A C programming model for OS/2 device drivers

by D. T. Feriozi

The recent growth in the number of new and different types of devices for use with personal computers has challenged software engineers to plan new and better ways of developing software to run the devices. For Operating System/2® (OS/2®) device drivers, an improvement would be to code in a high-level language rather than to use assembly language. A practical and proven method of writing OS/2 device drivers in the C programming language is presented here. The C language was chosen because of its documented suitability as a systems programming language and because of its universal availability for use on small systems.

The software development process has evolved over the years into a highly structured software engineering discipline. Early programmers manipulated switches on the computer in order to perform useful work. Switches were later replaced by binary code that could be fed into the computer and executed. The use of assembly language was a major breakthrough. Assemblers were able to translate English-like instructions into code that a computer could execute. High-level languages introduced abstraction and machine-independence into the process. Programs written in a high-level language more closely follow human thought processes and modes of expression. As a result, they are easier to produce and maintain than assembly language programs. In addition, the overall work effort is reduced because a high-level language program can be used on many different types of machines, as long as a compiler is available for each machine to which the program is targeted.

While application programmers have embraced high-level languages with unbridled enthusiasm, systems programmers have been slow to move away from assembly language. The problem has been that many programmers have perceived most high-level languages as lacking the power and efficiency of assembly language. In many circumstances, an appropriate high-level language is not available that maps into the machine code. The C programming language contains a rich set of operators that closely correspond to the assembly language operators common to most computers. Compiler technology has advanced to the point that modern optimizing compilers may actually generate code that, overall, is more efficient than code produced by the average assembly programmer. This is not to say that assembly language is obsolete. It is still required for certain operations and in certain circumstances; for example, certain real time, or machine-specific tasks require assembly code. A mixed programming model is the best choice for systems usage. The bulk of the code is written in C, and assembly language is used only where necessary.

The UNIX® operating system is perhaps the most well-known example of system code that is written in C. The UNIX kernel, device drivers, and utilities are almost entirely written in C, with only the lowest-level, machine-dependent code written in assembly language. The wide acceptance and success of UNIX is evidence that the C programming language is suitable for use in the de-

velopment of device drivers and other system programs.

The general outline for a C language programming model for Operating System/2® (OS/2®) device drivers is presented in this paper. The intention is to provide broad concepts as well as the specific details necessary to allow the development of OS/2 device drivers using the C programming language. This paper addresses only the implementation language specific issues of OS/2 device drivers. For a general introduction to the basic OS/2 device driver model, see A. M. Mizell.[1] The reader is also encouraged to refer to the OS/2 technical reference[2] for more information about OS/2 device drivers. Our programming model was developed for use with the IBM C/2™[3] compiler. It should translate to the Microsoft® C compiler with very little modification. The general concepts should work with any C compiler; however, specific details of implementation may vary somewhat from one compiler to another. Consult the specific compiler's reference manual to resolve any discrepancies.

OS/2 device drivers

OS/2 device drivers have traditionally been written in assembly language. The model for OS/2 device drivers is based on the DOS device driver model, converted to a multitasking environment. This is an oversimplification, but it does explain why the structure of an OS/2 device driver is not compatible with the normal C program structure. At the time DOS was being developed, the C programming language was not a major force in personal computers. As a result, the specification of the model for DOS device drivers was planned for assembly language, disregarding possible conflicts with C programming models.

An OS/2 device driver consists, primarily, of two segments, the code segment and the data segment. The data segment must physically appear first in the device driver load file so that its device driver header will be conveniently accessible to the OS/2 kernel. This causes a problem when writing a device driver in C because the normal ordering of C programs is code first, then data.

Another problem encountered when developing device drivers in C is that a normal C program includes many routines that establish an executable environment. These startup routines are neither desirable nor necessary in a device driver, and must be eliminated. The C programming language was originally developed for systems programming. As such, it had few of the convenient services that application programmers demand, such as input/output, screen handling, and so forth. As these features were added to standard implementations of C, it became more difficult to use C for some types of systems programming in certain environments.

An OS/2 device driver consists, primarily, of two segments, the code segment and the data segment.

It is the purpose of this paper to show specifically how the incompatibilities between the OS/2 device driver model and the C programming model can be resolved to allow OS/2 device drivers to be written in C. The concepts presented can be generalized and used as a basis for developing C programming models for any type of system code.

