l  1    ; environment pointer (for getenv())
_pathcnt:   do.w  1    ; number of open paths at process creation
_sysglob:   do.l  1    ; pointer to system globals (if system-state
process)

* Use word tables rather than longwords since the module is <64K

SubTable:   dc.w  Nam0,summarize_proust
  dc.w  Nam1,ex_parrot
  dc.w  Nam2,lumberjack
  dc.w  0,0

Nam0   dc.b  "summarize_proust",0
Nam1   dc.b  "ex_parrot",0
Nam2   dc.b  "lumberjack",0

            ends
```

This code may be compiled directly using the OS-9 assembler **r68**. It is important, however when compiling the accompanying C code to remember that subroutine modules may not contain initialized data. So, for example, SCCS identifier strings should be contained within **#ifdef** directives and compiled out. In addition, it is important to recognize that the above substart code does not include stack checking, so in compiling the C source for the subroutine module the **-s** compiler option of the Microware compiler must be used to turn stack checking off. Once the C source and the substart code has been compiled and assembled, it may be linked using **l68**. Be advised that the options to **l68** do not parallel those for **cc68** and that libraries typically included by **cc68** must be explicitly referenced when using **l68**. Consult the example makefile included in the appendices for a concrete example.

Loading Subroutine Modules

A subroutine module, like any other OS-9 module, may be loaded into RAM and entered into the module directory via the system calls **modload** or **modloadp** (i.e., **FSLoad**). If the module is already resident in RAM, it may be entered into the module directory by the privileged system call **F$VModul** (see PIMA Technical Note

#47, *Real Time Code Loading* , for more information on this technique).

However, in the case of a single CD-I application that needs to load code on demand to be used only by that program, there is no need for the operating system to be aware of the presence of the module in memory. Therefore, it is not necessary to enter the module into the module directory—the module may be loaded by the application into memory that it has allocated, and when the application no longer needs the services of the module, it may simply reuse the memory for another purpose.

The advantage gained by using the system calls to enter the module into the module directory is the cyclic redundancy check that the system performs on the module's data. However, it is possible for an application to perform the same check using the system call **F$CRC**. Use of this call has the added advantage of preventing an unexpected system memory allocation if the new module forces the operating system to extend the size of the module directory.

In consulting the examples in the appendices, note that both methods for loading subroutine modules into memory are provided, depending on whether the preprocessor symbol MODULES is defined.

Calling Functions in Subroutine Modules

Once a subroutine module is resident in RAM, the calling process must find the offset to the function's execution entry point. The address that results from adding this offset to the start of the module in RAM is the beginning of the executable code for that function. The calling process must simply point a function pointer at this address and call the function with the proper parameters.

In order to locate the function offset pointer, the calling process must have the name of the function, as described in the module's substart file, declared as a string internally. The table of name and entry point offsets for the functions declared in the module is also coded as an offset from the beginning of the module, and this offset always immediately follows the module header, that is, $30 bytes from the start of the module. Once the table is found, the calling process must simply index through the table, comparing its internal string with that pointed to by the first value in the offset table. Once a match is found, the corresponding code offset is added to the module address and the entry point is recorded—the process is now prepared to call the function. If, in searching for the right function name, a string offset of zero is found accompanied by a code offset of zero, the list of functions has been exhausted and a match was not found.

Because one of the ideas of this practice is to minimize memory overhead, it may seem wasteful to pay twice for the names of the functions (once in the calling process and once in the subroutine module). Because the subroutine module is designed to be loaded only when needed and then disposed of, it makes sense to group all the functions pertaining to an activity in one module. For these reasons, as well as the fact that functions within the module may call each other in the

"normal" way, the examples provided employ a strategy that is recommended by the author: create a single (apologies to Balboa) "dispatcher" function whose first parameter is a function code. This function code identifies which of the functions in the module is to be called, and the dispatcher function simply makes the direct call. The dispatcher function is then a simple switch statement, much like a signal handler routine. Thanks to Douglas Yoon for suggesting this scheme.

Development Concerns

Because the global variable **errno** is known to the substart module, intelligent error reporting is possible through subroutine modules. However, it is not possible to use the source level debugger on subroutine modules. Code fragments that are destined for subroutine modules should be developed and debugged initially as an integral part of the application. Once the bugs have been eliminated, the functions may then be stripped out into the subroutine module.

Summary and Conclusions

Subroutine modules provide an application with the ability to load and execute code on demand and then to reuse the memory devoted to this code for other purposes. Using this method, the overall memory overhead of a CD-I program may be reduced. In addition, the distribution of code between the two planes of video memory may be balanced more evenly, or in a way that best suits the memory requirements of the application. While the development of these modules requires some work beyond that normally required for monolithic programs, the effort is necessary only at the initial stages of developing the technique. The advantages and flexibility more than offset these initial costs.

Appendix A: Example Program

The example contained here consists of a main program (**cdi_example.c**), two subroutine modules (**init_dispatch.c** and **audio_dispatch.c**), substart code for each of the two subroutine modules (**substart.init_dispatch.c** and **substart.audio_dispatch.c**), and a makefile. In addition, a Unix C shell script which illustrates how to automate the creation of a subroutine module substart file from its C source file is included (**extract_subnames**).

The idea of the program is very simple: initialize the audio, video, and disc devices, then put up a picture, and play a soundmap twice. The subroutine module **init_dispatch** is used to initialize the various devices, and the subroutine module **audio_dispatch** is used to perform various audio functions.

> *Note: This code is not complete—there are external functions referenced in the file* ***init_dispatch.c*** *for accessing the CSD that are not given here. If you wish to actually compile this code, you may replace each of the calls to* ***my_csd_devname()*** *with a call to* ***csd_devname()****, removing the third parameter to the function.*

