A split model for OS/2 SCSI device drivers

by D. T. Feriozi

The concept of splitting one logical device driver into two or more physical units is presented. The specific case of an Operating System/2® (OS/2®) SCSI (Small Computer System Interface) device driver is used as an example. The primary reason for splitting the device driver is to reduce the development effort required to produce new SCSI device drivers. Common code is isolated in a separate driver in order to prevent its reinvention as each new SCSI device becomes available. Additional benefits are that the overall device driver size is reduced, and the performance of the SCSI subsystem is enhanced. The complete separation of the upper- and lower-level drivers provides the ability to replace one of the device drivers without affecting any of the other components of the system. This is particularly important because it enables backward compatibility for older device drivers, while allowing for the support of emerging technology.

The function of a device driver is to provide a bridge between an operating system and a peripheral device. As a result, a new device driver must be written for each new device and for each operating system to which the device can be attached. A substantial development effort is required if several devices must be supported by several different operating systems. Code layering can be used to provide a partial solution to this problem for SCSI (Small Computer System Interface) devices in the Operating System/2* (OS/2*) environment. Figure 1 illustrates three possible approaches that could be taken to the layering of code in SCSI device drivers. In Figure 1A, a new device driver must be written to interface with the SCSI interface, supported by the controller microcode.

Since it connects an operating system to a peripheral device, a device driver contains information that is both operating-system-specific and hardware-specific. The device driver must accept requests from the operating system, and translate those requests into operations that are performed by the device adapter and by the device. These two levels of control—operating system and device—tend to polarize a device driver into two corresponding sections. Some parts of the device driver are more concerned with interfacing with the operating system, while other parts of the device driver primarily act to control the operation of the device.

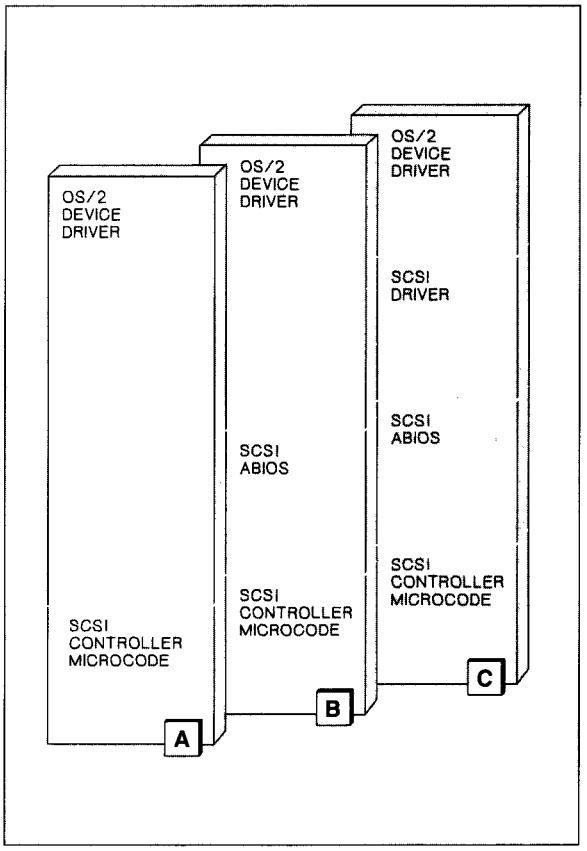
In the extreme case, the logical functional split of a device driver can be extended to a physical separation into two distinct, cooperating entities. The manner and method of separation depends on both the operating system involved and the type of device that is targeted. The OS/2 operating system is well-suited to a split device driver model for two reasons. First, OS/2 contains a built-in method to allow two different device drivers to communicate. Second, since OS/2 is a multitasking operating system, it relies heavily on overlapped input/output (I/O) for many different devices. The SCSI device protocol also lends itself to a split device driver model because it contains a standard command set that is implemented by all SCSI devices.