The C programming model

The current OS/2 model for device drivers is based on the assumption that they will be written in assembly language. As a result, it was not possible to write an OS/2 device driver completely in C, using the IBM C/2 compiler. The programming model discussed in this paper provides the ability to develop OS/2 device drivers largely in C. The exact ratio of C code to assembly code in the device driver is unimportant. The primary concern is that sections of code that require high-level data and control structures are written in C, whereas sections of code that require hardware access or very high performance are written in assembly.

Problems to be solved

The C compiler creates object modules that are linked together to form an executable program.

The problem is that a device driver is not an executable program. It is a special type of file that resembles what is referred to as a library module. The programming model used for an OS/2 device driver must be able to do the following:

- Order the segments with data first, then code.
- Eliminate the C startup routines.
- Eliminate the C run time.
- Convert a register calling convention into a stack calling convention.
- Discard the initialization code.

These tasks are accomplished by an assembly entry module in conjunction with a set of link and compile flags.

Device driver structure

The device driver source code consists of the following modules, at a minimum:

- OS2DD.ASM—main entry module
- COMMANDS.C—device driver commands
- IOCTL.C—generic ioctl commands
- INT.C—interrupt handler
- DEVHLP.ASM—device helper interface
- INIT.C—device driver initialize command

Each of these modules is described separately in order to build a picture of the device driver from its components.

OS2DD.ASM. This is the main module of the device driver. It performs the following functions:

- Declares, groups, and orders all segments
- Provides the device driver header
- Provides the main entry point
- Routes the device driver command

This module begins with the following assembly code:

```
DGROUP      group   _DATA, _BSS, CONST, INIT_DATA
CGROUP      group   _TEXT, INIT_CODE

_DATA       segment word PUBLIC 'DATA'
            assume  ds:DGROUP
_DATA       ends

_BSS        segment word PUBLIC 'BSS'
            assume  ds:DGROUP
_BSS        ends

CONST       segment word PUBLIC 'CONST'
            assume  ds:DGROUP
CONST       ends
INIT_DATA   segment word PUBLIC 'INIT_DATA'
            assume  ds:DGROUP
INIT_DATA   ends

_TEXT       segment word PUBLIC 'CODE'
            assume  cs:CGROUP
_TEXT       ends

INIT_CODE   segment word PUBLIC 'INIT_CODE'
            assume  cs:CGROUP
INIT_CODE   ends
```

The default compiler-produced segments are grouped and ordered by this code. Two additional segments are included to allow the initialization code to be discarded. The INIT_DATA segment is an assembly segment that contains the discardable data. The INIT_DATA segment is needed because the compiler splits the data into three segments, making it impossible to define a clear cutoff point from the C code. The INIT_CODE segment contains any assembly code that is to be discarded.

A device driver is not an executable program. It is a special type of file.

The address of the end of data is defined somewhere in the INIT_DATA segment. This segment is a convenient place to define data objects that can be sized at initialization time. In this case, the end-of-data address would immediately follow the last dynamically allocated data object. If no data need to be defined at run time, the end of data would simply be a label at the beginning of the INIT_DATA segment. Additional discardable data segments may be declared after INIT_DATA, if needed.

The address of the end of code is defined in the TEXT segment. This is because most of the initialization code is written in C. Since the INIT_CODE segment follows the TEXT segment, it is discarded entirely. The INIT_CODE segment

may be empty if no assembly initialization routines are required.

It should be clear at this point that the proper definition and ordering of segments is crucial to allowing the initialization code and data to be discarded. In addition, it is necessary to specify the sequence of module linking in order to be able to properly size the TEXT segment. The initialization code is sequestered in the INIT.C module for this reason. The link statement must specify INIT.OBJ as the last link module so that no code will follow it and be discarded by accident. The link statement should be something like:

The proper definition and ordering of segments is crucial to allowing the initialization code and data to be discarded.

```
link OS2DD.OBJ COMMAND.OBJ INT.OBJ
     DEVHLP.OBJ INIT.OBJ
```

The main module must be specified first, and the init module must be specified last. The order of the other modules is not important.