```
/*****************************************************************************
*       Filename:                                                  cdi_example.c
*       Project:                    Simple example of using subroutine modules
*       functions:                 main(), linkto(), unlinkfrom(), sig_catch()
*       author:                                                  Charles Golvin
*       date:    Thu Aug 13 17:22:02 PDT 1992
*       SCCS: 1.1                                                      8/13/92
*****************************************************************************/
#if !defined(NO_BVC)
static char SCCS_cdi_example_c[] = "@(#)cdi_example.c 1.1 8/13/92";
#endif

#include <stdio.h>
#include <errno.h>
#include <cdfm.h>
#include <ucm.h>
#include <module.h>
#include <modes.h>

                                                    /* Some general defines */
#define SYSERR (-1)
#define MODOVHD 68
#define funcptr int(*)()

                                                /* Function codes for audio handling */
#define AUDIO_BASE 0x10
#define SM_OUT  AUDIO_BASE
#define SM_OFF  AUDIO_BASE+1
#define SM_STAT AUDIO_BASE+2
#define SM_INFO AUDIO_BASE+3
#define SC_ATTEN AUDIO_BASE+4
#define SM_CREAT AUDIO_BASE+5

                                /* Function codes for initialization handling */
#define INIT_BASE                                                       0x20
#define INIT_AUDIO INIT_BASE
#define INIT_VIDEO INIT_BASE+1
#define INIT_DISC INIT_BASE+2
#define INIT_PTR INIT_BASE+3
                                            /* Some signals we'll receive */
#define SHSIG   1024
#define KEYSIG  1025
                                    /* The names of the dispatcher functions */

static char aud_disp[] = "audio_dispatch";
static char init_disp[] = "init_dispatch";

                            /* Parameter block to the initialization functions */
typedef struct _init_p
{
        int path;                                       /* Path to the device */
        unsigned char *tbuf;                        /* Buffer for CSD handling */
        int fa,fb,la,lb;                                    /* FCTs and LCTs */
} init_parms;

                                    /* Parameter block to the audio functions */
typedef struct _astuff
{
        unsigned first, second, third, fourth;
} audio_stuff;


void sig_catch();                                   /* Signal handler function */

                                  /* data structure for subroutine module */
```

```
typedef  struct  _submod
{
        mod_exec  *head;                          /* Pointer to head of module */
        char  *name;                             /* Name of dispatcher function */
        int  *fptr                                  /* Actual function pointer */
        size;                                                /* size of module */
}  submod;

/*****************************************************************************
*  Function:   main()
*  Purpose:    Initialize cd, audio, and video devices, show a picture and
*              play a couple of sounds
*  Passed:     pointer to array of char pointers to arguments and # of args
*  Returned:   exit() to system with error number
*****************************************************************************/
main(ac,av)
int  ac;
char  **av;
{
        submod  do_audio,                    /* Audio functions subroutine module */
           do_init;               /* Initialization functions subroutine module */
        int  (*dest)(),                                /* Generic function pointer */
           ap, vp, cdp,                                     /* Some path variables */
           smid, sz, *bfptr;                                        /* Other junk */
        init_parms  iparms;                    /* Initialization parameter block */
        audio_stuff  astuff;                             /* Audio parameter block */

                                                        /* And assorted other junk */
        mod_exec  *modptr, *modlink();
        unsigned  char  buf[2048], *smmem,  *pp;
        STAT_BLK  stblk;

        intercept(  sig_catch  );
        errno = 0;
        memset(  (char  *)&iparms,  0,  sizeof(  iparms  )  );;