Splitting a device driver into two parts is a logical extension of the personal computer concept of the Basic Input/Output System (BIOS)[1] that is used in the single-threaded DOS operating system. The BIOS code that is a part of the personal computer firmware can actually be thought of as the hardware-dependent part of a device driver. The BIOS code presents a standard device interface to the device driver. If a device driver invokes device services indirectly through the BIOS interface rather than by manipulating the adapter or the device directly, it is effectively isolated from changes in the underlying hardware. If the hardware is modified or even replaced entirely, only the BIOS needs to be updated—the device driver is unaffected.

In a multitasking operating system such as OS/2, the Advanced Basic Input/Output System (ABIOS)[1] provides the BIOS service in a way that is compatible with the more complex environment. In particular, the ABIOS code must be reentrant so that it may service more than one request at a time. This enables the overlapping of I/O operations. While the operating system is waiting for a relatively slow device to complete its task, other operations and other I/O can be initiated. As is the case with BIOS, the ABIOS provides the lowest layer of software insulation. It hides the specific hardware implementation behind a standard interface that can be accessed by higher levels of software. If the hardware is changed, only the ABIOS layer must be modified. The upper layers of software are not affected. In Figure 1B a device driver written to the ABIOS interface requires less code than that in Figure 1A, written to the SCSI interface.

The ABIOS is designed to operate in any multitasking, interrupt-driven environment. ABIOS is operating-system-independent. There is a price for this generality of implementation. The interface to ABIOS is complex and procedurally oriented. This is necessary in order to hide the details of the operating system from the ABIOS as well as to hide the details of the hardware implementation from the operating system. One of the goals of the split model for device drivers is to insulate the upper-level device drivers from the complexities involved with interfacing to the ABIOS. In order to accomplish this, an extra layer of software is inserted between the ABIOS and the operating system. This layer takes the form of an extra device driver that simplifies the interface to ABIOS and reduces the overall code size for SCSI device drivers. Figure 1C illustrates that less software is required for the OS/2 device driver when the SCSI driver software is inserted between the ABIOS interface and the OS/2 driver interface.

Figure 1 Code layering to reduce device driver development effort



SCSI architecture

The SCSI architecture[2] provides a standard device interface that greatly simplifies the operating system software that is required to drive the device. SCSI devices have the capability to respond to commands that are imbedded in data structures called control blocks. This means that the SCSI device driver's primary task is simply to translate an operating system request block into a control block that the device can understand. All that remains is to deliver the control block to the device, and then to transfer the data to or from system memory.

For SCSI devices, the ABIOS and the SCSI controller are essentially reduced to a delivery system. The SCSI ABIOS relays the control block to the SCSI controller which in turn passes it on to the device. This is a simplified view of the actual procedure in order to make the point that the standardized SCSI interface makes the job of controlling the device much easier. The SCSI controller does not need to know how the device works, it just needs to know how to send a control block to the device. It follows that the SCSI controller does not need to know the type of SCSI device it is controlling. It is enough to know that it is a SCSI device that conforms to the standard SCSI architecture. This forms the basis for the most attractive feature of SCSI, which is the ability to connect several different types of SCSI devices to the same bus and the same controller.

A bus master SCSI controller[3] requires slightly more information than just the SCSI control block. Since it takes full responsibility for moving any data to or from the device, the bus master controller needs to know the memory storage address of the data and the requested direction of data flow, as well as the amount of data to be moved. All of this information is placed in a super control block that contains the SCSI control block information in combination with the extra control information needed by the SCSI controller. From the software perspective, device control is still through a data structure, just a slightly larger one.

The split model

At this point it should be apparent that the SCSI environment is significantly different from other device protocols. Many different types of devices from different vendors can be controlled by one type of SCSI controller. Device control is achieved through a standard data structure interface. Any SCSI device is able to respond to the same standard SCSI command set. This means that it is possible for a single device driver to control different types of devices, as well as similar devices produced by different manufacturers.

The OS/2 environment provides additional features that can facilitate the full exploitation of the SCSI architecture. OS/2 initializes the ABIOS and provides system services to simplify device driver access to the services provided by ABIOS. OS/2 provides an interdevice driver communication facility (IDC)[4] that allows device drivers to communicate at run time. The IDC allows a single logical device driver to be split into two or more physical units.