The second major function of the OS2DD.ASM module is to define the device header at the beginning of the device driver. The device driver header is defined as the first data object in the DATA segment. Since the DATA segment is the first segment of the DGROUP, and the DGROUP is the first group of the device driver, and OS2DD is the first link module, the device header appears at offset zero of the device driver, as required by the OS/2 device driver model.

The strategy routine entry point is also defined in the OS2DD.ASM module. The entry point to the device driver must be defined in an assembly module in order to convert the OS/2 register parameter passing protocol to a stack-based protocol that can be recognized by C functions. Since the address of the request packet is contained in the ES:BX register pair (microprocessor register designation), pushing the ES register followed by the BX register converts the request packet address to a far pointer. If a C function is now called, it will receive the far pointer to the request packet as its parameter. The called function could then route the request to a C worker routine to actually process the request.

Actually, device driver command verification and routing can be executed more efficiently in the assembly module, without loss of code clarity. An assembly call table is used to provide the parameter used to indirectly call the C routine that satisfies the OS/2 request. The strategy procedure of OS2DD does the following:

1. Verify the OS/2 command code.
2. Push ES:BX onto the stack.
3. Indirectly call CommandTable[command].
4. Pop ES:BX.
5. Set the DONE bit of the status that was returned.
6. Move the status code to the request packet.
7. Return to OS/2.

The OS2DD.ASM module is almost entirely reusable as source code. Only minor changes would be needed in the device header. For instance, the device name and attribute bits may change from one driver to another.

Refer to Appendix A for a sample copy of OS2DD.ASM. Note the conditionally compiled code that is marked as UNIT_TESTING. This code enables the device driver to be unit tested[4] as a normal executable program by using a symbolic debugger.

COMMANDS.C. The COMMANDS.C file contains C functions that correspond to the OS/2 device driver commands. These are the entry points that are called indirectly by the strategy procedure through its table of OS/2 command codes. Example function names are:

- MediaCheck (MediaReqBlk far * RequestBlock)
- Read (ReadReqBlk far * RequestBlock)
- DeviceOpen (OpenReqBlk far * RequestBlock)

Each function can access its OS/2 request block through the far pointer that is passed to it from the strategy procedure. The following duties are performed:

1. Satisfy the OS/2 request.
2. Set any necessary values in the command-specific part of the request header.
3. Return the completion status code to the strategy routine.

The status code is returned to the routine containing the strategy procedure, rather than being set directly for two reasons. One is that it makes for a clean interface, with the C function satisfying a request and then returning the result of that request to its caller. The other reason is that it is more efficient to set the status from the assembly module. Access through far pointers is somewhat expensive from C routines.

Extra care should be taken when coding an interrupt handler.

IOCTL.C. The IOCTL.C module contains the entry point for the Generic Ioctl command. This command and the Initialize command are the only two OS/2 device driver commands that are not included in COMMANDS.C. A separate module is used for the Generic Ioctl command because it requires further routing to a worker routine. Also, the Ioctls comprise a good functional unit that can be separated from the rest of the code.

The Generic Ioctl function must first verify the request packet parameters and then route the Ioctl request to another C routine. Routing can be done by a SWITCH statement, a call table, or a combination of the two. A SWITCH statement alone is appropriate if the driver only needs to handle a few different Ioctl commands. A common occurrence is to have only a few different Ioctl categories, with each one containing many Ioctl codes. In this case, a SWITCH statement on the Ioctl category combined with call tables for the Ioctl codes of each Ioctl category results in compact, efficient, clear code. Call-table definition and indirect function calling can of course be done directly in C.

Except for the additional level of indirection, the IOCTL module performs exactly the same duties as the COMMANDS module. That is:

1. Satisfy the OS/2 request.
2. Set any necessary values in the command-specific part of the request header.
3. Return the completion status code.

The extra function call means that the Generic Ioctl routine must pass its argument along to the Ioctl worker routine. In turn, the worker routine must return the status completion code to the Generic Ioctl function, which passes it back to the strategy routine. This may seem like a lot of shuffling of the status code; however, the C compiler handles function return values efficiently so that very little penalty is incurred.