                                                    /* Initialize function names */
        do_audio.name  =  aud_disp;
        do_init.name  =  init_disp;
                                    /* Link to the initialization subroutines */

        if  (  linkto(  &do_init  )  ==  SYSERR  )
        {
           fprintf(stderr,"Can't  get  init_stuff  subroutine  module\n");
           exit(errno);
        }
                                                     /* Do some initializations */

        dest  =  (funcptr)(do_init.fptr);
        iparms.tbuf  =  buf;
                                                           /* First the cd device */
        if  (  (*dest)(  INIT_DISC,  &iparms  )  ==  SYSERR  )
        {
           fprintf(stderr,  "Error  initializing  cd  device\n"  );
           exit(errno);
        }
        cdp  =  iparms.path;                    /* Record the path number received */
                                                          /* Now the audio device */

        if  (  (*dest)(  INIT_AUDIO,  &iparms  )  ==  SYSERR  )
        {
           fprintf(stderr,  "Error  initializing  audio  device\n"  );
           exit(errno);
        }
```

```
ap  =  iparms.path;                    /*  Record  the  path  number  received  */
                                                    /*  Now  the  video  device  */

if  (  (*dest)(  INIT_VIDEO,  &iparms  )  !=  0  )
{
   fprintf(stderr,  "Error  initializing  video\n"  );
   exit(errno);
}
vp  =  iparms.path;                    /*  Record  the  path  number  received  */

                        /*  Link  to  a  picture  I  have  already  in  plane  A  */

if  (  (int)(modptr  =  modlink(  "cdi_paulina",  MT_ANY  ))  ==  SYSERR  )
{
   fprintf(  stderr,  "Can't  link  to  cdi_paulina\n"  );
   exit(  errno  );
}
pp  =  (unsigned  char  *)modptr  +  64;
munlink(  modptr  );


                                                          /*  Prepare  the  DCP  */
bfptr  =  (int  *)buf;
bfptr[0]  =  cp_po(  PR_AB  );
bfptr[1]  =  cp_icm(ICM_DYUV,  ICM_OFF,  NM_1,  EV_OFF,  CS_A);
bfptr[2]  =  cp_tci(  MIX_OFF,  TR_OFF,  TR_OFF  );
bfptr[3]  =  cp_icf(  PA,  ICF_MAX  );
bfptr[4]  =  cp_matte(  0,  MO_END,  MF_MFO,  0,  0  );
bfptr[5]  =  cp_matte(  4,  MO_END,  MF_MFO,  0,  0  );
bfptr[6]  =  cp_yuv(  PA,  128,  128,  128  );
bfptr[7]  =  cp_dadr(  cp_dadr((int)pp)  );
bfptr[8]  =  cp_dprm(  RMS_NORMAL,  PRF_X2,  BP_NORMAL  );

if  (  dc_wrfct(  vp,  iparms.fa,  0,  9,  bfptr  )  ==  SYSERR  )
{
   fprintf(stderr,  "Error  %d  writing  FCT  instructions\n",  errno  );
   exit(errno);
}
                                                              /*  Now  show  it  */
if  (  dc_exec(  vp,  iparms.fa,  iparms.fb  )  ==  SYSERR  )
{
   fprintf(stderr,  "Error  %d  on  dc_exec\n",  errno  );
   exit(errno);
}
            /*  OK,  done  with  initializations-get  rid  of  that  module  */

if  (  unlinkfrom(  &do_init  )  ==  SYSERR  )
{
   fprintf(stderr,  "Error  unlinking  initialization  module\n"  );
   exit(errno);
}

                                  /*  Link  to  some  sound  I  have  in  memory  */

if  (  (int)(modptr  =  modlink(  "cdi_cmem2",  MT_ANY  ))  ==  SYSERR  )
{
   fprintf(  stderr,  "Can't  link  to  cdi_cmem2\n"  );
   exit(  errno  );
}
sz  =  (modptr->_mh._msize  -  MODOVHD)/2304;
pp  =  (unsigned  char  *)modptr  +  64;

                                                    /*  Now  do  some  audio  stuff  */
if  (  linkto(  &do_audio  )  ==  SYSERR  )
```

```
{
    fprintf(stderr,"Can't get audio_stuff subroutine module\n");
    exit(errno);
}

dest = (funcptr)(do_audio.fptr);

                                                    /* Create a soundmap */
astuff.first = ap;
astuff.second = D_CMONO;
astuff.third = sz*18;
astuff.fourth = (int)&smmem;
if ( (smid = (*dest)( SM_CREAT, &astuff )) == SYSERR )
{
    fprintf(stderr, "Error creating soundmap\n" );
    exit(errno);
}
else                                          /* Copy the soundmap data */
{
    memcpy( smmem, pp, sz*2304 );
    munlink( modptr );
}
                                                      /* Set attenuation */
astuff.first = ap;
if ( (*dest)( SC_ATTEN, &astuff ) == SYSERR )
{
    fprintf(stderr, "Error setting atten\n" );
    exit(errno);
}

                                                         /* Play the map */
astuff.first = ap;
astuff.second = smid;
stblk.asy_stat = 0;
stblk.asy_sig = SMSIG;
astuff.third = (int)&stblk;
if ( (*dest)( SM_OUT, &astuff ) == SYSERR )
{
    fprintf(stderr,"Error outputting smap\n" );
    exit(errno);
}
else
    sleep(0);                                  /* Wait for it to complete */