The approach taken here is to split the SCSI device driver into two parts. The upper-level driver provides the direct connection to the operating system while the lower-level driver provides access to the SCSI devices through the SCSI ABIOS services. SCSI ABIOS is that portion of the ABIOS that is concerned with controlling the SCSI devices in the system. In Figure 1A–1C, respectively, the shaded portion represents the code that must be written for a new device. In Figure 1A, which shows no ABIOS, a significant amount of code must be written. With the introduction of ABIOS (Figure 1B), the lower interface is the ABIOS architecture, reducing the code. By splitting the device driver into a relatively fixed part (the SCSI driver) and the OS/2 device driver, as shown in Figure 1C, only the OS/2 device driver has to be rewritten. Refer to Figure 2 for a diagrammatic representation of the split device driver model.

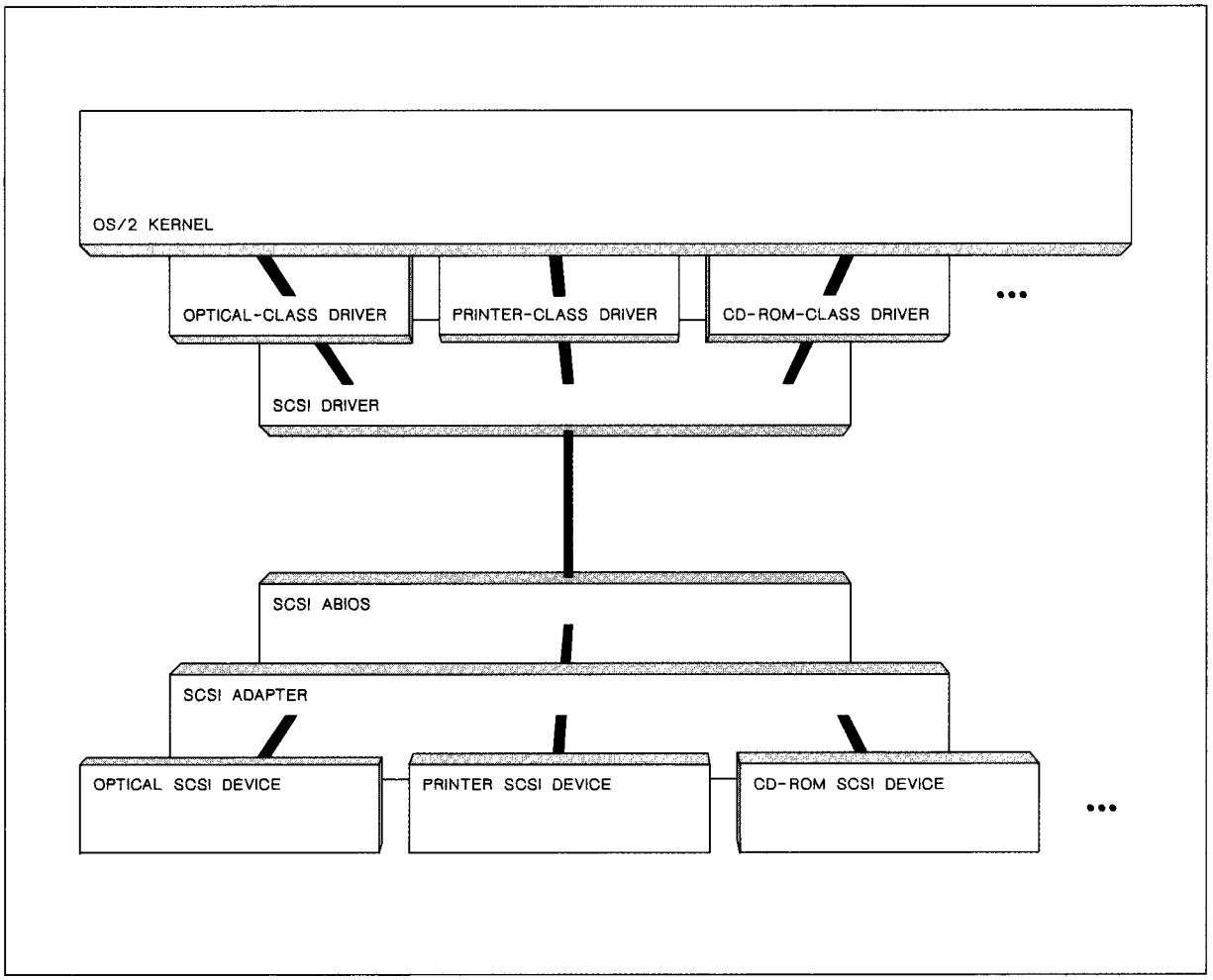
The extra interface provided by this model allows changes to be made to the upper-level device driver without affecting the lower-level device driver. In the extreme case, a completely new upper-level driver can be substituted for the current one. Taking this one step further, the lower-level driver can be designed to support more than one upper-level driver at the same time. This results in reduced development effort for new SCSI device drivers, once the lower-level driver has been written. Resident code size is significantly reduced since common code is isolated in the lower-level driver instead of being repeated in each upper-level driver.

