A FLAT MODEL BASE DEVICE DRIVER FOR OS/2™

June 1996

Unionville, Connecticut

Abstract of Project

The recent interest in Network Computers (NC) has kindled somewhat of a rebirth in as a platform for an Intel-based NC. With the bloated Workplace Shell and other Graphical User Interface (GUI) components removed, the OS/2 kernel makes a good foundation for a Java-based workstation. It requires the installation of a Java Virtual Machine (JVM) and the required operating system support for the Java Application Programming Interface (API). Once enabled, the system will allow for the execution of Java applets and Java applications using the underlying OS/2 kernel for memory management, scheduling, dispatching, and device support.

In order to take advantage of the new crop of device architectures now emerging, such as Universal Serial Bus (USB), Firewire, and others, the OS/2 device driver interface must be redesigned to support both the older 16-bit device drivers and a new crop of 32-bit device drivers[1]. In order to insure continued support, the drivers should be written with currently

---

available and supported 32-bit tools such as Visual Age C++ for OS/2 and the Microsoft 32-bit macro assembler.

To allow OS/2 to be used in a network configuration, perhaps in a diskless mode, the system must be able to dynamically load device drivers as needed from a local disk or from a server. This is a major change in the way OS/2 operates today. Currently, all device drivers are loaded at boot time, and are specified in the special CONFIG.SYS file. Adding or deleting a driver from the system requires that the CONFIG.SYS file is edited and the system rebooted to effect the change. Therefore, we regard the dynamic device driver-loading feature to be an integral part of our requirements and necessary for the successful implementation of our design.

Trademarks and Copyrights

Table of Contents

List of Figures

Listings

Chapter 1. - Introduction

Problem

OS/2 device driver development is at a standstill. The OS/2 device driver model was derived from a DOS device driver model. It retains much of the 16-bit architecture of the 16-bit Intel 80286 processor on which was first implemented in OS/2 1.0. Since that time, the mainstream Personal Computer (PC) processor has grown from a 16-bit architecture to a 32-bit architecture. Since the newer processors are backward compatible, the older driver software still runs without change. However, 16-bit compilers for OS/2 are no longer available, stunting the growth of new driver development. The mixed mode of 16-bit and 32-bit interfaces makes driver development difficult, and programmers spend too much time creating specialized code to handle the address conversions.

Purpose

The purpose of this study is to identify the amount of work necessary to convert existing 16-bit device drivers to 32-bit device drivers, and to provide a framework for new device driver development using 32-bit tools. It should be noted that this proposal describes the changes necessary to convert existing 16-bit device drivers to 32-bit, and to allow new 32-bit drivers to co-exist with the older model. While it would have been easy to design a new model based on current technology such as the I2O initiative, the goal of this proposal was to preserve the investment in current device driver development and training.

Importance of Study

This study will help determine if converting existing 16-bit device drivers to 32-bit is feasible, and if the return on programming investment is worth the effort. The data gathered, along with the conclusions, will determine the level of funding and programming resources to allocate.

Scope of Study

This proposal suggests a new device driver model for the OS/2 operating system, and as such, contains several design suggestions and criteria that are specific to OS/2. OS/2 is a hybrid system, part 16-bit and part 32-bit. Many of the functions or methods used or presented in this design are specific to the OS/2 operating system, and therefore not necessarily valid or applicable for other operating systems such as Windows NT or Unix.

Rationale of Study

Since the OS/2 device driver development environment can be problematic, we decided that the best way to verify our design was to actually implement it. Many design flaws or omissions occur only under load, in actual operating conditions, especially those issues related to timing, race conditions, or interrupt latency. By using real working device drivers, we felt that we could expose these problems more quickly than any other method.

Definition of Terms

API. Application Programming Interface, a definition of what external functions are available to a program and how to call them.

Arbitrate. In programming terms, the ability of the device driver to act as a "traffic cop" to grant or deny hardware access to an application.

Abstract (verb). In programming terms, the verb abstract means to "hide the details". A device driver abstracts or "hides the details" of the low-level device architecture from the application software.

Device Driver. The software program that acts as the interface between an application program and a hardware device. The device driver is responsible for converting high-level requests from the application program into low-level commands that the hardware can understand.

DMA. Direct Memory Access, a method of transferring data to and from memory by using a specialized piece of hardware.

Dynamic Linking. An architecture that allows external references to be resolved at program load time. This results in smaller executables but a slightly longer initial load time.

File System. The internal subsystem in an operating system that controls access to hardware devices.

Flat Model. A term that describes a 32-bit addressing mode in which the address contains the actual memory page information and offset. The processor hardware decodes the physical page information. Flat pointers can directly access up to 4GB of memory.

Hardware. The physical electrical components of the computer system or device interface.

Legacy Devices. A term used to denote older devices and their support hardware. These devices are usually ISA bus devices.

Multiprocessor. A computer architecture where the computer is equipped with more than one Central Processing Unit (CPU).

Polling. A programming method that involves waiting in a tight loop for a particular bit to change state or a particular event to occur. Polling results in inefficient use of the processor cycles because other programs are prevented from running while polling is being performed.

Protect Mode. A specific mode of operation for the Intel 80x86 series of processors that provides for hardware-based memory protection and a 32-bit flat memory architecture.

Real Mode. A specific mode of operation for the Intel 80x86 series of processors that provides a one-megabyte memory size with no memory protection. MS-DOS runs in real mode.

Segment:Offset. A method of memory addressing for the Intel 80x86 series of processors that forms a 32-bit address from a 16-bit selector and a 16-bit offset. The selector is actually an index into a table of memory descriptors that contain the physical page number and access rights of the memory. The offset portion is only large enough to describe a 64K segment of memory.

Spinlock. A specialized program loop that prevents other operations or programs in the system for being executed while the special loop is being executed.

Thunking. The process of converting 32-bit pointers to their 16-bit equivalents, and 16-bit pointers to their 32-bit equivalents. Thunking from 32 to 16 can be very time-consuming.

Overview of the Study

Early Personal Computer (PC) operating systems such as MS-DOS were single tasking, i.e.; they were capable of executing only one program at a time. Even though these systems were somewhat slow, they were still much faster than the devices they needed to access. Most output information was sent to a line printer and most input data was read from a keyboard. If a program needed to perform input or output (I/O) to one of these devices, the system would effectively remain idle while waiting for the data to be sent or entered. This method of performing I/O, called **polling**, was very inefficient. Even though the computer was capable of executing thousands of instructions in between each keystroke, it was kept busy while I/O was being performed. If a program needed to print something on a printer, it would send the data one character at a time, waiting for the device to acknowledge that the character was accepted before sending the next character. Since the computer processed the data faster than it could be printed, it would sit idle for much of the time waiting for the printer to do its job. As technology progressed, faster I/O devices became available, but so did faster computers. The computer was at the mercy of the input and output devices it needed to access.

The problem was exacerbated by the fact that each one of these I/O devices required data in a different form. Some devices required data in a serial fashion, bit by bit, while others required data 8 or 16 bits at a time. Some line printers printed on 8 1/2 by 11-inch paper and some on 11 by 14-inch paper. Magnetic tape storage devices used different size tapes and formats, and disk storage devices differed in both the amount and the method of storage.

The **device driver** solved the problems associated with the different types of devices, and provided a more efficient method of utilizing the processing power of the computer while it was performing input and output operations. The device driver is a small program inserted between the program performing the I/O and the actual **hardware** device such as a printer or magnetic disk drive.

The device driver is programmed with the physical characteristics of the device, and acts as an interface between the program and the device. For example, the device driver for a line printer might be programmed with the characteristics of the printer, such as the number of characters per line or the size of the paper that the device supports. The device driver for a magnetic tape drive might be programmed with the physical characteristics of the tape mechanism, such as the tape format and density. The ability of the device driver to hide or *abstract* the details of a hardware device is a fundamental concept in the design of today's operating systems and applications software. Programmers writing software do not need detailed knowledge of the devices that the program may use, and do not have to provide specialized programs to access those devices. For example, the software for an application that needs to send data to a printer can be written the same way no matter what type of printer is installed on the system. The same data can be sent to a high-resolution laser printer or a low-resolution dot-matrix printer with the same results. Old printers can be deleted, or new types of printers added without the need to change the application software.

Device drivers also address the problem of polling. Since the device driver has intimate knowledge of how to deal with the hardware, there is no reason why the application program has to wait around for each character to be printed. It can, for example, send the device driver a block of 256 characters and return to continue executing the application program while the device driver handles the details of sending the data to the device. When the device driver finishes sending all the data, it notifies the application program that it needs more data. The application then sends the next block of data to the device driver for processing. While the device driver is performing the I/O, the application can continue processing data or performing other operations. This results in more efficient use of the system processor.

The use of device drivers became even more important when operating systems such as Windows and OS/2 appeared that could run more than one program simultaneously. In systems that can run many programs at the same time, it is possible that more than one program might try to access a hardware device at the same time. The device driver performs an important function by controlling access to the hardware device. In some cases, such as a printer, the device driver serializes or **arbitrates** access to the device so that one program's output does not appear in the middle of another program's output. Once the printer has begun servicing a program, subsequent attempts to access the printer are refused, and the "busy" indication is sent back to the requesting program. The requesting program can then decide to wait for the printer to become available or to access the printer at a later time.

Device drivers are an irreplaceable and critical link between the operating system and the I/O device (see Figure 1-1). They can interact directly with the Central Processing Unit (CPU) and operating system, and in some cases, can allow or block the execution of programs. Device drivers usually operate at the most trusted level of system integrity, so the device driver writer must test the driver code thoroughly to assure bug-free operation. Failures at a device driver level can be fatal causing the system to crash or experience a complete loss of data.

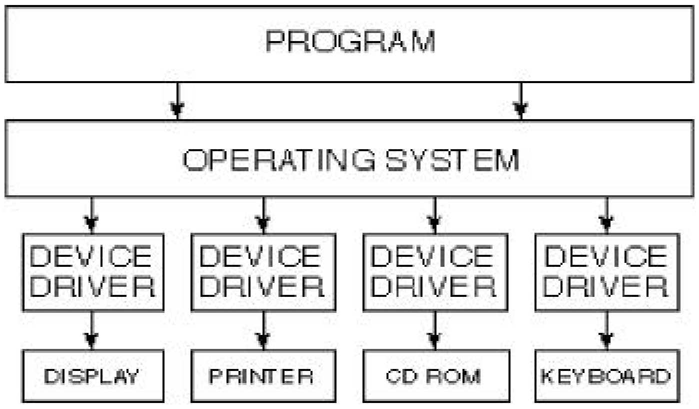

Figure 1-1. The Role of the device driver.

The use of computers for graphics processing has become widespread. It would be impossible to support the many types of graphics devices without device drivers. Today’s hardware offers dozens of different resolutions and sizes. For instance, color graphics terminals can be had in pixel sizes of 800 by 600, 1024 by 768, 1280 by 1024, and as high as 2048 by 2048. Each resolution can support a different number of bits per pixel, or color depth.

Printers vary in dots per inch (DPI), font selection, and interface type. Since all of these formats and configurations are in use, the supplier of a graphics design package that needs to send data to a printer would have to support all of the configurations in order to offer a marketable software package. A graphics application program might direct the output device to print a line of text in Helvetica bold italic beginning at column 3, line 2. Each graphics output device, however, might use a different command to print the line at column 3, line 2, so the device driver resolves these types of differences. Instead of having to write, debug, and support all of these special device drivers, the graphics application reads from and writes to these graphics devices using a standard set of APIs, which in turn call the device driver specific to the device currently installed. Without this standardized interface, the software vendor would be forced to supply device drivers for the hundreds of different types of input and output devices. Some word processors, for example, would be forced to supply hundreds of printer device drivers to support all makes and models of printers, from daisy wheel to high-speed laser and color printers.

In summary, the device driver:

- Contains the specific device characteristics and removes any responsibility of the application program for having knowledge of the particular device.

- Allows for device independence by providing for a common program interface, allowing the application program to read from or write to generic devices. It also handles the necessary translation or conversion that may be required by the specific device.

- Serializes access to the device, preventing other programs from corrupting input or output data by attempting to access the device at the same time.

- Protects the operating system and the devices owned by the operating system from errant programs which may try to write to them, causing the system to crash.

OS/2 Overview

OS/2, introduced in late 1987, was originally called MS-DOS 4.0. It was later named MS-DOS 5.0, and finally OS/2. OS/2 was designed to break the MS-DOS 640KB **real mode** memory barrier by utilizing the **protect mode** of the 80286 processor. The protect mode provided direct addressing of up to 16 megabytes (MB) of memory and a protected environment where badly written programs could not affect the integrity of other programs or the operating system.

The Intel processors are capable of operating in one of two modes. These are called real mode and protect mode. One of the most popular computer operating systems, MS-DOS, runs in real mode. In real mode, the processor is capable of addressing up to one megabyte of physical memory. This is due to the addressing structure, which allows for a 20-bit address in the form of a segment and offset (see Figure 1-2).