                                              /* Play the map then abort it */
astuff.first = ap;
astuff.second = smid;
stblk.asy_stat = 0;
stblk.asy_sig = SMSIG;
astuff.third = (int)&stblk;
if ( (*dest)( SM_OUT, &astuff ) == SYSERR )
{
    fprintf(stderr,"Error outputting smap\n" );
    exit(errno);
}
else
{
    tsleep( 50 );
    if ( (*dest)( SM_OFF, &astuff ) == SYSERR )
    {
        fprintf(stderr, "Error stopping smap\n" );
        exit(errno);
    }
}
```

```
                                /* OK, done with audio - get rid of that module */

        if ( unlinkfrom( &do_audio ) == SYSERR )
        {
           fprintf(stderr, "Error unlinking audio module\n" );
           exit(errno);
        }
                                    /* Wait for a keyboard interrupt, then leave */
        _ss_ssig( 0, KEYSIG );
        sleep(0);
        exit( errno );
}


/*************************************************************************
*  Function:   linkto()
*  Purpose:    Load the desired subroutine module and set up its data structure
*  Passed:  Pointer to subroutine module data structure
*  Returned:   0 or SYSERR if the module fails to load
*  Note:       This routine will use either a direct open,read method or the
*              operating system calls modload, modloadp depending on whether
*              the preprocessor symbol MODULES is defined
*************************************************************************/
int linkto( smod )
submod  *smod;
{
        char  *sptr, *srqcmem();
        int  *tptr;
        mod_exec  *modlink(), *modloadp();

#if  defined(MODULES)
                      /* This is how to load using the OS - you get the benefit of
                       * the CRC check this way (although you can check it  yourself)
                      */

        if  ((int)(smod->head  =  modlink(smod->name,0)) ==  SYSERR  )
           if  ((int)(smod->head  =  modloadp(smod->name,0,0)) ==  SYSERR  )
           {
              fprintf(stderr,"can't  find  %s",smod->name);
              return( errno );
           }
        smod->size  =  (smod->head)->_mh._msize;

#else
                                          /* This is how you get it directly */
        int  sp;

        if ( (sp = open( smod->name, S_IREAD )) == SYSERR )
        {
           fprintf(stderr,"Can't  open  %s\n",smod->name);
           exit(errno);  }

        if ( (smod->size = lseek( sp, 0, 2 )) == SYSERR )
        {
           fprintf(stderr,"Can't  get  size\n");
           close(sp);
           exit(errno);
        }
```

```
        if ( lseek( sp, 0, 0 ) == SYSERR )
        {
            fprintf(stderr,"Can't seek back\n");
            close(sp);
            exit(errno);
        }

        if ( (int)((char *)smod->head = srqcmem( smod->size, 0 )) == SYSERR )
        {
            fprintf(stderr,"Can't allocate memory\n");
            close(sp);
            exit(errno);
        }

        if ( read( sp, (char *)smod->head, smod->size ) != smod->size )
        {
            fprintf(stderr,"Can't read file\n");
            close(sp);
            exit(errno);
        }

        if ( close( sp ) == SYSERR )
        {
            fprintf(stderr,"Can't close file\n");
            exit(errno);
        }
#endif


            /* tptr first points to the offset to the first offset in the table */
        tptr = (int *)((char *)smod->head + smod->head->_mexec);



        sptr = (char *)smod->head + (*tptr >> 16);
        if ( strcmp( sptr, smod->name ) != 0 )
        {
            fprintf(stderr,"linkto: strings didn't match\n");
            return( SYSERR );
        }
            /* got the match - the low word of tptr hold the offset to the code */
        /* in the case of longwords (*tptr & 0x0000ffff) would become just *++tptr */
        smod->fptr = (int *)((char *)smod->head + (*tptr & 0x0000ffff));

        return( 0 );
}

/*------------------------------------------------------------------------
*  Function:   unlinkfrom()
*  Purpose:    Unlink a loaded subroutine module
*  Passed:     Pointer to subroutine module data structure
*  Returned:   0 or SYSERR if the module fails to unload
*  Note:       This routine will use either a direct _srtmem method or the
*              operating system calls munload depending on whether
*              the preprocessor symbol MODULES is defined
*------------------------------------------------------------------------*/
int unlinkfrom( mod )
submod  *mod;
{
#if  defined(MODULES)
```

```
        if  (  munlink(mod->head)  ==  SYSERR  )
        {
           fprintf(stderr,  "Error  unlinking  subroutine  module\n");
           return(SYSERR);
        }

#else

        if  (  _srtmem(  mod->size,  mod->head  )  ==  SYSERR  )
        {
           fprintf(stderr,  "Error  returning  memory\n");
           return(SYSERR);
        }