The SCSI driver

The lower-level device driver is referred to as the SCSI driver because it is device-independent. It can be used to drive any SCSI device with the help of an upper-level driver. The SCSI driver is designed to include all of the functionality that would be common to any SCSI device driver. The duties of the SCSI driver can be summarized as follows:

- Queue all SCSI requests by device
- Provide the interface to SCSI ABIOS
- Field all SCSI interrupts
- Detect and handle time-out conditions

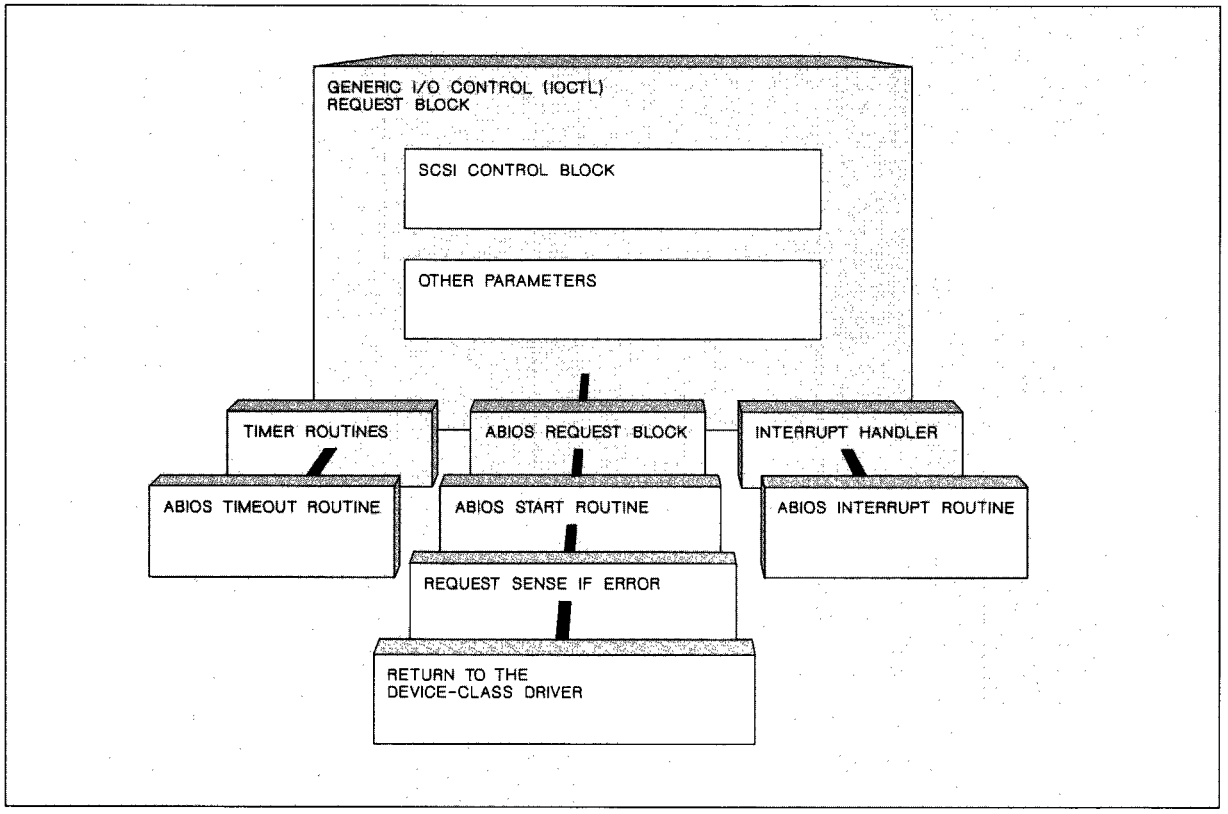
Figure 2 Split device driver model for OS/2