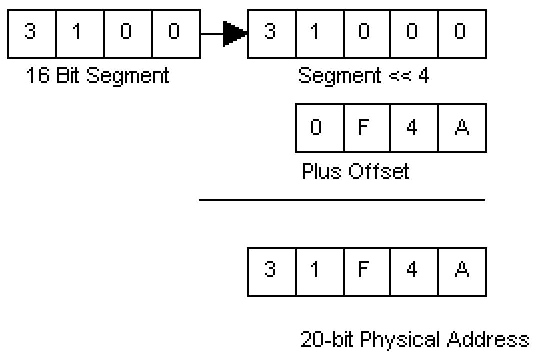

Figure 1-2. Real mode address calculation.

Real mode allows a program to access any location within the one-megabyte address space. There are no protection mechanisms to prevent programs from accidentally (or purposely) writing into another program's memory area. There is also no protection from a program writing directly to a device, say the disk, and causing data loss or corruption. MS-DOS applications that fail generally hang the system and call for a <ctrl-alt-del> reboot, or in some cases, a power-off and a power-on reboot (POR). The real mode environment is also ripe for viruses or other types of sabotage programs to run freely. Since no protection mechanisms are in place, these types of "Trojan horses" are free to infect programs and data with ease.

The protect mode of the Intel 80286 processor permits direct addressing of memory up to 16MB, while the Intel 80386 and 80486 processors support the direct addressing of up to four gigabytes (4,000,000,000 bytes). The 80286 processor uses a 16-bit selector and 16-bit offset to address memory (see Figure 1-3). A selector is an index into a table that holds the actual address of the memory location.

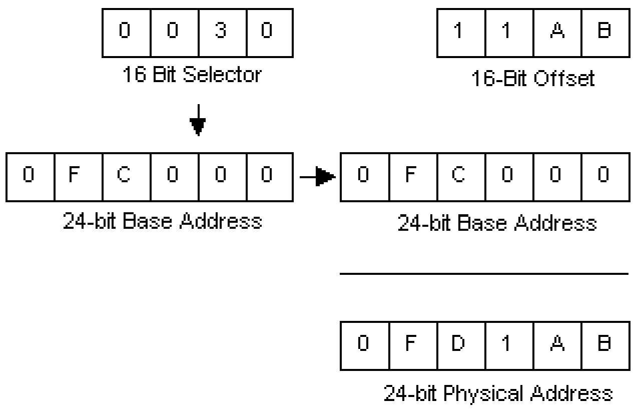

Figure 1-3. 80286 protect mode addressing.

The offset portion is the same as the offset in real mode addressing. This mode of addressing is commonly referred to as the 16:16 addressing. Under OS/2, the 80386 and 80486 processors address memory using a **linear address**. The linear address is a 32-bit flat address consisting of three parts. The first part, which is 10 bits long, is an index into a page table referred to as the PTE. The second part, which is also 10 bits long, specifies a particular page frame within the page table. The third part is an offset into the page frame. The page information is decoded by the paging hardware located on the processor. The physical address is formed by locating the PTE, indexing to the correct page frame, and adding in the offset portion of the address. This mode of addressing is referred to as the 0:32 or flat addressing (see Figure 1-4).

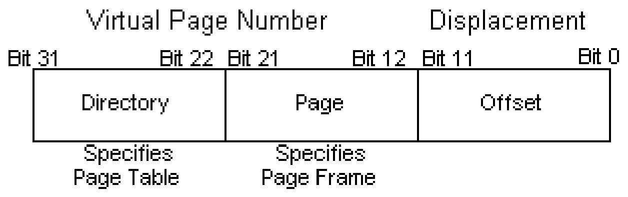

Figure 1-4. 80386 linear addressing.

The protect mode provides for hardware memory protection, prohibiting a program from accessing memory owned by another program. While a defective program in real mode can bring down the entire system (a problem frequently encountered by systems running MS-DOS). A protect mode program that fails in a multitasking operating system merely reports the error and is terminated. Other programs running at the time continue to run uninterrupted.

To accomplish this memory protection, the processor keeps a list of memory belonging to a program in the program’s Local Descriptor Table, or LDT. When a program attempts to access a memory address, the processor hardware verifies that the address of the memory is within the memory bounds defined by the program’s LDT. If it is not, the processor generates an exception and the program is terminated.

The processor also keeps a second list of memory called the Global Descriptor Table, or GDT. The GDT usually contains a list of the memory owned by the operating system, and is only accessible by the operating system and device drivers. Application programs have no direct access to the GDT except through a device driver.

OS/2 was designed with several goals in mind. First, OS/2 had to provide a graphical user interface that was consistent across applications and graphics hardware.

Second, OS/2 had to support **Dynamic Linking**. With Dynamic Linking, some functions required by an application can reside in a separate file, which is loaded only at run time. This feature makes application file sizes smaller, allows more than one client to use the Dynamic Link Library (DLL), and allows functionality to be placed in the DLL without changing the application code base. The majority of OS/2 is implemented in DLLs.

Third, OS/2 had to provide an efficient, preemptive multitasking kernel. The kernel had to run several programs at once, yet provide an environment where critical programs could get access to the CPU when necessary. OS/2 uses a priority-based preemptive scheduler. The preemptive nature of the OS/2 scheduler allows it to “take away” the CPU from a currently running application and assign it to another application.

OS/2’s smallest granularity of execution is the **thread**, which is an instance of execution. Processes consist of one or more threads, and each thread can execute at its own priority. OS/2 has four priority classes with 32 levels within each priority class. Threads with higher priority can interrupt the execution of lower priority threads.

Fourth, OS/2 had to provide a robust, protected environment with virtual memory. OS/2 uses the protect-mode of the 80286 and above processors, which has built-in hardware memory protection. Applications that attempt to read or to write from memory that is not in their specific address space are terminated without compromising the operating system integrity. OS/2 uses an efficient memory allocation and paging scheme consisting of a combination of first-fit, Least Recently Used (LRU), and compaction to minimize fragmentation.

Fifth, OS/2 had to support the older 16-bit protect-mode applications that used the **segment:offset** addressing scheme as well as new, 32-bit **flat model** executables. OS/2 offers a rich set of Application Program Interfaces (APIs) to allow programs to access system services. The OS/2 APIs are classified into eight major categories; file systems, graphics interface, inter-process communications, systems services, process/thread management, memory management, signals, and dynamic linking.

Finally, OS/2 had to run most MS-DOS programs in an MS-DOS-compatibility mode. OS/2 allows MS-DOS programs to run in their own one megabyte of virtual memory space, providing protection from other MS-DOS or OS/2 programs.

At the time OS/2 was written, the most powerful mainstream processor available for the PC market was the Intel 80286. At that time, memory was still quite expensive, and most PC systems were shipped with a maximum of 4MB of memory installed. Accordingly, the operating system and support code was written using 16-bit tools that produced smaller and faster code. Because of space and speed considerations, a majority of the OS/2 kernel and supporting DLLs were written in assembly language.

After Microsoft and IBM split on the direction for OS/2, IBM embarked on a project to convert 16-bit OS/2 into a 32-bit operating system. This was now possible for two reasons. First, the mainstream processor shipped with Intel-based PCs was the 80386, a full 32-bit processor. Second, the price of memory had dropped dramatically, and most systems were now shipped with up to 8MB of memory[2].

During this conversion, IBM elected to make a radical change in the user interface rather than to rewrite the OS/2 kernel. This was done for two reasons. The first was that like most companies, they had limited programming resources that could be brought to bear on the rewrite. The second reason was that OS/2 device drivers were difficult to write, and IBM wanted to retain support for any existing 16-bit device drivers that had already been written. All of the system's internal data structures were byte or word aligned, the scheduling, dispatching, and virtual memory data structures were all written in assembly language, and accessed via 16-bit selector:offset addresses. The **file system**, the heart of the device driver interface, was also written entirely in 16-bit code. IBM made the decision to work on the area of OS/2 that would make the most visual impact while leaving all of the internal plumbing untouched. This turned out to be a serious long-term error in judgement.

For tools, IBM used the Microsoft 16-bit assembler and Microsoft 16-bit C compiler. For the few 32-bit components that were part of OS/2, IBM used an unreleased 32-bit version of Microsoft's C compiler. Worse, some components would compile and link only with certain versions of the assembler or compiler. Microsoft, seizing a golden opportunity, discontinued the availability of the 16-bit tools and never released the 32-bit version of their 32-bit C compiler. This brought a grinding halt to mainstream device driver development, relegating it instead to a few independent driver development shops that still had a license to the old tools. Today, almost seven years later, OS/2 still suffers from those fateful decisions to leave the file system and device driver interface untouched.

The Proposal

This proposal suggests a new flat-model base device driver model written in C. Appendix A also suggests a new base driver model written in C++ utilizing Object Oriented (OO) design techniques[3]. OO purists may find some of the C++ code and techniques objectionable because they don't adhere to a strict OO paradigm. However, it is our observation that interaction directly with the hardware requires a somewhat procedural approach, even if an object oriented language is used. Using C++ presents some possible problems, such as the timely collection of dead objects from the system heap. In OS/2, the system heap is swappable, which may lead to several page faults when destructors are called or during system heap compaction.

We have chosen C because of its universal acceptance and its ability to easily perform low-level functions. In Appendix A, we present a sample driver written in C++. Although some C++ programmers may find this model more “comfortable”, we see no advantage in using C++ over conventional C. The allocation of objects in a device driver is usually done at load time, and the resources remain locked while the device driver is loaded in memory. If, for example, the driver needed to access a buffer at interrupt time, it would not be possible to allocate the buffer or swap it in off the disk in time to service the interrupt, so the buffer must remain in memory. Also, C++ calls object destructors automatically, which might result in an object going out of scope at just the wrong time. This could cause intermittent or fatal errors, and would be difficult to track down. A feature of C++ is the ability to return to the heap the space that was allocated by objects that have since gone out of scope. The ability to perform timely garbage collection and heap compaction can be critical for the proper operation of a C++ device driver. Because there are some issues related to C++ that we are not sure can be easily solved, we decided to use C for the purposes of this design.

Our proposal intends to solve the following problems:

1. Currently, device drivers are loaded in the order that they appear in the CONFIG.SYS file. This can cause conflicts in certain hardware configurations where port or memory addresses may overlap.
2. In the current implementation of OS/2, no file I/O services are available from within the driver[4]. There is no method of logging activity or events during driver execution.
3. There are no profiling services available at the driver level.
4. There are virtually no supported 16-bit compilers and assemblers with which to build new device drivers.
5. Device drivers are currently unable to allocate more than 64KB of contiguous physical memory.
6. To use DMA, the driver must write directly into the 8237 DMA controller registers, perform several memory allocations until a memory object is allocated on the correct segment boundary and paragraph alignment, and double-copy data if the application buffer is above the 16MB boundary[5].

7. Device drivers are loaded statically by references in the CONFIG.SYS file. The drivers must be able to be loaded dynamically or “as needed”.
8. Drivers are loaded in low memory, below the 640KB boundary limiting the size and number of device drivers that can be loaded at boot time.

To satisfy these requirements, we have made the following assumptions:

1. The file system will be rewritten to handle the proper device driver calls. This might include a 32-bit to 16-bit thunk layer or memory aliasing to provide a seamless interface to the 32-bit device driver. In other words, the 32-bit device driver should be able to be installed, configured, run, and deinstalled without being aware if the underlying architecture is 16-bit or 32-bit.
2. The file system will be modified to accept long device names. Currently, OS/2 supports only 8 character device names, a relic from the old 8.3 file naming conventions.
3. The system will be modified to support a persistent data store or registry for supported devices, either installed or uninstalled.

4. The system will correctly handle mapping or aliasing pointers passed by 16-bit applications inside the request packet. It is the programmer's responsibility to handle any imbedded pointers in private data buffers shared between the application and the device driver.
5. The system will be modified to add the DeInstall command that removes the driver from the system, freeing up any resources owned by the driver[6].
6. OS/2 will provide a configuration utility to examine and modify the persistent configuration store or registry to permit legacy device settings to be stored.
7. OS/2 will be modified to allow access to the least significant bits of the 8254 timer to provide a reasonable granularity for the driver profiling services.
8. The OS/2 system loader will be modified to load the new device driver model. This loader should operate both at boot time and later when a device driver is dynamically loaded. We think this effort will be a major task.

9. Wherever possible, the new driver model shall incorporate code, functions, subroutines, and macros from the existing 16-bit device driver model to ease the conversion of older drivers. The conversion of older drivers should be easy and straightforward, and if possible, should follow the architecture of the existing 16-bit model[7].

Chapter 2. - Review of Related Literature

Requirements

During boot, the system will enumerate the system hardware to determine the current configuration. The current configuration will be compared to the last good configuration from the persistent store. If the information is different, the appropriate event mechanism will be started[8]. If the boot process is successful, the old configuration information should be backed up to a file, and the new configuration should become the current configuration. The system should always keep a known good configuration in the event of a system crash or corruption of the configuration persistent store. If a corrupt configuration file is encountered during boot, the user should be given the choice of using the last known good configuration or loading a new configuration from a file on floppy disk or hard disk. This operation is already performed by many popular operating systems such as Windows NT.

The device driver model will support the ability to be dynamically loaded, configured, and removed, and the system should automatically scan the machine to determine if any devices have been added or removed (Shanley, 1995). The system should attempt to detect legacy devices as well as enumerating Plug and Play devices, PCI devices, and devices on other bus types such as Universal Serial Bus (USB). A good example of how this is performed can be observed in Windows 95.