#endif

        return(  0  );
}

/*************************************************************************
*  Function:   sig_catch()
*  Purpose:    Intercept  any  signals  meant  for  this  process
*  Passed:     Signal  number
*  Returned:   No  return  -  void  function
*************************************************************************/
void  sig_catch(  sig  )
int  sig;
{
        switch(  sig  )
        {
           case  SMSIG:                              printf("Soundmap  completed\n");
                                                              break;

           case  2:
           case  3:
           default:                                 printf("outta  here  on  %d\n",sig);
                                                          exit(errno);
                                                              break;
        }
        return;
}
```

```
/*****************************************************************************
*       Filename:                                                  init_dispatch.c
*       Project: Simple example of using subroutine modules
*       purpose: Illustrate division of simple CD-I program in modules
*       internal functions: init_disc(), init_audio(), init_video(), init_ptr()
*       externally known functions: init_dispatch()
*       author: Charles Golvin
*       date:   Thu Aug 13 17:22:02 PDT 1992
*       SCCS: 1.1                                                       8/13/92
*
*       Note:   The following line identifies the functions that must be
*               identified in the corresponding substart file.  It MUST be here
*               for the automated substart generator to work correctly.
*
#   INCLUDED_SUBROUTINE_FUNCTIONS:                                 init_dispatch

*****************************************************************************/
#if  !defined(NO_BVC)
static char SCCS_init_dispatch_c[] = "@(#)init_dispatch.c    1.1        8/13/92";
#endif
#include   <modes.h>
#include   <csd.h>
#include   <ucm.h>

#define    CSD_NAME                                                  "/nvr/csd"
#define    FCT_SIZE                                                        150
#define    PAL_LCT                                                         560
#define    NTSC_LCT                                                        480
#define    PAL_625_STR    "625"              /* Specifies a 625 line system */

#ifndef    SYSERR
#define    SYSERR                                                       (-1)
#endif
#ifndef    NULL
#define    NULL                                                          0L
#endif

#define    INIT_BASE                                                     0x20
#define    INIT_AUDIO                                             INIT_BASE
#define    INIT_VIDEO                                           INIT_BASE+1
#define    INIT_DISC                                            INIT_BASE+2
#define    INIT_PTR                                             INIT_BASE+3

                          /* Parameter block to the initialization functions */
typedef  struct  _init_p
(
        int  path;                                      /* Path to the device */
        unsigned  char  *tbuf;                  /* Buffer for CSD handling */
        int  fa,fb,la,lb;                               /* FCTs and LCTs */
)  init_parms;
        /* Functions for CSD handling */

extern  char  *find_pat(),  *find_pat(),  *my_csd_devname(),  *my_csd_devparam();
```

```
/*****************************************************************************
*  Function:   init_dispatch()
*  Purpose:    Simply call the right function as indicated in the passed code
*  Passed: Function code and pointer to parameter block
*  Returned:   Return value from called function or SYSERR if invalid function
*  code
*****************************************************************************/
int  init_dispatch( funccode, parms )
int  funccode;
init_parms  *parms;
{
        int  ret=0;

        switch( funccode )
        {
           case  INIT_AUDIO:                ret = init_audio( parms );     break;
           case  INIT_VIDEO:                ret = init_video( parms );     break;
           case  INIT_DISC:                 ret = init_disc( parms );      break;
           case  INIT_PTR:                  ret = init_ptr( parms );       break;
           default:                         ret = SYSERR;                   break;
        }
        return( ret );
}


/*****************************************************************************
*  Function:   init_audio()
*  Purpose:    Initialize the audio device
*  Passed: Pointer to parameter block
*  Returned:   0 or SYSERR
*****************************************************************************/
int  init_audio( aparm )
init_parms  *aparm;
{
        register  char  *devnam;

        if ( (devnam = my_csd_devname( DT_AUDIO, 1, aparm->tbuf )) == NULL )
           return( SYSERR );
        if ( (aparm->path = open( devnam, S_IREAD|S_IWRITE )) == SYSERR )
           return( SYSERR );
        return( 0 );
}

/*****************************************************************************
*  Function:   init_video()
*  Purpose:    Initialize the video device, create and link FCTs and LCTs
*  Passed: Pointer to parameter block
*  Returned:   0 or SYSERR
*****************************************************************************/
int  init_video( aparm )
init_parms  *aparm;
{
        register  char  *devnam, *dp;
        register  int  laz;
                                /* Open the audio device, obtain parameters */

        if ( (devnam = my_csd_devname( DT_VIDEO, 1, aparm->tbuf )) == NULL )
           return( SYSERR );
        if ( (aparm->path = open( devnam, S_IREAD|S_IWRITE )) == SYSERR )
           return( SYSERR );
        if ( (dp = my_csd_devparam( devnam, aparm->tbuf )) == NULL )
           return( SYSERR );
```