INT.C. The INT.C module contains the interrupt handler required by the OS/2 device driver model. By convention, the OS/2 kernel takes care of all the details of the context switch before passing control to the interrupt handler. This means that the entry point for the interrupt handler may be coded as a C subroutine. The interrupt handler must:

1. Determine ownership of the interrupt.
2. Service the interrupt.
3. Issue an end-of-interrupt through a DevHlp call.
4. Return ownership status to the OS/2 kernel.

Since the interrupt handler must be very efficient by its nature, the programmer may be tempted to code it in assembly language rather than in C. Experience has shown that in most cases this is not necessary. In any case, extra care should be taken when coding an interrupt handler. Restricting the functionality of the interrupt handler will result in greater performance gains than those that may be obtained by resorting to coding in assembly language. The most important thing to remember is that the interrupt handler should only do what is absolutely necessary to dismiss the interrupt. Usually this function is driven by a performance requirement.

DEVHLP.ASM. The DEVHLP.ASM assembly module is necessary because the Device Helper routines provided by the OS/2 kernel use a register parameter passing protocol. The OS/2 DevHlp routines provide services required by all device drivers. This module contains short assembly routines that are callable from the main body of the C code. Their simple function is to:

1. Load stack-based parameters into registers.

2. Make the indirect DevHlp call to OS/2.
3. Return the DevHlp return value to the calling C function.

Some of these routines may provide additional services. For instance, when calling the BLOCK DevHlp, it is usually necessary to disable interrupts and verify that the condition that requires blocking is still present. Before returning from the BLOCK subroutine, it is also advisable to verify that the blocking condition is no longer present. That is, check to be sure that this is not a spurious wakeup. The assembly subroutine can handle these simple chores very easily.

The reference manual for specific C compilers usually contains details on how to write assembly subroutines that are callable from C routines. Briefly, parameters are referenced indirectly through the BP register. Word-size return values are placed in the AX register and double-word size return values are passed in the DX:AX register pair.

Figure 1 illustrates how the assembly code provides a bridge between the device driver and the rest of the operating system.

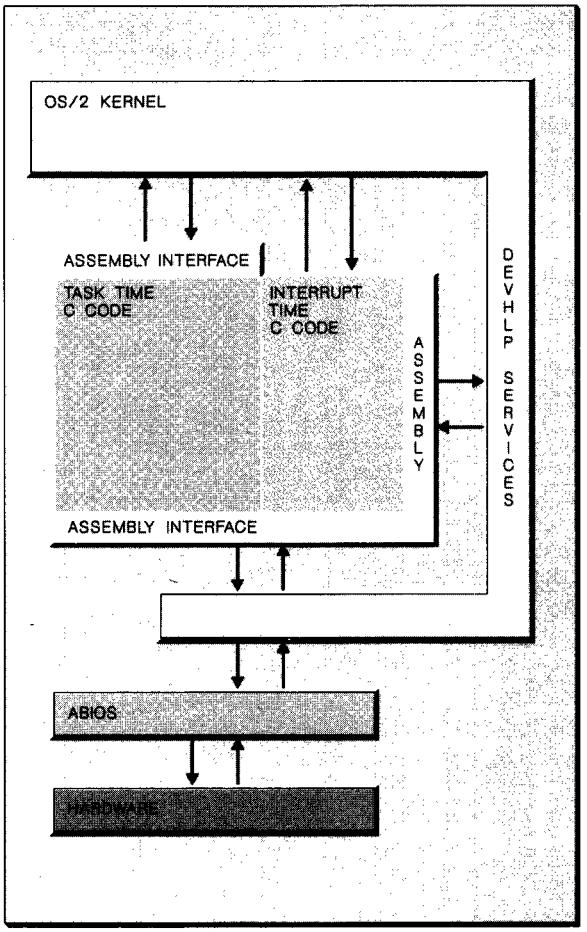
INIT.C. The INIT.C module contains all of the C language initialization code for the device driver. The Initialize command is segregated in this module so that it can be discarded after initialization time. Any auxiliary routines used only during initialization are also placed in this module and discarded after use. Some examples of INIT functions are:

- Display status and error messages.
- Find and register devices that are active.
- Initialize devices.
- Register the interrupt handler with OS/2.
- Set the end address of code and data.

As is the case with all of the OS/2 device driver commands, the Initialize command is called indirectly by the strategy routine, which passes it a far pointer to the request block. The Initialize command is required to return its status completion code to the strategy routine upon exit.