In addition to detecting devices and bus types at boot time, a system-based configuration manager or “sniffer” will be periodically run. Many of these devices support hot (power on) insertion and removal, so that perhaps every five to ten seconds or so, the configuration manager will attempt to determine if a new device has been inserted or an existing device removed. If the system detects that a new device has been installed, it looks up the configuration information for that device in the system’s device configuration table. If the configuration information is located, the configuration manager attempts to find and load the device driver associated with the device, and calls the initialization section of the device driver to perform its configuration process. If the device configuration information is not found in the system’s configuration table, the user is prompted to insert a diskette or path information to where the configuration information is located. The appropriate data is copied to the system's persistent store, and the configuration process begun. If the configuration process fails, the device driver is unloaded, and any resources claimed so far are automatically released[9].

Some devices cause immediate detection events, such as the insertion or removal of a PCMCIA card. Upon insertion of a PCMCIA card, the configuration data is read from tuples located in the PCMCIA card and compared with the persistent registry data (Mori, 1994). If the configuration information for the card already exists, the system loads the necessary driver and calls the driver's initialization code. The initialization code then configures the card using the configuration manager. Our proposed model supports older 16-bit PC cards as well as 32-bit CardBus PCI implementations (Anderson & Shanley, 1995). CardBus adapters can allocate 32-bit flat memory in any region of the PC memory address space.

The system registry must be able to support legacy devices whose resources are configurable only through jumpers or switches, or via a proprietary programming sequence (Kelsey, 1995). Some older devices require the booting of a MS-DOS diskette to download software to the adapter or configure the adapter's resources such as interrupt level, memory mapped address, DMA channels, or port addresses. The system registry utility is used to examine and manually enter the resources required by the legacy device. If the user attempts to claim a resource that is already in use, the request is refused[10]. In some cases, it will be necessary to write a "sniffer" that is specific to the particular device.

Addressing

Wherever possible, the system should avoid thunks. In particular, converting or **thunking** 32-bit addresses to 16-bit addresses are difficult to implement because the 32-bit pointer can cross 64KB boundaries (Deitel & Kogan, 1992). To avoid unnecessary thunks, the device driver interface should be implemented in 32-bit code, not thunked to their 16-bit counterpart. This will certainly cause the size of the kernel to grow, but the increased performance should more than make up for the larger kernel size or execution time. In a resource-constrained environment, it may be appealing to simply insert thunks for the 32-bit APIs. While this will save initial coding work, we believe it will have a significant impact on system performance.

Legacy Support

Although the system should provide seamless, flat-model interfaces to services and data structures, a few exceptions must be provided for. One of these exceptions is the support of **scatter-gather** lists for those devices that export only 24 address lines. Some SCSI adapters require that they perform reads and writes to certain physical addresses. Because the adapter recognizes only 24 bits of address, those physical addresses must reside below the 16MB boundary. Another exception to the exclusive use of 32-bit pointers is the use of Direct Memory Access, or DMA. The DMA hardware in most systems is a holdout of the ISA bus architecture, and is thus limited to 24 bits of address. Adapters that perform DMA reads and writes must use data buffers that are below the 16MB boundary. If the application is running above the 16MB boundary, the system must provide the ability to copy the data from the buffer below the 16MB boundary to the application's buffer in high memory. The device driver interface should do this transparently via a device driver service.

Current OS/2 segmented device drivers contain at least one code segment and one data segment, and are loaded in low physical memory (Deitel & Kogan, 1995). The new kernel must allow the new 32-bit device drivers to be loaded anywhere in physical memory, and not restrict them to any particular region[11]. In addition, the current 16-bit driver model restricts interrupt and timer handlers to the first code segment. Our proposal assumes that no such limitation will exist. The system should have the maximum flexibility in choosing how to distribute interrupt requests and should support Advanced Priority Interrupt Controller (APIC) if present (Anderson & Shanley, 1995).

The new device driver model should support a **symmetric multiprocessor** (SMP) configuration. Writing a device driver for an MP system requires care, just as it requires care to write an application for a multiprocessor system. Global variables should be avoided wherever possible (IBM, 1993), and platform-specific APIs for low-level access should be called rather than accessing the hardware directly. Although there may be some performance gain in bypassing the published hardware interfaces, the result will be code that runs on one particular platform but not another. Proper operation in an MP system requires serialized access to hardware and software resources. Manipulating hardware directly could cause these serialization functions to be ineffective, and introduce intermittent bugs and system failures that would be nearly impossible to locate. Race conditions are normal in an MP system. A thread blocked on one processor can be restarted on another processor or swapped out to disk at the system's discretion. Failing to adhere to the correct procedures could cause the system to become unstable and ultimately fail. All functions and subroutines for the new driver model shall be made MP safe.

Extreme care must be exercised when using global variables. Since device driver operations usually happen asynchronously, a condition might occur where two routines or functions attempt to access the global variable at the same time. Normally, global variables accessed by an application are protected by some type of semaphore mechanism to prevent this from happening. However, the device driver operates in kernel mode of the processor, and the semaphore functions are not available in kernel mode. One reason for this is that the OS/2 driver, while in kernel mode, is interruptible but not preemptible. While the OS/2 device driver is running it can be interrupted by an external device interrupt but it can not be suspended in favor of another higher priority device driver. This means that while the OS/2 device driver is executing, no other system service or program can be running.

The use of global variables becomes even more critical when the device driver is operating in a system with multiple processors. If a device driver becomes blocked on one processor, it can be restarted on another processor when a particular event occurs. OS/2 solves this by reflecting all interrupts to CPU0, while Windows NT uses a ring-0 semaphore mechanism called a **spin-lock**. Our proposal suggests a design where the device driver uses an object-oriented strategy by eliminating most global variables and encapsulating the variables inside functions. Real encapsulation is not possible at the driver level since we do not actually create an object in the traditional sense, but the function will access global variables that only the particular function is aware of. Since only one thread of the driver code can be executing at a time, this method provides serialized access to those variables. We believe that by adopting this strategy, all device drivers will be inherently MP-safe, providing they follow all the other guidelines for safe MP operation.

Chapter 3. - Methodology

Approach

Our approach involved the conversion of three specific device drivers from 16-bit to 32-bit. We felt that this was the best method, as we would have an existing model against which to verify our results and benchmark performance. Complete program listings of the three device drivers before and after conversion can be found in Appendix A.

Method and Database of Study

To validate our proposed design, we converted three existing 16-bit device drivers to the new 32-bit design. Because our design requires several other changes to the OS/2 kernel that may not be finished at the time we will be ready to begin testing, we were forced to simulate some operations in software. We built a software simulator that calls the device driver functions and executes as much code as possible. To monitor execution time, we took a snapshot of the millisecond timer upon entry into the device driver and another snapshot when exiting. While we could not test every execution path, this data did give us a general idea of execution time. While this was certainly not an optimal solution, it was possible to test the basic architecture of our design and compare some results with older 16-bit device drivers.

Validity of Data

Since we used three actual device drivers for our conversion, we believe the data contained herein to be valid, at least for these three cases. Since many components were not finished or installed at the time of our study, it was impossible to guarantee that we had uncovered all of the problems associated with the conversion. We have attempted to point out these unknown issues throughout this document, and have summarized them in Chapter 5.

Originality and Limitations of Data

Since the concept outlined in this document has never been attempted, data associated with the proposal has never been collected. While some general concepts, such as decreased performance and larger executable size are generally known and accepted, the actual application of those concepts in our proposal has not been previously documented.

It would have been easy to start with an entirely new device driver model, but this would render existing device drivers useless. The challenge is to provide an enhanced architecture while still maintaining backward compatibility with the hundreds of current OS/2 device drivers. This proposal suggests an architecture for the OS/2 base device driver model only. OS/2 uses several other types of device drivers, but most have already been converted to 32-bit models so they are not covered here.

Summary

The object file format of the device driver is the same as a DLL, so we loaded the device driver as a DLL for testing purposes. It was impossible to test real timer or device interrupts, since interrupts are only available in ring 0 and the DLL loads at ring 2. The simulator made calls to the timer and interrupt handlers to test their correct operation.

We encountered several problems and oversights with our initial design during the preliminary testing process. Some of these problems caused us to revise our design, and others required further system changes.

One concern that had been expressed to us is the fear that our design would cause the size of the installed imaged to grow very large. Indeed, our past experience had shown that converting 16-bit code to 32-bit code resulted in large increases in executable code size, sometimes in excess of 100 percent. We took no extra steps to optimize the executable size by changing the way we wrote the code to insure the comparisons with the 16-bit executables were valid.

During our testing, we noted the size increases for our new drivers and support software. They are discussed in detail later in Chapter 5.

Existing 16-bit device drivers primarily use 16-bit selectors and 16-bit offsets to form 32-bit addresses. Address and pointer conversions in 16-bit mode are very fast, especially when the memory range is equal to or less than 64KB. We expected that direct addressing using 32-bit pointers would be faster, especially since we could utilize the native block moves in the Intel processor. Our results are summarized in Chapter 5.

Drivers frequently create 32-bit alias pointers which point to other areas in memory, such as user buffers or memory-mapped areas of a device. The driver calls a system function to map a 16 or 32-bit physical address to a virtual or user virtual address. During this conversion, the system allocates a 32-bit selector from the pool of 32-bit selectors. Since OS/2 was designed to handle both 16-bit applications and 32-bit applications, we were unsure if the current pool of selectors available for allocation would be large enough, or contained enough 32-bit selectors to accommodate our new design. We wrote some test code to determine the number of selectors necessary.

In OS/2, application programs do not call device drivers directly. Instead, they call a system API that in turn calls the device driver through an established interface. This interface performs initial parameter validation and provides the ring transition from user mode to kernel mode and back. The communication medium between the APIs and the device driver is called the Request Packet. The request packet contains the operation code and parameters associated with the request such as pointers and data. OS/2 allocates a limited number of request packets, based on the available memory in the system heap space, so we were unsure that that OS/2 was allocating enough space for the packets. We later concluded that the system should provide up to 512 16-bit packets and 512 32-bit packets to insure that enough packets were available for even the largest number of devices.

In OS/2, the device header structure contains pointers to functions located in the device driver. It is a static structure created at compile and link time, and provides entry points for the initialization and strategy sections of the device driver. In the current implementation of OS/2, these pointers are kept only as 16-bit offsets since the driver model is 16-bit. With a flat model, these pointers will become 32-bit flat addresses. Programs or system components that need to call the device driver functions do so by locating the address of those functions from the driver’s header information. These system components need to be changed to refer to the address values correctly. We will insert a special bit in the device header to indicate the driver is a new flat model driver, and that addresses should be assumed to be linear.

The request packet contains pointers to buffers and data items that are located in the application program address space. Even though the driver is a 32-bit implementation, it is possible that the application that needs to access the device driver is still 16-bit. The system and device driver will allow these programs to be used by transparently handling any address conversions or address mapping.

Chapter 4. - Analysis of the Problem

How We Tested

For our first conversion from the 16-bit model to the new flat model, we decided to start with a simple device driver that is used to access the parallel port called MMAP.C (see Appendix A – Listings). Programmers have frequently used this public domain device driver as a basis for learning how to write OS/2 device drivers[12].

The device driver contains the two main sections of an OS/2 device driver, the **Initialization** section, and the **Strategy** section. In order to keep the device driver simple; the driver does not contain an interrupt handler or timer handler (See Appendix – Listings).

We began by identifying the variables that needed to be converted to accommodate our new model. Before modifying the driver variables, we decided to start with the local header files (see Appendix A - Listings) that are included by the device driver.

While operating in the 16-bit mode, the instruction set of the Intel processor allows two distinct types of addresses. To access data or instructions within a particular 64KB segment, a program can use **near** addressing. Near addressing assumes that the code or data being accessed resides in the same segment that is currently selected. Since the same segment is used, the only portion of the address that needs to change is the 16-bit offset.

Far addressing allows access to code or data anywhere within the 16MB memory area. In far addressing, the pointer contains a segment as well as an offset. Like near addressing, the offset portion specifies an address within a 64KB segment.

A 32-bit flat model program makes no distinction in addressing modes. All pointers and addresses are 32 bits long, and are referred to as **linear** addresses. In fact, the 32-bit C and C++ compilers do not allow the use of the common 16-bit modifiers *near* and *far*, so we began by deleting all of the near and far modifiers in the header files. For purposes of readability, we also deleted any references to the data type FARPOINTER, including any instances where FARPOINTER was used in function prototypes[13].

Header File Conversion

To avoid unnecessary duplication of 32-bit data types, we began by changing all of the unsigned short variables in the structure definitions to standard 32-bit data types defined in the 32-bit compiler’s header files. Variables that contain pointer data types are automatically expanded to 32-bits by the compiler when the compiler-defined pointer data type is used in the declaration. We also made sure that any variables that were not defined as pointers but that might hold a pointer temporarily were converted to 32-bit variables (see Appendix A - Listings).