```
                                                        /* Are we PAL or NTSC? */
        while ( *dp != '\0' )
            if (strncmp( dp++, PAL_625_STR, strlen(PAL_625_STR) ) == 0)
                break;
                                /* LCT is 280 if "625" is found, 240 otherwise */

        lsz = ( *dp == '\0' ) ? PAL_LCT : NTSC_LCT;
                                                        /* Create and link the DCP */

        if ( (aparm->fa = dc_crfct( aparm->path, PA, FCT_SIZE, BP_NORMAL ))
            == SYSERR )
            return( SYSERR );
        if ( (aparm->fb = dc_crfct( aparm->path, PB, FCT_SIZE, BP_NORMAL ))
            == SYSERR )
            return( SYSERR );
        if ( (aparm->la = dc_crlct( aparm->path, PA, lsz, BP_NORMAL ))
            == SYSERR )
            return( SYSERR );
        if ( (aparm->lb = dc_crlct( aparm->path, PB, lsz, BP_NORMAL ))
              == SYSERR )
            return( SYSERR );
        if ( dc_flnk( aparm->path, aparm->fa, aparm->la, 0 ) == SYSERR )
            return( SYSERR );
        if ( dc_flnk( aparm->path, aparm->fb, aparm->lb, 0 ) == SYSERR )
            return( SYSERR );
        return( 0 );
}

/*************************************************************************
*  Function:   init_disc()
*  Purpose:    Initialize the cd device
*  Passed:  Pointer to parameter block
*  Returned:   0 or SYSERR
*************************************************************************/
int init_disc( aparm )
init_parms  *aparm;
{
        register char *devnam;

        if ( (devnam = my_csd_devname( DT_CDC, 1, aparm->tbuf )) == NULL )
            return( SYSERR );
        if ( (aparm->path = open( devnam, S_IFDIR|S_IREAD )) == SYSERR )
            return( SYSERR );
        return( 0 );
}

/*************************************************************************
*  Function:   init_ptr()
*  Purpose:    Initialize the pointer device
*  Passed:  Pointer to parameter block
*  Returned:   0 or SYSERR
*************************************************************************/
int init_ptr( aparm )
init_parms  *aparm;
{
        char *devnam;

        if ( (devnam = my_csd_devname( DT_PTR, 1, aparm->tbuf )) == NULL )
            return( SYSERR );
        if ( (aparm->path = open( devnam, S_IREAD|S_IWRITE )) == SYSERR )
            return( SYSERR );
        return( 0 );
}
```

```
/*****************************************************************************
*       Filename:                                                audio_dispatch.c
*       Project:                Very simple example of using subroutine modules
*       internal  functions:              my_sm_out(),  my_sm_off(),  my_sm_stat(),
*                                      my_sm_info(),  my_sc_atten(),  my_sm_creat()
*       functions  known  externally:                            audio_dispatch()
*       author:                                                    Charles  Golvin
*       date:     Thu Aug 13 17:22:02 PDT 1992
*       SCCS:     1.1                                                      8/13/92
*
*       Note:     The following line identifies the functions that must be
*                 identified in the corresponding substart file.  It MUST  be
here
*                 for the automated substart generator to work correctly.
*
#   INCLUDED_SUBROUTINE_FUNCTIONS:                                  audio_dispatch

*****************************************************************************/

#if   !defined(NO_BVC)
static  char  SCCS_audio_dispatch_c[]  =  "@(#)audio_dispatch.c      1.1      8/13/92";
#endif

#include   <cdfm.h>

#define    SYSERR                                                          (-1)
#define    AUDIO_BASE                                                      0x10
#define    SM_OUT                                                    AUDIO_BASE
#define    SM_OFF                                                   AUDIO_BASE+1
#define    SM_STAT                                                  AUDIO_BASE+2
#define    SM_INFO                                                  AUDIO_BASE+3
#define    SC_ATTEN                                                 AUDIO_BASE+4
#define    SM_CREAT                                                 AUDIO_BASE+5

                        /*  Of course these meaningless names would be replaced  */
typedef  struct  _astuff
{
        unsigned  first,  second,  third,  fourth;
}  audio_stuff;


/*****************************************************************************
*  Function:   audio_dispatch()
*  Purpose:    Simply call the right function as indicated in the passed code
*  Passed:     Function code and pointer to parameter block
*  Returned:   Return value from called function or SYSERR if bad function code
*****************************************************************************/
int  audio_dispatch(  funccode,  parms  )
int  funccode;
audio_stuff  *parms;
{
        int ret = 0;

        switch(  funccode  )
        {
           case  SM_OUT:                          ret  =  my_sm_out(  parms
);break;
           case  SM_OFF:                          ret  =  my_sm_off(  parms  );break;
           case  SM_STAT:                         ret  =  my_sm_stat(  parms  );break;
           case  SM_INFO:                         ret  =  my_sm_info(  parms  );break;
           case  SC_ATTEN:                        ret  =  my_sc_atten(  parms  );break;
           case  SM_CREAT:                        ret  =  my_sm_creat(  parms  );break;
           default:                               ret  =  SYSERR;break;
```