Initialization code is discarded by the OS/2 kernel. External data that are defined in this module are not discardable. Data must be defined in the INIT_DATA segment in order for the data to be discarded after use.

Figure 1 Device driver in relation to the system



Discardable data that are defined in an assembly module are made visible to C routines by the PUBLIC declaration and by using the proper naming conventions. C compilers generally prefix an underscore to all external names. Therefore, a data object would be declared in the C file as:

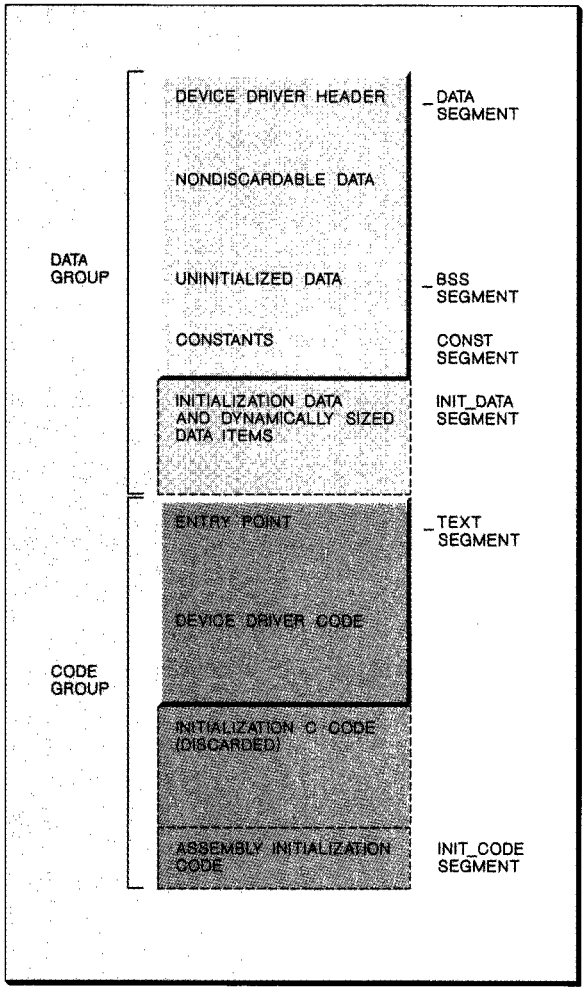
```
extern int      data_object
```

and defined as:

```
                public  _data_object
_data_object    dw
```

in the assembly module. This naming convention also applies to assembly subroutines that are to be

Figure 2 Device driver structure



called from C code, and to C subroutines that are called from assembly code.

Figure 2 summarizes the overall structure of the device driver.

Structure summary. The developed device driver is actually an assembly program with C language subroutines. The normal C ordering of segments is reversed so that the device header will appear at the beginning of the device driver. The linkage ordering is further controlled in order to allow the initialization code to be discarded after use. This is particularly important since the initialization code usually tends to be large, and low-end memory is at a premium in the OS/2 environment.

Though the device driver is technically an assembly program, it appears to be a C language program to the programmer. Most of the code is written in C. The logic of the assembly code is mainly trivial, easy-to-understand bookkeeping. In addition, both assembly modules are reusable. The assembly code command router never changes. Unsupported commands can be handled in the COMMANDS module as empty C functions that return UNKNOWN COMMAND to the strategy routine. The only possible change to the DEVHLP module would be to edit out unused DevHlp interface routines in order to conserve space.

Compile and link issues

The C startup routines as well as the C run time can be eliminated by specifying that default libraries should not be searched. With the IBM C/2 compiler, this is accomplished by using compile and link control flags. The compile flag is /Zl, and the link flag is /NOD.

It should be apparent that the C startup routines have no place in a device driver. They establish the environment for an executable program, which a device driver is not. The reason for not using the C run time may not be so obvious. The answer is that any library code that is used will be linked after the main body of the program. This means that it will be discarded along with the initialization code. Also, the library routines may contain stack probes or system calls that will fail in the device driver environment.

The only library that can safely be used is the OS/2 application programming interface (API) library, DOSCALLS.[2] The device driver uses a limited subset of the OS/2 API only during initialization. After initialization, the DevHlp interface is used. This works out well; the DOSCALLS information is discarded along with the rest of the initialization code.