The queueing of pending device requests is managed by the SCSI driver for several reasons. Since all device drivers are required to queue requests, this is a common function that can be isolated in the SCSI driver instead of being repeated in each of the upper-level drivers. The relatively simple queueing services provided by the OS/2 kernel for use by device drivers are not used. Instead, the queueing algorithm has been optimized to work synergistically with the SCSI driver's interrupt handler in order to improve the overall throughput of requests. Maintenance of the relatively complex queueing code is simplified by the fact that there is only one copy of it.

The primary function of the SCSI driver is to provide the SCSI ABIOS services to the upper-level drivers through a simplified, declarative interface. All of the procedural details of calling ABIOS are hidden by the SCSI driver. Even the ABIOS request block data structure is hidden. The upper-level driver needs only to provide a few simple parameters to the SCSI driver in order to have a device request processed. The SCSI driver acts as a subroutine to the upper-level driver. When control is returned from the SCSI driver, the SCSI device request has been completed. Refer to Figure 3 for an overview of the functionality of the SCSI driver.

Figure 3 Control flow of the SCSI driver



Communication between the upper-level drivers and the SCSI driver is through the OS/2 architected interdevice driver communication (IDC) facility, as alluded to earlier. At initialization time, the SCSI driver registers its IDC entry point with the OS/2 kernel. Later, when an upper-level driver initializes, it obtains the SCSI driver's IDC entry point from the kernel. During task time, the SCSI driver services are invoked much like a simple subroutine call from the upper-level driver. However, at initialization time the IDC entry point cannot be used because it is located at protection level 0, whereas the initialization code takes place at protection level 3. The only method of invoking the SCSI driver at initialization time is to make an OS/2 dynamic link function call. The DosDevIoctl call serves as the entry into the SCSI driver during the upper-level driver's initialization. This works out well since the DosDevIoctl call uses the OS/2 generic ioctl (I/O control) interface, which is general and flexible.

In order to keep things simple, the generic ioctl interface is also used at task time. This means that there is only one entry point into the SCSI driver. The strategy entry point and the IDC entry point are the same. At task time, the upper-level driver follows approximately the same procedure that the kernel follows to invoke the SCSI driver. The upper-level driver constructs a generic ioctl request packet, points a hardware register pair to it, sets up the context for the SCSI driver, and then calls the SCSI driver's IDC entry point.

The generic ioctl request packet contains two routing parameters so that the request can be efficiently directed to the proper service routine. The primary parameter is the function category. The SCSI driver uses the first user-definable category, 128, to provide the generic SCSI ABIOS services. The categories from 129 through 255 are available for the possible future expansion of SCSI driver services. The secondary routing parameter

is the function code. The generic ioctl function code that is used is a linear mapping of the generic SCSI ABIOS function number. The zero-based ABIOS functions are converted to 64-based generic ioctl function codes in order to follow the generic ioctl conventions. That is, 64 is added to each ABIOS function code in order to get the corresponding generic ioctl function code that will be used to invoke the SCSI ABIOS service.

Once the request has been routed, the service routine uses two other generic ioctl request packet parameters in order to satisfy the request. The parameter buffer pointer points to a data structure that contains the parameters that the SCSI driver will use to invoke the generic SCSI ABIOS services. The primary parameter used here is a pointer to the SCSI control block that was constructed by the upper-level driver. These parameters vary according to the generic SCSI ABIOS service being requested.