```
        )
        return( ret );
}

/*****************************************************************************
*  Function:   my_sm_out()
*  Purpose:    Output a soundmap
*  Passed:  Pointer to audio parameter block
*  Returned:   0 or SYSERR
*****************************************************************************/
int my_sm_out( buf )
audio_stuff  *buf;
{
        if ( sm_out( buf->first, buf->second, buf->third ) == SYSERR )
           return( SYSERR );
        return( 0 );
}
```

```
/*****************************************************************************
*  Function:   my_sm_off()
*  Purpose:    Turn off soundmap output
*  Passed:  Pointer to audio parameter block
*  Returned:   0 or SYSERR
*****************************************************************************/
int my_sm_off( buf )
audio_stuff  *buf;
{
        if ( sm_off( buf->first ) == SYSERR )
           return( SYSERR );
        return( 0 );
}

/*****************************************************************************
*  Function:   my_sm_stat()
*  Purpose:    Return soundmap status block
*  Passed:  Pointer to audio parameter block
*  Returned:   0 or SYSERR
*****************************************************************************/
int my_sm_stat( buf )
audio_stuff  *buf;
{
        if ( sm_stat( buf->first, buf->second ) == SYSERR )
           return( SYSERR );
        return( 0 );
}

/*****************************************************************************
*  Function:   my_sm_info()
*  Purpose:    Return soundmap descriptor
*  Passed:  Pointer to audio parameter block
*  Returned:   0 or SYSERR
*****************************************************************************/
int my_sm_info( buf )
audio_stuff  *buf;
{
        if ( (int)((SoundmapDesc *)buf->third = sm_info( buf->first, buf-
>second ))
              == SYSERR )
           return( SYSERR );
        return( 0 );
}

/*****************************************************************************
*  Function:   my_sc_atten()
*  Purpose:    Set audio attenuation
*  Passed:  Pointer to audio parameter block
*  Returned:   0 or SYSERR
*****************************************************************************/
int my_sc_atten( buf )
audio_stuff  *buf;
{
        if ( sc_atten( buf->first, buf->second ) == SYSERR )
           return( SYSERR );
        return( 0 );
}
```

```
/*****************************************************************************
*  Function:   my_sm_creat()
*  Purpose:    Create a soundmap
*  Passed: Pointer to audio parameter block
*  Returned:   Return value from sm_creat
*****************************************************************************/
int my_sm_creat( buf )
audio_stuff  *buf;
{
        return( sm_creat( buf->first, buf->second, buf->third, buf->fourth ) );
}
```

```
*******************************************************************************
*  substart.init_dispatch.a
*     version of 'cstart' for initialization functions
*
*
        use  defsfile

Edition    equ   1
Typ_Lang   equ   (Sbrtn<<8)+Objct
Attr_Rev   equ   (ReEnt<<8)+0

        psect   subs_a,Typ_Lang,Attr_Rev,Edition,0,SubTable

* Resolve the references generated from Cstart - originate from -$8000
        org  -32768
_sttop     do.l  1                                                ; Stack top
_mtop:     do.l  1                                  ; current non-stack memory top
_stbot:    do.l  1                                    ; current stack bottom limit
errno:     do.l  1                                              ; global error holder
_totmem:   do.l  1                                         ; total data memory used
_sbsize:   do.l  1                                  ; top of process memory (sbrk)
_fcbs:     do.l  1                               ; file control block ptr (for CIO)
environ:   do.l  1                           ; environment pointer (for getenv())
_pathcnt:  do.w  1                  ; number of open paths at process creation
_sysglob:  do.l  1          ; pointer to system globals (if system-state process)

           * Since this subroutine module is less than 64 K, use word offsets

SubTable:  dc.w  Nam0,init_dispatch
        dc.w  0,0

Nam0   dc.b  "init_dispatch",0

        ends
```

```
*******************************************************************************
*   substart.audio_dispatch.a
*     version of 'cstart' for audio functions
*
*
        use  defsfile

Edition    equ   1
Typ_Lang   equ   (Sbrtn<<8)+Objct
Attr_Rev   equ   (ReEnt<<8)+0

        psect   subs_a,Typ_Lang,Attr_Rev,Edition,0,SubTable


* Resolve the references generated from Cstart - originate from -$8000
        org  -32768
_sttop     do.l  1                                                ; Stack top
_mtop:     do.l  1                                  ; current non-stack memory top
_stbot:    do.l  1                                    ; current stack bottom limit
_errno:    do.l  1                                              ; global error holder
_stmem:    do.l  1                                          ; total data memory used
_sbsize:   do.l  1                                   ; top of process memory (sbrk)
_fcbs:     do.l  1                                ; file control block ptr (for CIO)
environ:   do.l  1                             ; environment pointer (for getenv())
_pathcnt:  do.w  1                       ; number of open paths at process creation
_sysglob:  do.l  1          ; pointer to system globals (if system-state process)