The other recommended compile flags for use with the IBM C/2 compiler are: Gs, G2, Zp, and Ox.

The Gs option instructs the compiler not to generate stack probes with the code. The stack probe code is designed to work with the stack that is created by the C startup routines. The device driver is linked without a stack segment, since

OS/2 provides a stack for use at run time. Under these conditions, the stack probes could not be expected to work properly.

The G2 option causes the compiler to generate Intel® 80286-specific code. This seems like a good idea, since OS/2 is not designed for prior-level processors.

The Zp option directs the compiler to pack data structures. That is, structures are created with contiguous members. If this option is not specified, the compiler may generate structures that contain holes in order to align data items on their boundary type. Device driver data structures must be packed because the OS/2 request packets are packed.

The Ox option causes the compiler to maximally optimize the code. A good optimizing compiler can produce surprisingly efficient code if given a free rein to do so. For example, with this option specified, the IBM C/2 compiler will convert a data movement FOR loop statement in C to a REP MOVSW assembly instruction.

It is also suggested that the /ML option be used with the assembly modules, and that the /NOI link option be used. Using these options will enforce case sensitivity of names within the assembly modules and across the link process. Case sensitivity must be preserved in order to ensure compatibility with the C language modules, since names in C are always case-sensitive.

Performance considerations

Since an OS/2 device driver is system-level code in a multitasking environment, every effort should be made to ensure that the device driver code is efficient, without sacrificing clarity in the source code. The C programming language is very flexible, providing many different ways to accomplish the same task. This makes it incumbent on the programmer to make the best choices for the application at hand. Experienced C programmers have learned many tricks; the following are some tips for the beginner.

- Use register variables, especially for pointers and counters.
- Inspect the compiler assembly listing. This is useful for finding possible inefficiencies in the code. A knowledge of how the compiler handles different constructs makes it easier to write efficient C code.
- Do a performance analysis of the code and recode any bottlenecks in assembly if it will help. The original C code serves as pseudocode and documentation.
- Pay particular attention to far-pointer usage. Function calls will cause the compiler to forget a selector, resulting in the reloading of a segment register. The use of local temporary variables to store values until they can all be moved through the same far pointer is sometimes helpful.

These really are minor optimization points that have little effect on the overall performance of the device driver. The choice of the algorithms that are used always plays the major role in system performance. The use of a high-level language allows the programmer to concentrate on algorithm development because the compiler keeps track of so many of the low-level details of the code. In many cases, code developed in the C language will actually perform better than similar code that was developed directly in assembly language. The greater the size and complexity of the project, the more likely it is that the compiler will generate the more efficient code.

The C programming language combines the best features of a high-level language with the best features of a low-level language to produce a general-purpose programming language. The high-level data and control constructs make the structure and function of the code visible, while relieving the programmer of the tedious details associated with assembly language programming. The close correlation with machine-level instructions allows the compiler to produce compact and efficient code. These qualities make the C programming language a good choice for OS/2 device driver development.