The generic ioctl request packet also contains a pointer to a data buffer. This is set to point to the area that the upper-level driver would like to use for sense data. The SCSI architecture provides a well-defined error reporting and recovery protocol. The most common SCSI error that the device driver encounters is called a check condition. When a check condition is reported, the device driver is expected to issue a request sense command to the SCSI device. The device then returns detailed error information in a sense data structure. When a check condition is detected, the SCSI driver automatically generates a request sense command. The resulting sense data structure is returned to the upper-level driver in the data buffer that was provided as part of the generic ioctl request packet.

The ABIOS architecture provides three entry points for use by device drivers. They are the START, INTERRUPT, and TIMEOUT routines. A somewhat complex set of procedures must be followed according to the return code from each of the three ABIOS routines. The most common scenario is for the device driver to first call the START entry point in order to initiate a device request. The START routine usually returns as staged-on-interrupt. At this point the task time thread of the device driver suspends its execution by blocking or returning to the operating system. When the device completes the operation, it generates an interrupt that is fielded by the device driver's interrupt routine. The interrupt routine then calls the ABIOS INTERRUPT entry in order to release the interrupt at the device level. The device driver interrupt routine is also responsible for finishing up the request by running the blocked thread or by doing the remaining processing itself.

This is a simplified version of the procedural details of using ABIOS that are hidden by the SCSI driver. Many other scenarios are possible, and all must be anticipated. In addition, each request must be timed to be sure that it completes in a reasonable period. If not, the ABIOS TIMEOUT routine must be called. To further complicate matters, it is possible for the TIMEOUT routine to time out.

It should be apparent that the common functionality provided by the SCSI driver is also the difficult part of an OS/2 device driver. All of the duties associated with a multitasking environment are performed by the SCSI driver. The request queueing, request timing, and interrupt handling code must be carefully coordinated with the ABIOS code in order for a device driver to be efficient and still be somewhat hardware-independent. Sequestering these functions in a separate driver is a natural and logical enhancement to device driver development in an OS/2 SCSI environment.

The device-class driver

The upper-level device driver is referred to as a device-class driver because it generally drives devices that belong to a specific SCSI classification. Printer, CD-ROM, and read/write optical are examples of SCSI device types that require a different upper-level device driver to work in concert with the lower-level SCSI driver. The primary functions of the device-class driver can be summarized as follows:

- Present a logical view of the device to the operating system
- Translate an OS/2 request block to a SCSI control block
- Provide multiple vendor support by handling vendor-unique features of the device

OS/2 device drivers are separated into two classifications. Block device drivers control random-access mass storage devices such as fixed disk or diskette. Character device drivers are used for sequential access devices such as printers. The

first job of the device-class driver is to classify the device as being either character or block. Within these major groupings, smaller subgroupings are necessary due to other differences in the nature of the devices. For instance, CD-ROM and read/write optical devices are similar in many ways and yet different enough that they require separate classification. The medium is optical for both devices, but CD-ROM is read-only, has a larger block size, and supports a large number of audio commands that are not used with a read/write optical device.

Within a SCSI device class, it is possible for one device driver to support many different devices. Because of the standard SCSI command set, similar devices produced by different manufacturers can be controlled by the same device-class driver. The slight behavioral variation from one vendor to another can easily be handled by the device-class driver because the SCSI architecture provides a standard method of identifying the device.

The device-class driver is also capable of handling variations in media size and configuration. Read/write optical devices provide a good example of this ability. These devices come in basically two major types, those that use 3.5-inch media and those that use 5.25-inch media. Within media size groupings, different media densities are available. One optical device-class driver can provide support easily and without a great deal of extra code for the different media sizes, the different media densities, and the different vendors. Since the primary SCSI command set supported is approximately equivalent for the different devices and media types, the device driver's major task is just to let the operating system know about the different media configurations that are being supported.

The actual mechanics of controlling a SCSI device are almost entirely provided by the layers of code beneath the device-class driver. The SCSI controller, the generic SCSI ABIOS, and the SCSI driver act together to reduce the function of the device-class driver basically to that of a translator. The device-class driver converts an OS/2 request block into a request block that the SCSI controller and the SCSI device can understand, and then passes it on to the SCSI driver. Refer to Figure 4 for a simplified view of the device-class driver.

Since the device-class driver is so simple, new SCSI device driver development can proceed very quickly. Development is further facilitated by the fact that the device-class drivers are similar in many ways. Once a prototype is available, new drivers can be derived from it. In some cases, a new device can be supported by simply adding it to an existing device-class driver.