We then converted any variables defined as BYTE or unsigned character to 16-bit. The reason we did this is that the compiler will attempt to optimize the storage of data structures by packing bytes together in adjacent memory locations. The data in the variables is sometimes passed by reference to another device driver or application. That is, the driver may send a pointer to the data structure to another device driver or application. The receiving driver or application may attempt to de-reference the pointer using a 16 or 32-bit pointer type, so the data must be word-aligned. Although we knew that memory requirements would be increased because of the larger data types, we felt that it was better to increase the memory size than to spend time debugging misaligned data structures. Unfortunately, we had to reverse this operation because we could not be sure that the system-level variables defined by these fields would be changed.

We then changed the system library function prototypes from NEAR PASCAL to PASCAL. The PASCAL parameter passing method is much easier to implement because a fixed number of parameters must be passed and the caller does not have to worry about cleaning up the **stack**. This makes even more sense because the device driver does not own the stack; therefore the device driver cannot manipulate the stack or stack pointer.

Driver Code Conversion

When we finished converting all of the header files, we began converting the variables in the device driver. OS/2 device drivers usually contain two types of variables, **global** and **local**. Global variables exist outside the C source code, and can be accessed by any function or subroutine in the device driver. Local variables are variables created within a particular function. Local variables are stored on the stack. We converted all of the pointers to the correct type, and then began changing some of the driver's internal structures. Whenever we encountered a structure with a built-in pointer, we changed that pointer to a standard 32-bit pointer type. We also changed an important part of the device driver, the **device header**. This structure defines the name of the driver, the type of device the driver supports, its capabilities, and pointers to major sections of internal device driver code. This has been a source of problems with the 16-bit device driver model. In the 16-bit design, the device header contains 16-bit offset pointers to the Initialization and Strategy sections of the device driver. The Initialization entry point is called when the device driver is loaded and the Strategy section is called for every other command sent to the driver. Because those pointers were only 16-bit pointers, the largest driver that could be written had to fit within a 64K-code segment. There was a workaround, though, by having the header point to an entry point in the first 64K segment. The first segment would then jump to a second code segment. Implementing this was quite tricky and resulted in some ugly code. Making these pointers 32-bit allows the driver to install an interrupt or timer handler anywhere in physical memory[14].

This is important because OS/2 loads all 16-bit device drivers in low memory (less than 1MB). This memory is a precious commodity, and it's easy to run out of space with several large multi-segment device drivers. Allowing the driver to be located anywhere in memory releases it from many historical limitations and helps free up lower memory. This means a major change to the system memory allocation routines, however, since the areas where device drivers are loaded must be permanently locked in memory and must be arranged as to prevent **fragmentation**. As part of our rewrite, we have specified that the OS/2 kernel must perform a post-boot compaction of device drivers as to avoid fragmentation. We also specified that the driver contains a special bit to override compaction.

Another limitation that we addressed was the entry points for timer and interrupt handler routines. When a device driver needs to install an interrupt or timer handler, it calls a system function, passing the entry point of the interrupt or timer handler code. Because these pointers were 16-bit, the timer and interrupt handlers had to reside in the first code segment so the handler could be reached with a 16-bit pointer. Once in the interrupt handler, the handler code could call or jump into another segment, but this was also difficult to implement. Crossing segment boundaries is time consuming because the cache becomes invalidated and has to be reloaded each time a 64K boundary is crossed.

Driver Loading

For this rewrite, we have specified that OS/2 device drivers can be loaded after the system is booted and running. This requires a large change in the system loader and support code, but we feel this will be made easier by allowing the device drivers to be loaded anywhere in memory. When loading device drivers, the system keeps a list of installed device drivers in a chain of pointers. The chain pointer is initialized to the address of the first device driver’s header. Each subsequent device driver header contains the address of the next device driver in the chain. The 16-bit device driver header uses an unsigned long (ULONG) for this pointer, so the system should be able to chain 16-bit and 32-bit device drivers without any extra logic in the chaining algorithm.

When the driver is opened, the system will check a specific bit in the device driver's header that identifies the driver as a new 32-bit driver. Once the determination has been made, the system can selectively send **request packets** that are formatted correctly for the model. A READ request packet, for example, contains the address of a data buffer in program space where data is to be stored. For the 16-bit model, those pointers will be passed as 16:16, for the 32-bit model, the pointers will be passed as linear addresses. The system will handle thunking 32-bit pointers to 16-bit, and 16-bit to 32-bit. By using this transparent thunking, current 16-bit drivers will continue to work normally while allowing the new 32-bit model to be supported. This is also important because OS/2 supports both 16-bit applications and 32-bit applications at the same time. It is possible, for example, for a 16-bit application to call a 32-bit driver, or a 32-bit application to call a 16-bit driver. Existing applications should continue to run unchanged, and to be able to utilize the new device driver model without a recompile[15].

Resource Allocation

Device drivers allocate and use system hardware and software resources to perform operations. If the device uses **DMA** to transfer data, the device driver must reserve (or have available) a DMA channel it can use when it needs to transfer data. If the device uses interrupts, the driver must reserve that interrupt so it can handle the interrupt from the device. If the driver waited until the last possible moment to see if the resource was available, it would fail if a program tried to use the device and the resource was not available. This might be satisfactory for a printer, but not for a cardiac monitor or bank teller machine.

Driver Information File

The device driver determines the acceptable hardware configuration by referring to a driver information file, or DIF (see Figure 4.1. - Sample DIF File.

```
; name of the driver
[DriverName]
Sample$
; file name and location
[DriverLocation]
c:\os2\drivers\sampledd.sys
; locator program
[Locator]
c:\os2\locator\samploc.exe
; ASCII description
[Description]
Sample base device driver
; 16 for 16-bit model, 32 for flat model
[DriverModel]
32
; free form ASCII text
[Company]
Our Company, Inc.
; version number nn.nn
[Version]
1.00
; hardware bus type and number
[Bus]
PCI:0 PCI:1
; interrupt levels IRQ:sharing (E=exclusive, S=shared)
[Interrupts]
5::E 7::E 9::E 11::E
; memory map (hex) address:length
[MemoryAddress]
c0000::1000 cc000::1000 d4000::1000 d8000::1000 dc000::1000
; dma channels
[DMA]
; ports (hex) address:length
[Ports]
300::16 280::16 220::16 200::16 260::16
; if adapter needs code downloaded to it
[Download]
c:\os2\drivers\init.cod::E0000
; specify driver power management support
[PowerManagement]
APM1.0
[ROM]
C8000
```

Figure 4.1. Sample DIF File

Using the information in the DIF file and the current configuration of the hardware, the driver registers for the resources it requires by calling the system's **Resource Manager**. The resource manager keeps track of the allocation and use of system resources. When a driver requests a resource, it calls the Resource Manager to request the use of that resource. The Resource Manager returns the status of that resource as AVAILABLE, NOT_AVAILABLE, or AVAILABLE_SHARED. Based on this information, the driver can determine what action to take. If the resource is available and the driver must absolutely have the resource, the driver calls the Resource Manager to reserve that resource exclusively. In some cases, the driver might allow sharing of that resource, but this is usually only safely done if the other driver or drivers sharing the resource are also aware that it will be shared. The driver uses the DIF file for a list of valid resources that the hardware supports. The DIF file also contains a list of substitute resources should the desired resource be unavailable. For instance, a particular device might request interrupt 5, but might also be able to use IRQ 4, IRQ 3, or IRQ 11. The DIF file contains a list of these resources in a hierarchy of desired resource allocation (see Figure 4.2).

```
[Interrupts]
5 4 3 11
```

Figure 4.2. DIF File Interrupt Entry

If the first resource in the category is unavailable, the driver will query the resource manager using the next supported resource until the request has been satisfied or all of the supported resources have been tried. The resources are organized in a hierarchical fashion, specifying the most optimal settings first followed by the least optimal. Using this method, the driver can determine the best configuration based on its intimate knowledge of the device. For instance, while configuring an interrupt, the driver could decide that based on the priority of the interrupt level it is granted, that the DMA channel should be an 8 bit DMA channel instead of a 16-bit DMA channel. The driver can make that decision based on its detailed knowledge of the device.

After the driver has been loaded, the user can change the values of these parameters using the Configuration Manager application. This application displays the current configuration and settings for all devices, and wherever possible, allows these settings to be changed. Settings that cannot be changed are displayed in a lighter color to indicate that they cannot be adjusted manually.

To make this concept work, existing 16-bit drivers should also include a DIF file and the associated code to request resources. It is not an absolute requirement, however.

Recovery

Each time the Configuration Manager is run, or a driver installed or changed, the current configuration is backed up to a file in case the current configuration causes the system to become unbootable or unstable. Each backup file is created with a different name using the current time and date. For example, a configuration file saved on October 23, 1996 would be called something like C102396.DAT and stored in the Configuration Manager's default directory.

When the system is rebooted, a menu will appear for 10 seconds that will allow the user to reboot using a different configuration than the current configuration. If the user selects this option, a menu of the current saved configurations is listed. The user highlights the desired configuration and continues the boot progress. If the boot process succeeds, the system will ask the user if the new configuration should replace the existing configuration. This will allow the user to boot the system with several experimental configurations without losing the last saved good configuration. The system always keeps a copy of the original configuration used when the operating system was installed to allow a default configuration to be used in the event that the system is rendered unbootable.

Device Locator

Each device driver must supply a device **locator** program that verifies the presence of the particular device and verifies its configuration. The configuration manager uses this program to detect if the device is present, and to locate the device driver in the event that it needs to install the device driver. The locator program name and path are located in the driver’s DIF file.

Configuration Manager

The OS/2 configuration manager should run every ten to fifteen seconds to detect the installation of new hardware devices. It queries the current configuration and compares it against the last stored configuration to see if any hardware has been added. To determine the current configuration, the configuration manager runs a list of locators. A locator is a program written by the device driver writer that queries the particular device to see if it is present. In addition to running every 10 seconds, the configuration manager will also be run if it detects a PCMCIA card insertion event. During system boot, one of the first drivers that are loaded is the configuration manager itself. At that time, the configuration manager driver signs up to receive power management and PCMCIA events. If a PCMCIA card is inserted or removed, the configuration manager is called to determine the new configuration.

The configuration manager will allow the user to enter settings for device drivers and hardware that do not participate in the new configuration architecture. Existing 16-bit device drivers may elect to supply a DIF file and the code to arbitrate resources with the configuration manager. For those drivers that can’t be modified, the configuration manager program will supply a manual input to allow resources for legacy devices and drivers to be reserved. If, for example, a certain device has a non-movable port address of 0x300, the user can reserve that port address using the resource manager.

Dynamic Loading

If the configuration manager detects that a new device has been installed, it looks in the \OS2\DRIVERS directory for a DIF file that matches the device. If it finds the DIF file, it calls the system device driver loader to load the device driver. If the device driver is loaded successfully, the locator associated with that device driver is added to the list of installed locators. The next time the configuration manager runs, the locator associated with that device will find the device already present.

If the correct DIF file cannot be found, the system will ask the user to enter a floppy disk that contains the DIF file and driver associated with the device. The system copies that DIF file and driver to the system's hard disk, and restarts the configuration process. In some cases, such as drive letter reassignment, rebooting the system may be required.

Dynamic Driver Binding

In current versions of OS/2, the device driver code uses static linking. Run time libraries, supplied by IBM and other vendors are linked with the device driver at build time and become part of the driver executable code. Because these driver libraries were created by different companies, the names of the functions are different, as well as the number and name of the function parameters. We plan to solve this problem by supplying a 32-bit standard driver run time library that is **dynamically linked** at driver load time. The driver links with the import library of the run time DLL just like a normal application links with a DLL's import library. The difference is that the driver gets its own copy of the run time library which becomes statically linked at driver load time, not at build time, resulting in much smaller driver executables. The driver library has the look and feel of a standard DLL, but a separate instance of the DLL is created for each driver. Unlike traditional DLLs, this DLL-like entity can be loaded at boot time and while the system is operating at ring 0. Updated library DLLs can be shipped via the normal update channels. This DLL could have been shipped as a standard library (LIB) file, but by using a DLL format, we can include debug symbol information and allow the library to be run and tested at ring 3. We can also insure that the current shipping library DLL is compatible with the current build of the operating system, and that the library is MP safe.

Debugger Modifications

One of the problems in developing OS/2 device drivers is the lack of good debugging tools. The current debugger is actually a replacement OS/2 kernel. It is loaded in place of the standard OS/2 kernel file, OS2KRNL. It provides support for debug information to be displayed and input from a standard terminal on a serial COM port. The current debugger is barely adequate, and does not support the display and modification of data structures. Commands and functions are cryptic and difficult to understand. We don’t feel, however, that the base command set should be changed. In spite of its weaknesses, developers have learned to use the debugger successfully, and we don’t want them to learn yet another command set. Since the debugger can already display and modify 32-bit values, it should be able to be used “as is” with few actual modifications.

Chapter 5. - Summary and Conclusions

Conclusions

It is worth reiterating here that the purpose of this effort was to flatten the *existing* base device driver model for OS/2. It would have been much easier to start from scratch with a brand new architecture and to ignore the support for the 16-bit model, but this was not an option.