Appendix A: Sample OS2DD.ASM file

```
title    Sample OS2 Device Driver
page     80,132

;******************************************************************************
;*
;* SOURCE FILE NAME:  OS2DD.asm
;*
;******************************************************************************

.286p
.seq

rh                      struc                   ; OS2 request header
rh_length               db    ?
rh_unit                 db    ?
rh_command              db    ?
rh_status               dw    ?
rh_reserved             dd    ?
rh_queue_linkage        dd    ?
rh                      ends

PRIVATE          macro                          ; dummy macro to
                 endm                           ;  label private

DONE             equ   0100h                    ; done bit in status

;* C functions to handle os2 requests

extrn            _Initialize:near
extrn            _MediaCheck:near
extrn            _BuildBPB:near
extrn            _Read:near
extrn            _PeekInput:near
extrn            _InputStatus:near
extrn            _InputFlush:near
extrn            _Write:near
extrn            _OutputVerify:near
extrn            _OutputStatus:near
extrn            _OutputFlush:near
extrn            _DeviceOpen:near
extrn            _DeviceClose:near
extrn            _RemovableMedia:near
extrn            _GenericIoctl:near
extrn            _ResetMedia:near
extrn            _GetLogicalDevice:near
extrn            _SetLogicalDevice:near
extrn            _Deinstall:near
extrn            _PortAccess:near
extrn            _PartitionableFixedDisks:near
extrn            _GetFixedDiskMap:near
extrn            _NonCachingRead:near
extrn            _NonCachingWrite:near
extrn            _NonCachingWriteVerify:near
extrn            _PrepareForSysShutdown:near
```

```
extrn           _UnknownCommand:near

DGROUP          group  _DATA, _BSS, CONST, INIT_DATA

CGROUP          group  _TEXT, INIT_CODE

_DATA           segment word PUBLIC 'DATA'
                assume  cs:CGROUP, ds:DGROUP, ss:nothing, es:nothing
_DATA           ends

_BSS            segment word PUBLIC 'BSS'
                assume  cs:CGROUP, ds:DGROUP, ss:nothing, es:nothing
_BSS            ends

CONST           segment word PUBLIC 'CONST'
                assume  cs:CGROUP, ds:DGROUP, ss:nothing, es:nothing
CONST           ends

INIT_DATA       segment word PUBLIC 'INIT_DATA'
                assume  cs:CGROUP, ds:DGROUP, ss:nothing, es:nothing
INIT_DATA       ends

_TEXT           segment word PUBLIC 'CODE'
                assume  cs:CGROUP, ds:DGROUP, ss:nothing, es:nothing
_TEXT           ends

INIT_CODE       segment word PUBLIC 'INIT_CODE'
                assume  cs:CGROUP, ds:DGROUP, ss:nothing, es:nothing
INIT_CODE       ends

;************************ DATA SEGMENT ****************************************

_DATA           segment

                PUBLIC  _Device_header, _Device_name
_Device_header  label   word
                dd      -1                      ; next device
                dw      1000000100000000b       ; character DD, level 2
                dw      strategy                ; entry point
                dw      0                       ; IDC entry point
_Device_name    db      'OS2DD$  '              ; device name or numberofunits
                dd      0                       ; reserved
                dd      0                       ; reserved

                PUBLIC  _DeviceHelp
_DeviceHelp     dd      CGROUP:dummy_DevHlp     ; ptr to DevHlp entry
                                                ; initialized to dummy for
                                                ;  debugging

; ============================================================================

CommandTable    label   word                    ; call table for os2 commands
                dw      _Initialize
                dw      _MediaCheck
                dw      _BuildBPB
```

```
                dw      _UnknownCommand
                dw      _Read
                dw      _PeekInput
                dw      _InputStatus
                dw      _InputFlush
                dw      _Write
                dw      _OutputVerify
                dw      _OutputStatus
                dw      _OutputFlush
                dw      _UnknownCommand
                dw      _DeviceOpen
                dw      _DeviceClose
                dw      _RemovableMedia
                dw      _GenericIoctl
                dw      _ResetMedia
                dw      _GetLogicalDevice
                dw      _SetLogicalDevice
                dw      _Deinstall
                dw      _PortAccess
                dw      _PartitionableFixedDisks
                dw      _GetFixedDiskMap          ; cmd 23
                dw      _NonCachingRead
                dw      _NonCachingWrite
                dw      _NonCachingWriteVerify
                dw      _UnknownCommand
                dw      _PrepareForSysShutdown    ; cmd 28
                dw      _UnknownCommand
                dw      _UnknownCommand
                dw      _UnknownCommand
                dw      _UnknownCommand
                dw      _UnknownCommand           ; cmd 33

MAX_IN_TABLE    equ     33

_DATA           ends

;************************ END DATA SEGMENT ***************************************

;************************* TEXT SEGMENT ******************************************

_TEXT           segment

entry_point:


;*****************************************************************************
;*
;* SUBROUTINE NAME:      strategy
;*
;* FUNCTION:     To call the C routine to process the request
;*
;*****************************************************************************


                PRIVATE
strategy        proc    far
```

```
                mov     al, es:[bx].rh_command ; call request
                cmp     al, MAX_IN_TABLE
                jbe     cmd_in_range           ; in the table
                mov     al, MAX_IN_TABLE       ; unknown command
cmd_in_range:   xor     ah, ah
                shl     ax, 1                  ; for word offset in table
                mov     si, ax
                push    es                     ; fcn parameters on stack
                push    bx                     ; to C, far ptr to rh
                call    CommandTable[si]       ; C function does command
                pop     bx
                pop     es
or              ax,     DONE
                mov     es:[bx].rh_status, ax  ; set status

                ret

strategy        endp

;************************ dummy_DevHlp ***************************************

ifdef UNIT_TEST
                PUBLIC  dummy_DevHlp
endif

dummy_DevHlp    proc    far                    ; mostly for debugging
                ret
                even                           ; to line up parts of _TEXT
dummy_DevHlp    endp

;*******************************************************************************

_TEXT           ends

;************************ END TEXT SEGMENT ***************************************

;************************ INIT CODE SEGMENT ***************************************

INIT_CODE       segment       ; init segment code is discarded
                              ; assembly init functions go here
                              ; they are linked after init.obj
                              ; init.obj forward is discarded

;*******************************************************************************
;*
;* SUBROUTINE NAME:    AllocGDTselector
;*
;* FUNCTION:     Allocate an array of GDT selectors
;*
;*******************************************************************************

                PUBLIC  _AllocGDTselector

_AllocGDTselector       proc    near

                push    bp
```

```
                mov     bp, sp
                push    di

                push    ds
                pop     es                      ; selector of array
                mov     di, [bp+4]              ; offset of array
                mov     cx, [bp+6]              ; number of selectors
                mov     dl, 2Dh                 ; AllocGDTselector
                call    [_DeviceHelp]

                pop     di
                pop     bp
                ret

_AllocGDTselector       endp

INIT_CODE       ends

;************************* END INIT CODE ************************************

;************************* INIT DATA SEGMENT ********************************

INIT_DATA       segment                         ; any init data can be put here
                                                ; and referenced from C
                                                ; this segment is discarded

                PUBLIC  _End_of_data, _Init_message, _Init_message_length
                PUBLIC  __acrtused

_End_of_data            label   byte
__acrtused              dw      1               ; for the MS compiler
_Init_message           db      'Bad or missing DEV.MSG', 0Dh, 0Ah, 0
                        db      256 dup (?)     ; space for DOSGETMESSAGE
_Init_message_length    dw      256

;======================== UNIT_TESTING ============================

ifdef   UNIT_TEST
                PUBLIC  local_stack
local_stack     dw      2000 dup ('s')          ; stack for using codeview
                                                ; sp is set to TOS
                                                   automatically
endif

; ============================================================

INIT_DATA       ends

;************************* END INIT DATA ************************************

                end entry_point
```