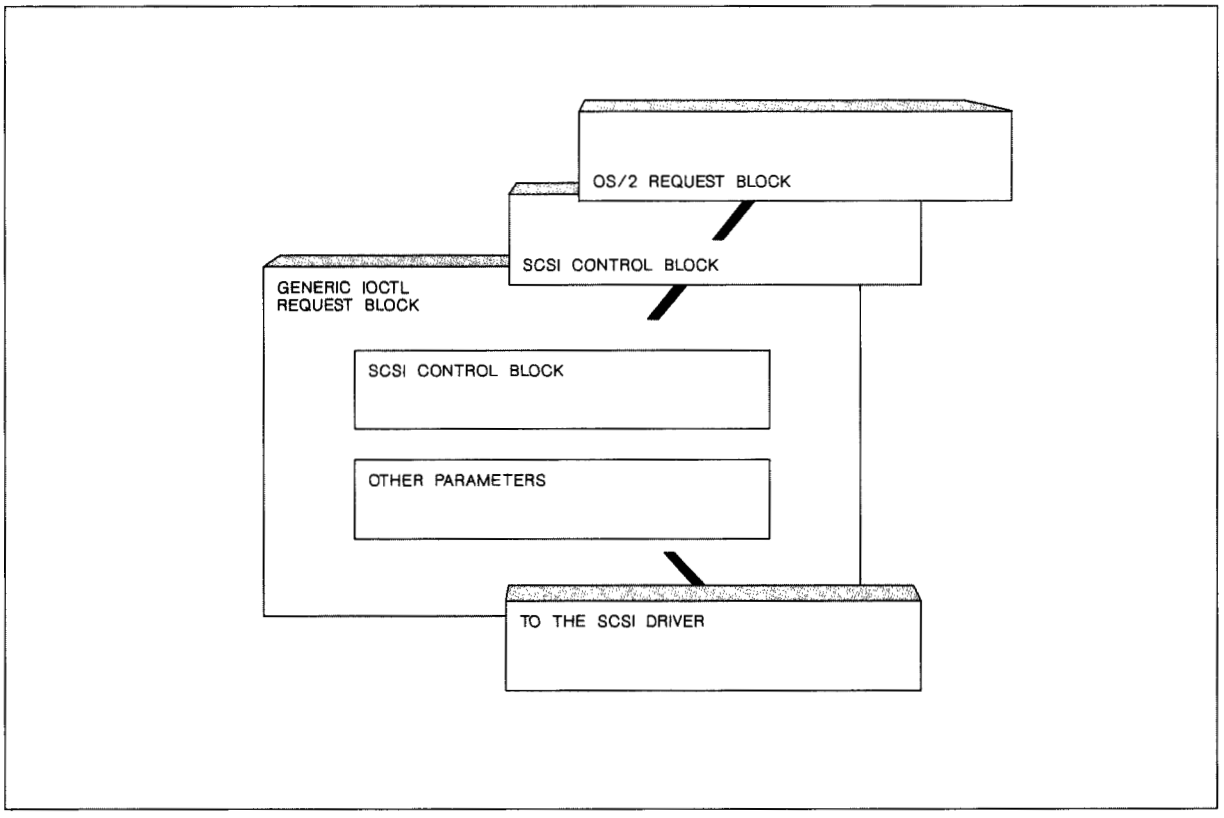
Advantages of the split model

The split device-driver model for OS/2 SCSI device drivers provides many advantages over the traditional model for OS/2 device drivers. The primary motivation in the design of the split model is to reduce the SCSI device driver development effort for current and future devices. Results in this area have been very good. New SCSI device drivers have been developed in short periods of time. In one case, a read/write optical device driver for a 3.5-inch media device was converted to also support 5.25-inch media devices. Support was added for four different vendor devices, each using a different type of medium with a different capacity. In addition, one of the devices supported two different media densities. The resulting single device-class driver supported five different read/write optical devices. Subsequently, this device driver was again modified to include support for two more devices, for a total of seven devices and eight different types of media.