Since the drivers we used were written in C, the conversion process was much easier than we expected. All three drivers were converted with no major problems. The C header file conversion was also easy. The first step was to get the drivers to compile and link successfully using a 32-bit compiler. For the compiler, we selected IBM’s Visual Age C++ compiler for OS/2. Using a compiler that is supplied by IBM makes the most sense since IBM also supplies the OS/2 operating system. We did not convert or use any assembly language stubs or programs because we did not want to supply a model that used them. Today’s C and C++ compilers do a great job at optimization, and any small performance gains realized by writing in assembler are insignificant compared to the software maintenance and support costs associated with assembler programming.

Recommended Steps

During the conversion process, we identified several steps that were necessary for the conversion of existing device drivers, and have arranged those steps above in Chapter 4, in their suggested order of implementation. This list is not complete because it only reflects the structure and coding techniques of the drivers we converted. Other drivers may use different coding techniques, library calls, or private data types, so the list should be used only as a general guideline when converting existing drivers.

Size

We were concerned that converting the drivers from 16-bit to 32-bit would cause an increase in size. Indeed, as we expected, the executable size of the drivers increased by an average of 30% or more, in one case the size increased 40%. However, given the choice of having no drivers or drivers that are larger, the choice is obvious.

Necessary Changes to OS/2

During the conversion process, we identified several major changes that must be implemented in the OS/2 kernel and file system to allow our new drivers to operate.

- All devices are accessed via the OS/2 **file system**. The file system is the operating system's interface to all devices and I/O. The current file system implementation is 16-bit. The file system must be modified to support 32-bit device drivers and their associated data. This means that the file system must support 32-bit pointers. The challenge here is that the file system should continue to support existing devices and device drivers while adding support for 32-bit drivers and devices. Request packets that are sent to a 16-bit device driver should have the appropriate 16-bit pointers, while request packets that are sent to 32-bit drivers should use 32-bit pointers. We recommend that the number of request packets allocated by the system be doubled to 512 16-bit packets and 512 32-bit packets. Since the system supplies the driver's stack, the stack pointers should be adjusted according to the driver model. Any system functions that are available to the device driver should have 16-bit and 32-bit entry points, allowing both models to coexist. For example, a call to set the entry point of the driver's interrupt or timer handler should accept a segment:offset for a 16-bit driver and a 0:32 flat address for a 32-bit device driver. The driver must maintain the ability to handle pointer conversions on behalf of 16-bit applications. It may be, in fact, easier to flatten the file system completely and then add back in the support for 16-bit applications and device drivers. This would have the theoretical effect of enhancing OS/2's support for large files and disk volumes, as well as faster file caching and buffering. The new file system must support long file names and large files[16].

- The OS/2 device driver loader must be modified to allow 16-bit and 32-bit device drivers to be loaded. 32-bit drivers can be loaded anywhere in memory, while 16-bit drivers will continue to be loaded in low memory[17]. The loader should not use the CONFIG.SYS file for loading device drivers, but should use the current configuration file to build a device tree while the system is booting up. The loader should support Dynamic Binding to allow the driver to be bound to an instance of the driver support library at load time.

- OS/2 must be modified to support file ring 0 file I/O services. This functionality is also required for the configuration manager to store the system configuration information.

- OS/2 must be modified to use the lower bits of the 8254 Counter-Timer to provide better granularity for system timer ticks and device driver profiling.

- OS/2 must supply a standard set of driver library functions that operate at ring 0 as well as ring 3. These libraries should export both 16-bit and 32-bit functions, and should be able to allocate and de-allocate memory using 32-bit linear addresses. These library routines must be MP safe and use the PASCAL calling convention.

- The system should be modified to supply what we call “auto thunking”. Auto thunking makes all the conversions from 16-bit pointers and stacks to 32-bit, and the reverse. A 16-bit device driver should always receive 16-bit pointers, while a 32-bit driver will always receive 32-bit pointers[18].

- OS/2 must supply a GUI-based configuration utility capable of displaying and editing the configuration file.

- A development kit must be supplied that contains accurate documentation on how to convert 16-bit device drivers, as well as the appropriate header files, make files, debugging kernel, and library support software.

Conclusions

It is our assessment that although this model will work, the device driver conversion effort is minimal when compared with the effort necessary to modify OS/2 to accommodate the new design. The biggest problem is that the underlying file system is still 16-bit, and flattening the file system is not a trivial task. We estimate that the file system alone could take one man-year or more of effort, and a lengthy additional period of integration testing with many different drivers and system configurations.

Bibliography

Anderson, Don (1995). CardBus System Architecture. New York: Mindshare.

Anderson, Don, and Shanley, Tom. (1993). Pentium Processor System Architecture. Texas: MindShare, Inc.

Deitel, Harvey M., and Kogan, Michael S. (1992). The Design of OS/2. New York: Addison-Wesley.

Hazzah, Karen. (1995). Writing Windows VxDs and Device Drivers. Kansas: R&D Publications.

IBM Corporation, (1993). Device Drivers for OS/2 SMP: A Brief Overview. Florida: IBM Corporation.

IBM Corporation. (1993). OS/2 for SMP V2.11 Reference. Florida: Author.

IBM Corporation. (1996). Visual Age C++ for OS/2 Version 1.0 Class Library Reference. Ontario, Canada: Author.

Kelsey, James. (1995). Programming Plug and Play. Indiana: Sams.

Leong, Kevin, Law, William, Love, Robert, Tsuji, Hiroshi, and Olson, Bruce. (1995). The OS/2 C++ Class Library. New York: John Wiley and Sons.

Mastrianni, Steven J. (1993). Writing OS/2 Device Drivers in C. New York: Van Nostrand Reinhold.

Mori, Michael T. (1994). The PCMCIA Developer's Guide. California: Sycard Technology.

Shanley, Tom. (1995). Plug and Play System Architecture. New York: Mindshare.

Shanley, Tom, and Anderson, Don. (1995). PCI System Architecture. New York: Addison Wesley.

Stroustrup, Bjarne. (1993). The C++ Programming Language, New York: Addison Wesley.

Tanenbaum, Andrew S., and Woodhull, Albert S. (1997). Operating Systems, 2nd Edition. New Jersey: Simon and Schuster.

Thielen, David, and Woodruff, Bryan. (1994). Writing Windows Virtual Device Drivers, New York: Addison-Wesley.

Appendix A – Listings

```
// sample 8255 parallel port device driver for OS/2

#include "drvlib.h"
#include "digio.h"

extern void STRATEGY();      /* name of strat rout. in drvstart*/
extern void TIMER_HANDLER(); /* timer handler in drvstart      */

DEVICEHDR devhdr = {
   (void far *) 0xFFFFFFFF,  /* link                           */
   (DAW_CHR | DAW_OPN | DAW_LEVEL1),/* attribute word          */
   (OFF) STRATEGY,           /* &strategy                      */
   (OFF) 0,                  /* &IDC routine                   */
   "DIGIO$  "                /* name/#units                    */
};

FPFUNCTION  DevHlp=0;         /* pointer to DevHlp entry point */
UCHAR       opencount = 0;    /* keeps track of open's         */
USHORT      savepid=0;        /* save thread pid               */
LHANDLE     lock_seg_han;     /* handle for locking appl. seg  */
PHYSADDR    appl_buffer=0;    /* address of caller's buffer    */
ERRCODE     err=0;            /* error return                  */
ULONG       ReadID=0L;        /* current read pointer          */
USHORT      num_rupts=0;      /* count of interrupts           */
USHORT      temp_char;        /* temp character for in-out     */
void        far *ptr;         /* temp far pointer              */
FARPOINTER  appl_ptr=0;       /* pointer to application buffer */
char        input_char,output_char; /* temp character storage  */
char        input_mask;       /* mask for input byte           */

/* messages */

char        CrLf[]= "\r\n";
char        InitMessage1[] = " 8 bit Digital I/O ";
char        InitMessage2[] = " driver installed\r\n";
char        FailMessage[]  = " driver failed to install.\r\n";

/* common entry point for calls to Strategy routines */

int main(PREQPACKET rp)
{
   void far *ptr;
   PLINFOSEG liptr;             /* pointer to global info seg  */
   int i;

   switch(rp->RPcommand)
   {
     case RPINIT:               /* 0x00                       */
       /* init called by kernel in protected mode */
       return Init(rp);

     case RPREAD:               /* 0x04                       */
       rp->s.ReadWrite.count = 0; /* in case we fail          */
       input_char = inp(DIGIO_INPUT);/* get data              */
       if (PhysToVirt( (ULONG) rp->s.ReadWrite.buffer,
                       1,0,&appl_ptr))
           return (RPDONE | RPERR | ERROR_GEN_FAILURE);
```

```
  if (MoveBytes((FARPOINTER)&input_char,appl_ptr,1))
      return (RPDONE | RPERR | ERROR_GEN_FAILURE);
  rp->s.ReadWrite.count = 1;  /* one byte read           */
  return (RPDONE);

case RPWRITE:                /* 0x08                    */
  rp->s.ReadWrite.count = 0;
  if (PhysToVirt( (ULONG) rp->s.ReadWrite.buffer,
                1,0,&appl_ptr))
       return (RPDONE | RPERR | ERROR_GEN_FAILURE);
  if (MoveBytes(appl_ptr,(FARPOINTER)&output_char,1))
       return (RPDONE | RPERR | ERROR_GEN_FAILURE);
  outp (DIGIO_OUTPUT,output_char); /* send byte          */
  rp->s.ReadWrite.count = 1; /* one byte written         */
  return (RPDONE);

case RPOPEN:                 /* 0x0d open driver         */
  /* get current process id */
  if (GetDOSVar(2,&ptr))
      return (RPDONE | RPERR | ERROR_BAD_COMMAND);
  /* get process info */
  liptr = *((PLINFOSEG far *) ptr);
  /* if this device never opened, can be opened by anyone*/
  if (opencount == 0)      /* first time this dev opened */
  {
      opencount=1;         /* bump open counter          */
      savepid = liptr->pidCurrent; /* save current PID   */
  }
  else
  {
      if (savepid != liptr->pidCurrent) /* another proc  */
          return (RPDONE | RPERR | ERROR_NOT_READY);/*err*/
      ++opencount;         /* bump counter, same pid     */
  }
  return (RPDONE);

case RPCLOSE:                /* 0x0e DosClose,ctl-C, kill*/
  /* get process info of caller */
  if (GetDOSVar(2,&ptr))
      return (RPDONE | RPERR | ERROR_BAD_COMMAND);
  /* get process info from os/2 */
  liptr= *((PLINFOSEG far *) ptr); /* ptr to linfoseg    */
  /*
   make sure that process attempting to close this device
   is the one that originally opened it and the device was
   open in the first place.
  */
  if (savepid != liptr->pidCurrent || opencount == 0)
      return (RPDONE | RPERR | ERROR_BAD_COMMAND);
  --opencount;             /* close counts down open cntr*/
  return (RPDONE);         /* return 'done' status       */

case RPIOCTL:              /* 0x10                       */
  /*
   The function code in an IOCtl packet has the high bit set
   for the DIGIO$ board. We return all others with the done
   bit set so we don't have to handle things like the 5-48
   code page IOCtl
  */
  if (rp->s.IOCtl.category != DIGIO_CAT)/* other IOCtls  */
      return (RPDONE | RPERR | ERROR_BAD_COMMAND);

  switch (rp->s.IOCtl.function)
```

```
{
   case 0x01:              /* write byte to digio port   */
      /* verify caller owns this buffer area */
      if(VerifyAccess(
      SELECTOROF(rp->s.IOCtl.parameters), /* selector    */
      OFFSETOF(rp->s.IOCtl.parameters),   /* offset      */
      1,                                  /* 1 byte      */
      0) )                                /* read only   */
          return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      if(MoveBytes(rp->s.IOCtl.parameters,
                  (FARPOINTER)&output_char,1))
          return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      outp(DIGIO_OUTPUT,output_char);     /*send to digio*/
      return (RPDONE);

   case 0x02:              /* read byte w/wait from port */
      /* verify caller owns this buffer area */
      if(VerifyAccess(
      SELECTOROF(rp->s.IOCtl.buffer), /* selector        */
      OFFSETOF(rp->s.IOCtl.buffer),   /* offset          */
      1,                              /* 1 bytes)        */
      0))                             /* read only       */
         return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      /* lock the segment down temp */
      if(LockSeg(
      SELECTOROF(rp->s.IOCtl.buffer), /* selector        */
      1,                              /* lock forever    */
      0,                              /* wait for seg loc*/
      (PLHANDLE) &lock_seg_han))    /* handle returned */
         return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      if(MoveBytes(rp->s.IOCtl.parameters,
                  (FARPOINTER)&input_mask,1))
          return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      /* wait for switch to be pressed */
      ReadID = (ULONG)rp;           /* block ID        */
      if (Block(ReadID,-1L,0,&err))
        if (err == 2)
          return(RPDONE | RPERR
                 | ERROR_CHAR_CALL_INTERRUPTED);

      /* move data to users buffer */
      if(MoveBytes((FARPOINTER)&input_char,
         rp->s.IOCtl.buffer,1))
          return(RPDONE | RPERR | ERROR_GEN_FAILURE);
      /* unlock segment */
      if(UnLockSeg(lock_seg_han))
          return(RPDONE | RPERR | ERROR_GEN_FAILURE);
      return (RPDONE);

   case 0x03:             /* read byte immed digio port */
      /* verify caller owns this buffer area */
      if(VerifyAccess(
      SELECTOROF(rp->s.IOCtl.buffer), /* selector        */
      OFFSETOF(rp->s.IOCtl.buffer),   /* offset          */
      4,                              /* 4 bytes         */
      0))                             /* read only       */
          return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      input_char = inp(DIGIO_INPUT);  /* get data        */
      if(MoveBytes((FARPOINTER)&input_char,
         rp->s.IOCtl.buffer,1))
          return(RPDONE | RPERR | ERROR_GEN_FAILURE);
      return (RPDONE);
   default:
```

```
                return(RPDONE | RPERR | ERROR_GEN_FAILURE);
          }

   /* don't allow deinstall */
   case RPDEINSTALL:            /* 0x14                       */
       return(RPDONE | RPERR | ERROR_BAD_COMMAND);
    /* all other commands are flagged as bad */
   default:
       return(RPDONE | RPERR | ERROR_BAD_COMMAND);
    }
}

timr_handler()
{
    if (ReadID != 0)
    {
      /* read data from port */
      input_char = inp(DIGIO_INPUT );/* get data               */
      if ((input_char && input_mask) !=0)
      {
        Run (ReadID);
        ReadID=0L;
      }
    }
}

/* Device Initialization Routine */

int Init(PREQPACKET rp)
{
    /* store DevHlp entry point */
    DevHlp = rp->s.Init.DevHlp;
    /* install timer handler */
    if(SetTimer((PFUNCTION)TIMER_HANDLER)) {
       /* if we failed, deinstall driver with cs+ds=0 */
       DosPutMessage(1, 8, devhdr.DHname);
       DosPutMessage(1,strlen(FailMessage),FailMessage);
       rp->s.InitExit.finalCS = (OFF) 0;
       rp->s.InitExit.finalDS = (OFF) 0;
       return (RPDONE | RPERR | ERROR_GEN_FAILURE);
    }
    /* configure 8255 parallel chip */
    outp (DIGIO_CONFIG,0x91);
    /* output initialization message */
    DosPutMessage(1, 2, CrLf);
    DosPutMessage(1, 8, devhdr.DHname);
    DosPutMessage(1, strlen(InitMessage1), InitMessage1);
    DosPutMessage(1, strlen(InitMessage2), InitMessage2);
    /* send back our code and data end values to os/2 */
    if (SegLimit(HIUSHORT((void far *) Init),
        &rp->s.InitExit.finalCS) || SegLimit(HIUSHORT((void far *)
        InitMessage2), &rp->s.InitExit.finalDS))
          Abort();
    return(RPDONE);
}
```