* Since this subroutine module is less than 64 K, use word offsets

SubTable:  dc.w   Nam0,audio_dispatch
        dc.w  0,0

Nam0   dc.b  "audio_dispatch",0

        ends
```

```
#############################################################################
#
#  Filename:  cdi_example.mak
#       Makefile for cdi_example and its accompanying subroutine modules
#
#############################################################################

PROJ   =  /x/subroutine_modules
RDIR   =  $(PROJ)/rels
SRC    =  $(PROJ)/src
BIN    =  $(PROJ)/bin
LIBS   =  -l=$$(CLIB)/cdi.l  -l=$$(CLIB)/cdisys.l  -l=$$(CLIB)/clib.l
-l=$$(CLIB)/sys.l

all:  cdi_example  audio_dispatch  init_dispatch

audio_dispatch:  $(RDIR)/substart.audio_stuff.r  $(RDIR)/audio_stuff.r
        l68  $(RDIR)/substart.audio_stuff.r  $(RDIR)/audio_stuff.r  $(LIBS)
-o=$(BIN)/audio_dispatch

init_dispatch:  $(RDIR)/substart.init_stuff.r  $(RDIR)/init_stuff.r
        l68  $(RDIR)/substart.init_stuff.r  $(RDIR)/init_stuff.r  $(LIBS)
-o=$(BIN)/init_dispatch

$(RDIR)/substart.init_stuff.r:  $(SRC)/substart.init_stuff.a
        r68  -o=$(RDIR)/substart.init_stuff.r  $(SRC)/substart.init_stuff.a

$(RDIR)/init_stuff.r:  $(SRC)/init_stuff.c
        cc68  -s  -dNO_BVC  -r=$(RDIR)  $(SRC)/init_stuff.c

$(RDIR)/substart.audio_stuff.r:  $(SRC)/substart.audio_stuff.a
        r68  -o=$(RDIR)/substart.audio_stuff.r  $(SRC)/substart.audio_stuff.a

$(RDIR)/audio_stuff.r:  $(SRC)/audio_stuff.c
        cc68  -s  -dNO_BVC  -r=$(RDIR)  $(SRC)/audio_stuff.c

cdi_example:  $(SRC)/cdi_example.c
        cc68  $(SRC)/cdi_example.c  $(LIBS)  -f=$(BIN)/cdi_example
```

```
#!/bin/csh  -f

#############################################################################
#  Project  root

set  PROJ  =  "/x/subroutine_modules"

#############################################################################
#  This  is  the  subroutine  substart  template

set  template  =  $PROJ/src/substart.template

#############################################################################
#  Build  the  name  of  the  output  file,  report  what  it  will  be

set  tmp  =  $argv[1]:t
set  head  =  $tmp:r
unset  tmp
set  target  =  $PROJ/src/substart.$head.a
echo  ""
echo  "Making  new  substart  file  $target"

#############################################################################
#  Copy  the  template

cp  $template  $target

#############################################################################
#  Find  the  functions  in  the  subroutine  module,  report  them

set  TheString  =  "INCLUDED_SUBROUTINE_FUNCTIONS:"
set  dummy  =  `grep  $TheString  $argv[1]`
if  (  $#dummy  ==  0  )  then
        echo  -n  "Failed  to  find  key  string:  "
        echo  $TheString
        echo  "Giving  up"
        exit
endif
while  (  $dummy[1]  !=  $TheString  )
        shift  dummy
end
shift  dummy
echo  ""
echo  "Function  names  are  $dummy"
echo  ""

#############################################################################
#  Now  build  the  subroutine  function  table
#  In  this  example  the  offsets  are  all  words  -  to  use  longwords  simply  change
#  the  three  instances  of  "dc.w"  to  "dc.l"
#
#

set  idx  =  0
set  str  =  `echo  SubTable:      dc.w  Nam$idx`
echo  -n  $str  >>$target
set  str  =  `echo  ,$dummy[1]`
echo  $str  >>$target
```

```
foreach  func  ($dummy[2-])
@        idx++
         echo  -n  "  "  >>$target
         set  str  =  `echo  dc.w  Nam$idx,$func`
         echo  $str  >>$target
end
echo  "  dc.w  0,0"  >>$target
echo  ""  >>$target
set  idx  =  0
set  count  =  $#dummy
while  ($idx  <  $count)
         set  str  =  `echo  Nam$idx  dc.b`
         echo  -n  $str  >>$target
         echo  -n  "  "  >>$target
         echo  -n  '"'  >>$target
         echo  -n  $dummy[1]  >>$target
         echo  '",0'  >>$target
         shift  dummy
@        idx++
end
echo  ""  >>$target
echo  "  ends"  >>$target
exit
```