Cited references and note

1. A. M. Mizell, "Understanding Device Drivers in Operating System/2," *IBM Systems Journal* **27**, No. 2, 170–184 (1988).
2. *IBM Operating System/2 Programming Tools and Information Version 1.2*, IBM Corporation (1991); available through IBM branch offices.
3. *IBM C/2 Version 1.10 Reference Manual*, IBM Corporation (1988); available through IBM branch offices.
4. Unit testing is the earliest phase of testing. It occurs in a scaffolded environment, testing the logic of the code in isolation from the rest of the system.

General references

IBM MASM/2 Version 1.10 Reference Manual, IBM Corporation (1988); available through IBM branch offices.

IBM Operating System/2 Technical Reference Version 1.1, Volume I, IBM Corporation (1988); available through IBM branch offices.

Kernighan and Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ (1988).

Dan T. Feriozi *IBM Entry Systems Division, 1000 N.W. 51st Street, Boca Raton, Florida 33429.*

Mr. Feriozi is a programmer in the Engineering Software Development Laboratory in Boca Raton. He is currently responsible for the design and development of SCSI device drivers for OS/2 as well as for DOS. Mr. Feriozi is widely recognized within IBM for his expertise in the field of device driver development. His models are currently being used on IBM sites in Japan, Canada, and Europe as well as in the United States. He has received several awards including a Division Award in recognition of excellence and achievement from the Entry Systems Division of IBM. Mr. Feriozi received his B.S. degree in chemistry from Georgetown University in Washington, D. C. He also holds a master's degree in computer science from Florida Atlantic University in Boca Raton, and is currently working toward a Ph.D. in computer science.

Reprint Order No. G321-5438.