Figure A-1. OS/2 parallel device driver, 16-bit version.

```
/*
 digio.h memory map for os/2 device driver
*/

#define  DIGIO_CAT    0x91      /* category for DosDevIOCtl     */
#define  DIGIO_BASE   0x2c0     /* board address                */
#define  DIGIO_OUTPUT DIGIO_BASE /* output port                 */
#define  DIGIO_INPUT  DIGIO_BASE+1 /* input port                */
#define  DIGIO_CONFIG DIGIO_BASE+3 /* initialization port       */
```

Figure A-2. Parallel driver header file.

```
// file drvlib.h
// This header file contains definitions intended to go with
// DRVLIB.LIB, a C-callable subroutine library.
//
//   This file is for OS/2 2.0
typedef unsigned char   UCHAR;
typedef unsigned short  USHORT;
typedef unsigned short  BOOLEAN;
typedef unsigned long   ULONG;
typedef UCHAR near      *PUCHAR;
typedef UCHAR far       *FPUCHAR;
typedef USHORT near     *PUSHORT;
typedef USHORT far      *FPUSHORT;
typedef ULONG near      *PULONG;
typedef ULONG far       *FPULONG;
typedef char near       *PCHAR;
typedef short near      *PSHORT;
typedef long near       *PLONG;
typedef void near       *POINTER;
typedef POINTER near    *PPOINTER;
typedef void far        *FARPOINTER;
typedef FARPOINTER near *PFARPOINTER;
typedef FARPOINTER far  *FPFARPOINTER;

typedef USHORT          ERRCODE;    // error code returned
typedef ERRCODE far     *PERRCODE;  // pointer to an error code
typedef UCHAR           FLAG;       // 8-bit flag
typedef FLAG far        *PFLAG;     // pointer to 8-bit flag
typedef USHORT          SEL;        // 16-bit selector
typedef SEL near        *PSEL;      // pointer to a selector
typedef SEL far         *FPSEL;     // far pointer to selector
typedef USHORT          SEG;        // 16-bit segment
typedef USHORT          OFF;        // 16-bit offset
typedef ULONG           LOFF;       // 32-bit offset
typedef USHORT          PID;        // Process ID
typedef USHORT          TID;        // Thread ID
typedef ULONG           PHYSADDR;   // 32-bit physical address
typedef ULONG           LINADDR;    // 32-bit linear address
typedef LINADDR  far    *PLINADDR;  // pointer to 32 bit address
typedef PLINADDR far    *PPLINADDR; // pointer to linear address
typedef PHYSADDR far    *PPHYSADDR; // pointer to 32-bit phys addr
typedef char near       *PSTRING;   // pointer to character string
typedef char far        *FPSTRING;  // far pointer to string
typedef USHORT          SHANDLE;    // short (16-bit) handle
typedef SHANDLE far     *PSHANDLE;  // pointer to a short handle
typedef ULONG           LHANDLE;    // long  (32-bit) handle
typedef LHANDLE far     *PLHANDLE;  // pointer to a long handle

//  pointers to functions
typedef int (pascal near          *PFUNCTION) ();
typedef int (pascal near * near  *PPFUNCTION) ();
typedef int (pascal far          *FPFUNCTION) ();
typedef int (pascal far  * near *PFPFUNCTION) ();

// macros
#define FALSE   0
#define TRUE    1
#define NP near pascal

// far pointer from selector-offset
#define MAKEP(sel, off)     ( (void far *) MAKEULONG(off, sel) )

// get selector or offset from far pointer
#define SELECTOROF(p)       ( ((USHORT far *) &(p)) [1])
#define OFFSETOF(p)         ( ((USHORT far *) &(p)) [0])

// Combine l(ow) & h(igh) to form a 32 bit quantity.
#define MAKEULONG(l, h)  ((ULONG)(((USHORT)(l))
                  |((ULONG)((USHORT)(h))) << 16))
#define MAKELONG(l, h)   ((LONG)MAKEULONG(l, h))
#define MAKEBIGOFFSETOF(p) ((ULONG) (OFFSETOF (p)))

// Combine l(ow) & h(igh) to form a 16 bit quantity.
#define MAKEUSHORT(l, h) (((USHORT)(l)) | ((USHORT)(h)) << 8)
#define MAKESHORT(l, h)  ((SHORT)MAKEUSHORT(l, h))

// get high and low order parts of a 16 and 32 bit quantity
#define LOBYTE(w)       LOUCHAR(w)
#define HIBYTE(w)       HIUCHAR(w)
#define LOUCHAR(w)      ((UCHAR)(w))
#define HIUCHAR(w)      (((USHORT)(w) >> 8) & 0xff)
#define LOUSHORT(l)     ((USHORT)(l))
#define HIUSHORT(l)     ((USHORT)(((ULONG)(l) >> 16) & 0xffff))

//  the driver device header
typedef struct DeviceHdr
{
   struct DeviceHdr far *DHnext;// pointer to next header, or FFFF
   USHORT DHattribute;          // device attribute word
   OFF    DHstrategy;           // offset of strategy routine
   OFF    DHidc;                // offset of IDC routine
   UCHAR  DHname[8];            // dev name (char) or #units (blk)
   char   reserved[8];
   ULONG  bit_strip;            // bit 0 DevIOCtl2
} DEVICEHDR;
typedef DEVICEHDR near *PDEVICEHDR;

//  driver device attributes word
#define DAW_CHR    0x8000       // 1=char, 0=block
#define DAW_IDC    0x4000       // 1=IDC available in this DD
#define DAW_IBM    0x2000       // 1=non-IBM block format
#define DAW_SHR    0x1000       // 1=supports shared device access
#define DAW_OPN    0x0800       // 1=open/close, removable media
#define DAW_LEVEL1 0x0080       // level 1
#define DAW_LEVEL2 0x0100       // level 2 DosDevIOCtl2
#define DAW_LEVEL3 0x0180       // level 3 bit strip
#define DAW_GIO    0x0040       // 1=generic IOCtl supported
#define DAW_CLK    0x0008       // 1=CLOCK device
#define DAW_NUL    0x0004       // 1=NUL device
```

```
#define DAW_SCR    0x0002       // 1=STDOUT (screen)
#define DAW_KBD    0x0001       // 1=STDIN  (keyboard)

// capabilities bit strip
#define CBS_SHD    0x0001       // 1=shutdown/DevIOCtl2
#define CBS_HMEM   0x0002       // high memory map for adapters
#define CBS_PP     0x0004       // supports parallel ports
#define CBS_ADD    0x0010       // driver is an ADD
#define CBS_INIT   010020       // driver receives InitComplete

// SaveMessage structure
typedef struct MessageTable
{
   USHORT      id;
   USHORT      fill_in_item;
   FARPOINTER  item1;
   FARPOINTER  item2;
   FARPOINTER  item_last;
} MESSAGETABLE;

// OS/2 circular character queues
#define QUEUE_SIZE  512         //  size of queues
typedef struct CharQueue
{
   USHORT   qsize;              // number of bytes in queue
   USHORT   qchrout;            // index of next char to put out
   USHORT   qcount;             // number of charactes in queue
   UCHAR    qbuf[QUEUE_SIZE];
} CHARQUEUE;
typedef CHARQUEUE near *PCHARQUEUE;

// AttachDD inter device driver communication data area
typedef struct AttachArea
{
   OFF realOFF;                 // offset of idc entry point
   SEG realCS;                  // real-mode CS of IDC entry point
   SEG realDS;                  // real-mode DS of IDC DD
   OFF protOFF;                 // protect-mode offset of entry pt
   SEL protCS;                  // protect-mode CS of entry point
   SEL protDS;                  // protect-mode DS of other DD
} ATTACHAREA;
typedef ATTACHAREA near *PATTACHAREA;

// driver request packet
typedef struct ReqPacket
{
   UCHAR RPlength;              // request packet length
   UCHAR   RPunit;              // unit code for block DD only
   UCHAR RPcommand;             // command code
   USHORT  RPstatus;            // status word
   UCHAR   RPreserved[4];       // reserved bytes
   ULONG RPqlink;               // queue linkage
   union
   {                            // command-specific data
     UCHAR   avail[19];
     struct
     {                          // init
       UCHAR      units;         // number of units
       FPFUNCTION DevHlp;        // &DevHlp
       char far   *args;         // &args
       UCHAR      drive;         // drive #
     }Init;
     struct
```

```
{
  UCHAR     units;          // same as input
  OFF       finalCS;        // final offset, 1st code segment
  OFF       finalDS;        // final offset, 1st data segment
  FARPOINTER BPBarray;      // &BPB
} InitExit;
struct
{                           // read, write, write w/verify
  UCHAR      media;         // media descriptor
  PHYSADDR   buffer;        // transfer address
  USHORT     count;         // bytes/sectors
  ULONG      startsector;   // starting sector#
  USHORT     reserved;
} ReadWrite;
struct
{                           // cached rd, wr, write w/verify
  UCHAR      media;         // media descriptor
  PHYSADDR   buffer;        // transfer address
  USHORT     count;         // bytes/sectors
  ULONG      startsector;   // starting sector#
  USHORT     reserved;
} CReadWrite;
struct
{                           // system shutdown
  UCHAR      subcode;       // sub request code
  ULONG      reserved;
} Shutdown;
struct
{                           // open/close
  USHORT     sysfilenum;    // system file number
} OpenClose;
struct
{                           // IOCtl
  UCHAR      category;      // category code
  UCHAR      function;      // function code
  FARPOINTER parameters;    // &parameters
  FARPOINTER buffer;        // &buffer
} IOCtl;
struct
{                           // read, no wait
  UCHAR      char_returned; // char to return
} ReadNoWait;
struct
{                           // media check
  UCHAR      media;         // media descriptor
  UCHAR      return_code;   // see #defines
  FARPOINTER prev_volume;   // &previous volume ID
} MediaCheck;
struct
{                           // build BPB
  UCHAR      media;         // media descriptor
  FARPOINTER buffer;        // 1-sector buffer FAT
  FARPOINTER BPBarray;      // &BPB array
  UCHAR      drive;         // drive #
} BuildBPB;
struct
{                           // query part. fixed disks
  UCHAR      count;         // # disks
  ULONG      reserved;
} Partitionable;
struct
{                           // fixed disk LU map
  ULONG      units;         // units supported
```

```
      ULONG      reserved;
    } GetFixedMap;
    struct
    {                            // get driver capabilities
      UCHAR      reserved[3];
      FARPOINTER capstruct;      // 16:16 pointer to DCS
      FARPOINTER volcharstruct;  // 16:16 pointer to VCS
    } GetDriverCaps;
  } s;                           // command info
} REQPACKET;

typedef REQPACKET far *PREQPACKET;
typedef PREQPACKET far *PPREQPACKET;
typedef PREQPACKET QHEAD;        // Queue Head is &ReqPacket
typedef QHEAD near *PQHEAD;

// Global Info Seg
typedef struct _GINFOSEG
{
  ULONG   time;
  ULONG   msecs;
  UCHAR   hour;
  UCHAR   minutes;
  UCHAR   seconds;
  UCHAR   hundredths;
  USHORT  timezone;
  USHORT  cusecTimerInterval;
  UCHAR   day;
  UCHAR   month;
  USHORT  year;
  UCHAR   weekday;
  UCHAR   uchMajorVersion;
  UCHAR   uchMinorVersion;
  UCHAR   chRevisionLetter;
  UCHAR   sgCurrent;
  UCHAR   sgMax;
  UCHAR   cHugeShift;