The development effort for a completely new SCSI device driver within the split model architecture is less than half of the effort that would be required for a traditional device driver. This is because approximately half of the device driver code is contained in the common SCSI driver that is used by all device-class drivers. If a sample device-class driver is available, the remaining development effort will again be reduced by about half, assuming that the code is modular, and that it was developed in the C programming language.[5] If so, at least half of the existing device-class driver code will probably be reusable.

Additional benefits of the split model include the reduction in size and enhanced performance. Both factors are improved in direct proportion to the number of different types of SCSI devices that are attached to a system. Each additional device-class driver represents a size savings equivalent to approximately the size of the SCSI driver. Execution performance is enhanced by the fact that there is only one interrupt handler registered for all of the devices controlled by the SCSI driver. Since the operating system must chain through

Figure 4 Control flow of the device-class driver



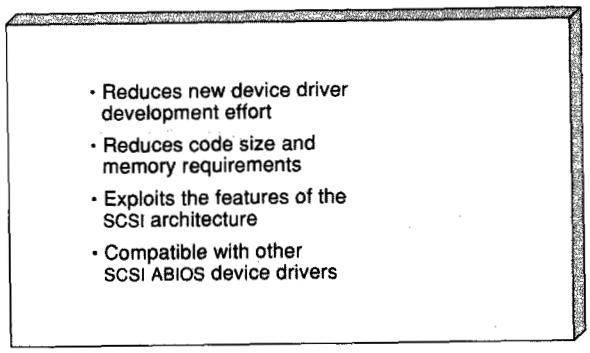
the interrupt handlers that are registered for a particular interrupt level, the performance benefit is again directly related to the number of different types of SCSI devices in the system, and therefore the potential number of different interrupt handlers. Utilizing only one interrupt handler seems to follow logically from the standardized nature of the SCSI architecture. The single SCSI controller type implies that the minimal interrupt service routine is equivalent for all of the devices that are connected to the SCSI driver.

The full compatibility of the SCSI driver with other SCSI ABIOS device drivers is a critical and important feature. The SCSI driver does not directly access the SCSI controller. The ABIOS is used exclusively for access to the SCSI devices. The SCSI driver does not claim any devices for use at its initialization time. Devices are claimed by the device-class driver during its initialization. Only those devices that the driver will actually control are claimed. This means that SCSI ABIOS device drivers that are not part of the split-model architecture can be loaded either before or after the split-model drivers without having any conflicts occur. Figure 5 summarizes the benefits of the split model for OS/2 SCSI device drivers.

Summary

The split model exploits the SCSI architecture to improve upon the basic design of an OS/2 device driver. Already existing features of OS/2 and the PS/2 are combined in a new way in order to create an optimized device driver model for SCSI devices. The generic ioctl interface and the inter-device driver communication facility are combined to allow a device driver to be split into two parts, without violating the OS/2 architecture. The generic SCSI ABIOS provides a relatively hardware-independent layer of code that is used to access the SCSI controller. Since the controller is

Figure 5 Advantages of the split model



a bus master, it provides extensive data movement and control services that greatly simplify the requirements of the device driver. The standardized SCSI command set provided by the SCSI architecture makes it easier to be able to modify existing device drivers to add support for new SCSI devices as they become available.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references

1. *IBM Personal System/2 and BIOS Interface Technical Reference*, IBM Corporation (1988); available through IBM branch offices.
2. *Small Computer System Interface* (SCSI), X3.131-1986, American National Standards Institute, New York (1986).
3. *IBM Personal System/2 Micro Channel SCSI Adapter Technical Reference*, IBM Corporation (1990); available through IBM branch offices.
4. *IBM Operating System/2 Programming Tools and Information Version 1.2, Device Drivers Volume 1*, IBM Corporation (1989); available through IBM branch offices.
5. Dan Feriozi, "A C Programming Model for OS/2 Device Drivers," *IBM Systems Journal* **30**, No. 3, 322–335 (1991).

General reference

A. M. Mizell, "Understanding Device Drivers in Operating System/2," *IBM Systems Journal* **27**, No. 2, 170–184 (1988).

Accepted for publication September 26, 1991.

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. G321-5465.