  UCHAR   fProtectModeOnly;
  USHORT  pidForeground;
  UCHAR   fDynamicSched;
  UCHAR   csecMaxWait;
  USHORT  cmsecMinSlice;
  USHORT  cmsecMaxSlice;
  USHORT  bootdrive;
  UCHAR   amecRAS[32];
  UCHAR   csgWindowableVioMax;
  UCHAR   csgPMMax;
} GINFOSEG;
typedef GINFOSEG far *PGINFOSEG;

// local info seg
typedef struct _LINFOSEG
{
  PID     pidCurrent;
  PID     pidParent;
  USHORT  prtyCurrent;
  TID     tidCurrent;
  USHORT  sgCurrent;
  UCHAR   rfProcStatus;
  UCHAR   dummy1;
  USHORT  fForeground;
  UCHAR   typeProcess;
  UCHAR   dummy2;
  SEL     selEnvironment;
  USHORT  offCmdLine;
  USHORT  cbDataSegment;
  USHORT  cbStack;
  USHORT  cbHeap;
  USHORT  hmod;
  SEL     selDS;
} LINFOSEG;

typedef LINFOSEG far *PLINFOSEG;
typedef struct _REGSTACK
{                          // stack usage structure
  USHORT  usStruct;        // set to 14 before using
  USHORT  usFlags;         // 0x01 means that the interrupt proc
                           // enables interrupts. All others resvd
  USHORT  usIRQ;           // IRQ of interrupt handler
  USHORT  usStackCLI;      // # of stack bytes with interrupts off
  USHORT  usStackSTI;      // # of stack bytes with interrupts on
  USHORT  usStackEOI;      // number of bytes needed after EOI
  USHORT  usNest;          // max number of nested levels         } REGSTACK;
typedef REGSTACK near *PREGSTACK;

// page list struct
typedef struct _PAGELIST
{
  ULONG pl_Physaddr;
  ULONG pl_cb;
} PAGELIST;
typedef PAGELIST far *PPAGELIST;

// RPstatus bit values
#define RPERR   0x8000     //  error occurred, err in RPstatus
#define RPDEV   0x4000     //  error code defined by driver
#define RPBUSY  0x0200     //  device is busy
#define RPDONE  0x0100     //  driver done with request packet

// error codes returned in RPstatus
#define ERROR_WRITE_PROTECT         0x0000
#define ERROR_BAD_UNIT              0x0001
#define ERROR_NOT_READY             0x0002
#define ERROR_BAD_COMMAND           0x0003
#define ERROR_CRC                   0x0004
#define ERROR_BAD_LENGTH            0x0005
#define ERROR_SEEK                  0x0006
#define ERROR_NOT_DOS_DISK          0x0007
#define ERROR_SECTOR_NOT_FOUND      0x0008
#define ERROR_OUT_OF_PAPER          0x0009
#define ERROR_WRITE_FAULT           0x000A
#define ERROR_READ_FAULT            0x000B
#define ERROR_GEN_FAILURE           0x000C
#define ERROR_DISK_CHANGE           0x000D
#define ERROR_WRONG_DISK            0x000F
#define ERROR_UNCERTAIN_MEDIA       0x0010
#define ERROR_CHAR_CALL_INTERRUPTED 0x0011
#define ERROR_NO_MONITOR_SUPPORT    0x0012
#define ERROR_INVALID_PARAMETER     0x0013
#define ERROR_DEVICE_IN_USE         0x0014

// driver request codes  B=block, C=character
#define RPINIT          0x00        //  BC
#define RPMEDIA_CHECK   0x01        //  B
#define RPBUILD_BPB     0x02        //  B
#define RPREAD          0x04        //  BC
```

```
#define RPREAD_NO_WAIT  0x05        //   C
#define RPINPUT_STATUS  0x06        //   C
#define RPINPUT_FLUSH   0x07        //   C
#define RPWRITE         0x08        //  BC
#define RPWRITE_VERIFY  0x09        //  BC
#define RPOUTPUT_STATUS 0x0a        //   C
#define RPOUTPUT_FLUSH  0x0b        //   C
#define RPOPEN          0x0d        //  BC
#define RPCLOSE         0x0e        //  BC
#define RPREMOVABLE     0x0f        //  B
#define RPIOCTL         0x10        //  BC
#define RPRESET         0x11        //  B
#define RPGET_DRIVE_MAP 0x12        //  B
#define RPSET_DRIVE_MAP 0x13        //  B
#define RPDEINSTALL     0x14        //   C
#define RPPARTITIONABLE 0x16        //  B
#define RPGET_FIXED_MAP 0x17        //  B
#define RPSHUTDOWN      0x1c        //  BC
#define RPGET_DRIVER_CAPS 0x1d      //  B

// check for monitor call in DosOpen/DosClose
#define MON_OPEN_STATUS   0x08      // open from DosMonOpen
#define MON_CLOSE_STATUS  0x08      // close from DosMonClose

// media descriptor byte
#define MDB_REMOVABLE     0x04      //  1=removable
#define MDB_EIGHT_SECTORS 0x02      //  1=8 sectors per track
#define MDB_DOUBLE_SIDED  0x01      //  1=double-sided media

// return codes from MediaCheck
#define MC_MEDIA_UNCHANGED 0x01
#define MC_MEDIA_CHANGED   0xFF
#define MC_MEDIA_UNSURE    0x00

// event numbers for SendEvent

#define EVENT_SM_MOUSE   0x00       // session switch via mouse
#define EVENT_CTRLBRK    0x01       // control break
#define EVENT_CTRLC      0x02       // control C
#define EVENT_CTRLNUMLK  0x03       // control num lock
#define EVENT_CTRLPRTSC  0x04       // control printscreen
#define EVENT_SHFTPRTSC  0x05       // shift printscreen
#define EVENT_SM_KBD     0x06       // session switch hot key

// defines for 1.x movedata function
#define   MOVE_PHYSTOPHYS   0       // move phys to phys memory
#define   MOVE_PHYSTOVIRT   1       // move phys to virt memory
#define   MOVE_VIRTTOPHYS   2       // move virt to phys memory
#define   MOVE_VIRTTOVIRT   3       // move virt to virt memory

// Micro Channel specific
int NP GetLIDEntry (USHORT,USHORT,USHORT,FPUSHORT);
int NP FreeLIDEntry (USHORT);
int NP ABIOSCall (USHORT,USHORT,FARPOINTER);
int NP ABIOSComm (USHORT,FARPOINTER);
int NP GetDeviceBlock(USHORT,FARPOINTER);

// special routines
void NP INT3  (void);
void NP Enable  (void);
void NP Disable  (void);
void NP Abort  (void);
int  NP SegLimit (SEL,OFF far *);
int  NP MoveBytes (FARPOINTER,FARPOINTER,FLAG);
int  NP MoveData (FARPOINTER,FARPOINTER,USHORT,USHORT);

// system services and misc.
int  NP GetDOSVar (USHORT,FPFARPOINTER);
int  NP SendEvent (USHORT,USHORT);
void NP SchedClockAddr (PFARPOINTER);
int  NP AttachDD (PSTRING,PATTACHAREA);
int  NP InternalError(PSTRING,USHORT);
int  NP SaveMessage(FPSTRING);
int  NP ProtToReal(void);
int  NP RealToProt(void);
int  NP SetROMVector(USHORT,PFUNCTION,PFUNCTION,FARPOINTER);

// process mgmt
void NP Yield  (void);
void NP TCYield  (void);
int  NP Block  (ULONG,ULONG,USHORT,FARPOINTER);
void NP Run  (ULONG);
void NP DevDone  (PREQPACKET);
int  NP VideoPause(USHORT);

// memory management
int  NP AllocPhys (ULONG,USHORT,PPHYSADDR);
int  NP FreePhys (PHYSADDR);
int  NP VerifyAccess (SEL,OFF,USHORT,USHORT);
int  NP LockSeg  (SEL, USHORT,USHORT,PLHANDLE);
int  NP UnLockSeg (LHANDLE);

// address conversion
int  NP AllocGDTSelector(USHORT,FARPOINTER);
int  NP PhysToGDTSelector(PHYSADDR,USHORT,SEL,PERRCODE);
int  NP VirtToPhys (FARPOINTER,PPHYSADDR);
int  NP PhysToUVirt (PHYSADDR,USHORT,USHORT,FPFARPOINTER);
int  NP PhysToVirt (PHYSADDR,USHORT,USHORT,FARPOINTER);
int  NP UnPhysToVirt (void);

// request packet queue stuff
int  NP AllocReqPacket(USHORT,PPREQPACKET);
void NP FreeReqPacket (PREQPACKET);
void NP PushReqPacket (PQHEAD,PREQPACKET);
void NP SortReqPacket (PQHEAD,PREQPACKET);
int  NP PullReqPacket (PQHEAD,PPREQPACKET);
int  NP PullParticular(PQHEAD,PREQPACKET);

// driver semaphores
int  NP SemHandle (LHANDLE,FLAG,PLHANDLE);
int  NP SemRequest (LHANDLE,ULONG,PERRCODE);
void NP SemClear (LHANDLE);

// circular character queues
void NP QueueInit (PCHARQUEUE);
void NP QueueFlush (PCHARQUEUE);
int  NP QueueWrite (PCHARQUEUE,UCHAR);
int  NP QueueRead (PCHARQUEUE,FPUCHAR);

// interrupt stuff
int  NP SetIRQ  (USHORT,PFUNCTION,USHORT);
int  NP UnSetIRQ (USHORT);
int  NP EOI  (USHORT);
void NP ClaimInterrupt(void);
void NP RefuseInterrupt(void);
int  NP RegisterStackUsage(PREGSTACK);
```

```
// timer stuff
int  NP SetTimer (PFUNCTION);
int  NP ResetTimer (PFUNCTION);
int  NP TickCount (PFUNCTION,USHORT);

// device monitors
int  NP MonCreate (PSHANDLE,FARPOINTER,FARPOINTER,PERRCODE);
int  NP Register (SHANDLE,USHORT,PID,FARPOINTER,OFF,PERRCODE);
int  NP MonWrite (SHANDLE,POINTER,USHORT,USHORT,ULONG,PERRCODE);
int  NP MonFlush (SHANDLE,PERRCODE);
int  NP DeRegister (SHANDLE,PID,PERRCODE);

// 2.0  specfic
int  NP RegisterPDD(FPUCHAR,FPFUNCTION);
int  NP RegisterBeep(FPFUNCTION);
int  NP Beep(USHORT,USHORT);
int  NP FreeGDTSelector(USHORT);
int  NP PhysToGDTSel(PHYSADDR,ULONG,SEL,USHORT,FPUSHORT);
int  NP VMLock(LINADDR,ULONG,LINADDR,LINADDR,ULONG,FPULONG);
int  NP VMUnlock(LHANDLE);
int  NP VMAlloc(PLINADDR,ULONG,ULONG,PLINADDR);
int  NP VMFree(PHYSADDR);
int  NP VMProcessToGlobal(LINADDR,ULONG,ULONG,PLINADDR);
int  NP VMGlobalToProcess(LINADDR,ULONG,ULONG,PLINADDR);
int  NP VirtToLin(FARPOINTER,PLINADDR);
int  NP LinToGDTSelector(SEL,LINADDR,ULONG);
int  NP GetDescInfo(SEL,FPUSHORT,FPULONG,FPULONG);
int  NP LinToPageList(LINADDR,ULONG,LINADDR,FPULONG);
int  NP PageListToLin(ULONG,LINADDR,PLINADDR);
int  NP PageListToGDTSelector(SEL,ULONG,LINADDR,USHORT,FPUSHORT);
int  NP RegisterTmrDD(FPFUNCTION,FPFARPOINTER,FPFARPOINTER);
int  NP AllocateCtxHook(OFF,ULONG,PLHANDLE);
int  NP FreeCtxHook(LHANDLE);
int  NP ArmCtxHook(ULONG,LHANDLE,ULONG);
int  NP VMSetMem(LINADDR,ULONG,ULONG);
int  NP OpenEventSem(LHANDLE);
int  NP CloseEventSem(LHANDLE);
int  NP PostEventSem(LHANDLE);
int  NP ResetEventSem(LHANDLE,FPULONG);
int  NP DynamicAPI(FARPOINTER,USHORT,USHORT,FPUSHORT);

// these are the only API's available to the driver at Init time
#define APIENTRY far pascal

USHORT APIENTRY DosBeep(USHORT,USHORT);
USHORT APIENTRY DosCaseMap(USHORT,FARPOINTER,FARPOINTER);
USHORT APIENTRY DosChgFilePtr(SHANDLE,long,USHORT,FARPOINTER);
USHORT APIENTRY DosClose(SHANDLE);
USHORT APIENTRY DosDelete(FARPOINTER,ULONG);
USHORT APIENTRY DosDevConfig(FARPOINTER,USHORT,USHORT);
USHORT APIENTRY DosDevIOCtl(FARPOINTER,FARPOINTER,USHORT,
                USHORT, USHORT);
USHORT APIENTRY DosFindClose(SHANDLE);
USHORT APIENTRY DosFindFirst(FARPOINTER,FARPOINTER,USHORT,
                FARPOINTER,USHORT, FARPOINTER, ULONG);
USHORT APIENTRY DosFindNext(SHANDLE,FARPOINTER,USHORT,
                FARPOINTER);
USHORT APIENTRY DosGetEnv(FARPOINTER,FARPOINTER);
USHORT APIENTRY DosGetMessage(FARPOINTER,USHORT,FARPOINTER,
                USHORT,USHORT, FARPOINTER, FARPOINTER);
USHORT APIENTRY DosOpen(FARPOINTER,FARPOINTER,FARPOINTER,
                ULONG,USHORT, USHORT, USHORT, ULONG);
```

```
USHORT APIENTRY DosPutMessage(SHANDLE,USHORT,FARPOINTER);
USHORT APIENTRY DosQCurDir(USHORT,FARPOINTER,FARPOINTER);
USHORT APIENTRY DosQCurDisk(FARPOINTER,FARPOINTER);
USHORT APIENTRY DosQFileInfo(SHANDLE,USHORT,FARPOINTER,USHORT);
USHORT APIENTRY DosQFileMode(FARPOINTER,FARPOINTER,ULONG);
USHORT APIENTRY DosRead(SHANDLE,FARPOINTER,USHORT,FARPOINTER);
USHORT APIENTRY DosWrite(SHANDLE,FARPOINTER,USHORT,FARPOINTER);

// end of DRVLIB.H
```

Figure A-3. Driver library header file, 16-bit.

```
//   file drvlib32.h 32-bit version
//   This header file contains definitions intended to go along with
//   DRVLIB.LIB, a C-callable subroutine library.
//
//   This file is for OS/2 2.x

typedef unsigned char   UCHAR;
typedef unsigned short  USHORT;
typedef unsigned short  BOOLEAN;
typedef unsigned long   ULONG;
typedef UCHAR           *PUCHAR;
typedef USHORT          *PUSHORT;
typedef ULONG           *PULONG;
typedef char            *PCHAR;
typedef short           *PSHORT;
typedef long            *PLONG;
typedef void            *PVOID;
typedef PVOID           *PPVOID;

typedef USHORT          ERRCODE;     // error code returned
typedef ERRCODE         *PERRCODE;   // pointer to an error code
typedef UCHAR           FLAG;        // 8-bit flag
typedef FLAG            *PFLAG;      // pointer to 8-bit flag
typedef USHORT          SEL;         // 16-bit selector
typedef SEL             *PSEL;       // pointer to a selector
typedef USHORT          OFFSET;      // 16-bit offset
typedef USHORT          SEG;         // 16-bit segment
typedef USHORT          PID;         // Process ID
typedef USHORT          TID;         // Thread ID
typedef ULONG           PHYSADDR;    // 32-bit physical address
typedef ULONG           LINADDR;     // 32-bit li address
typedef LINADDR         *PLINADDR;   // pointer to 32 bit li address
typedef PLINADDR        *PPLINADDR;  // pointer to li address pointer
typedef PHYSADDR        *PPHYSADDR;  // pointer to 32-bit physical address
typedef char            *PSTRING;    // pointer to character string
typedef USHORT          SHANDLE;     // short (16-bit) handle
typedef SHANDLE         *PSHANDLE;   // pointer to a short handle
typedef ULONG           LHANDLE;     // long  (32-bit) handle
typedef LHANDLE         HSPINLOCK;   // spinlock handle
typedef LHANDLE         *PLHANDLE;   // pointer to a long handle
typedef HSPINLOCK       *PHSPINLOCK; // pointer to spinlock handle

//  pointers to functions

typedef int (           *PFUNCTION) ();
typedef int (  *   *PPFUNCTION) ();
```

```
// macros

#define FALSE   0
#define TRUE    1

//  pointer from selector-offset

#define MAKEP(sel, off)     ( (void  *) MAKEULONG(off, sel) )

// get selector or offset from  pointer

#define SELECTOROF(p)       ( ((USHORT  *) &(p)) [1])
#define OFFSETOF(p)         ( ((USHORT  *) &(p)) [0])

// Combine l(ow) & h(igh) to form a 32 bit quantity.

#define MAKEULONG(l, h)  ((ULONG)(((USHORT)(l)) | ((ULONG)((USHORT)(h))) << 16))
#define MAKELONG(l, h)   ((LONG)MAKEULONG(l, h))
#define MAKEBIGPFUNCTIONSETOF(p) ((ULONG) (PFUNCTIONSETOF (p)))

// Combine l(ow) & h(igh) to form a 16 bit quantity.

#define MAKEUSHORT(l, h) (((USHORT)(l)) | ((USHORT)(h)) << 8)
#define MAKESHORT(l, h)  ((SHORT)MAKEUSHORT(l, h))

// get high and low order parts of a 16 and 32 bit quantity

#define LOBYTE(w)       LOUCHAR(w)
#define HIBYTE(w)       HIUCHAR(w)
#define LOUCHAR(w)      ((UCHAR)(w))
#define HIUCHAR(w)      (((USHORT)(w) >> 8) & 0xff)
#define LOUSHORT(l)     ((USHORT)(l))
#define HIUSHORT(l)     ((USHORT)(((ULONG)(l) >> 16) & 0xffff))

//  the driver device header

typedef struct DeviceHdr
{
   struct    DeviceHdr  *DHnext; // pointer to next header, or FFFF
   USHORT    DHattribute;           // device attribute word
   PFUNCTION DHstrategy;            // offset of strategy routine
   PFUNCTION DHidc;                 // offset of IDC routine
   UCHAR     DHname[8];             // dev name (char) or #units (blk)
   char      reserved[8];
   ULONG     bit_strip;             // bit 0 DevIOCtl2, bit 1 32 bit addr
} DEVICEHDR;
typedef DEVICEHDR  *PDEVICEHDR;

//  driver device attributes word

#define DAW_CHR    0x8000           // 1=char, 0=block
#define DAW_IDC    0x4000           // 1=IDC available in this DD
#define DAW_IBM    0x2000           // 1=non-IBM block format
#define DAW_SHR    0x1000           // 1=supports shared device access
#define DAW_OPN    0x0800           // 1=open/close, or removable media
#define DAW_LEVEL1 0x0080           // level 1
#define DAW_LEVEL2 0x0100           // level 2 DosDevIOCtl2
#define DAW_LEVEL3 0x0180           // level 3 bit strip
#define DAW_GIO    0x0040           // 1=generic IOCtl supported
#define DAW_CLK    0x0008           // 1=CLOCK device
#define DAW_NUL    0x0004           // 1=NUL device
#define DAW_SCR    0x0002           // 1=STDOUT (screen)
#define DAW_KBD    0x0001           // 1=STDIN  (keyboard)
```

```
// capabilities bit strip

#define CBS_SHD    0x0001           // 1=shutdown/DevIOCtl2
#define CBS_HMEM   0x0002           // hign memory map for adapters
#define CBS_PP     0x0004           // supports parallel ports
#define CBS_ADD    0x0008           // driver is an ADD
#define CBS_INIT   0x0010           // driver receives InitComplete

// SaveMessage structure

typedef struct MessageTable
{
   USHORT      id;
   USHORT      fill_in_item;
   PVOID  item1;
   PVOID  item2;
   PVOID  item_last;
} MESSAGETABLE;

// OS/2 circular character queues

#define QUEUE_SIZE  512             //  size of queues
typedef struct CharQueue
{
   USHORT   qsize;                  // number of bytes in queue
   USHORT   qchrout;                // index of next char to put out
   USHORT   qcount;                 // number of charactes in queue
   UCHAR    qbuf[QUEUE_SIZE];
} CHARQUEUE;
typedef CHARQUEUE  *PCHARQUEUE;

// PortIO structure for SMP systems

typedef struct _PORTIO_STRUCT
{
   ULONG    port;           // port to read/write
   ULONG    data;           // data to write or returned from read
   ULONG    flags;          // flags defined below
} PORTIO_STRUCT;
typedef PORTIO_STRUCT  *PPORTIO_STRUCT;

// defines for PortIOStruct flags

#define PORTIO_READ_BYTE   0x00
#define PORTIO_READ_WORD   0x01
#define PORTIO_READ_DWORD  0x02
#define PORTIO_WRITE_BYTE  0x03
#define PORTIO_WRITE_WORD  0x04
#define PORTIO_WRITE_DWORD 0x05
#define PORTIO_FLAG_MASK   0x07

// structures and equates for APM
// APM IDC return codes

#define APMIDC_Register          0x00
#define APMIDC_DeRegister        0x01
#define APMIDC_SendEvent         0x02
#define APMIDC_QueryStatus       0x03
#define APMIDC_QueryInfo         0x04

// APM IDC return codes
```

```
#define APMIDC_InvalidFunction   0x00
#define APMIDC_InvalidHandle     0x01
#define APMIDC_InvalidDevice     0x02
#define APMIDC_InvalidEvent      0x03
#define APMIDC_InvalidOther      0x04
#define APMIDC_InterfaceBusy     0x05
#define APMIDC_RequestRejected   0x06
#define APMIDC_InvalidMask       0x07

// APM Notification Mask Definitions

#define APMMASK_EnableAPM        0x08
#define APMMASK_DisableAPM       0x10
#define APMMASK_BIOS_Defaults    0x20
#define APMMASK_SetPowerState    0x40
#define APMMASK_BatteryLow       0x80
#define APMMASK_NormalResume     0x100
#define APMMASK_CriticalResume   0x200
#define APMMASK_AllAppNotifyBits 0x3f8

// APM Event Identification

#define APMEVENT_EnableAPM       0x03
#define APMEVENT_DisableAPM      0x04
#define APMEVENT_BIOS_Defauts    0x05
#define APMEVENT_SetPowerState   0x06
#define APMEVENT_BatteryLow      0x07
#define APMEVENT_NormalResume    0x08
#define APMEVENT_CriticalResume  0x09

// Notify defines

#define APMSTATE_READY           0x00
#define APMSTATE_STANDBY         0x01
#define APMSTATE_SUSPEND         0x02
#define APMSTATE_PFUNCTION             0x03

// APM IDC Structures

typedef struct _APMIDC_REGISTER_PKT
{
    USHORT    RegFunction;
    USHORT    ReghClient;
    USHORT    RegEventHandlerOff;
    USHORT    RegEventHandlerSel;
    ULONG     RegNotifyMask;
    USHORT    RegClientDS;
    USHORT    RegDeviceID;

} APMIDC_REGISTER_PKT;

typedef struct _APMIDC_NOTIFY_PKT
{
    USHORT    NotifyFunction;
    USHORT    SubID;
    USHORT    Reserved;
    USHORT    DevID;
    USHORT    PwrState;

} APMIDC_NOTIFY_PKT;

typedef struct _APMIDC_DEREGISTER_PKT
{
    USHORT    DregFunction;
    USHORT    DreghClient;

} APMIDC_DEREGISTER_PKT;

typedef struct _APMIDC_SENDEVENT_PKT
{
    USHORT    SendevFunction;
    USHORT    SendevSubID;
    USHORT    SendevReserved;
    USHORT    SendevDevID;
    USHORT    SendevPwrState;

} APMIDC_SENDEVENT_PKT;

typedef struct _APMIDC_QSTATUS_PKT
{
    USHORT    QstatFunction;
    USHORT    QstatParmLength;
    USHORT    QstatFlags;
    UCHAR     QstatACStatus;
    UCHAR     QstateBatteryStatus;
    UCHAR     QstateBatteryLife;

} APMIDC_QSTATUS_PKT;

typedef struct _APMIDC_QINFO_PKT
{
    USHORT    QinfoFunction;
    USHORT    QinfoParmLength;
    USHORT    QinfoBIOSFlags;
    UCHAR     QinfoBIOSMajor;
    UCHAR     QinfoBIOSMinor;
    UCHAR     QinfoDDMajor;
    UCHAR     QinfoDDMinor;

} APMIDC_QINFO_PKT;

// AttachDD inter device driver communication data area

typedef struct AttachArea
{
   PFUNCTION realPFUNCTION;                      // real-mode offset of idc entry
point
   SEG realCS;                       // real-mode CS of IDC entry point
   SEG realDS;                       // real-mode DS of IDC DD
   PFUNCTION protPFUNCTION;                      // protect-mode offset of entry
point
   SEL protCS;                       // protect-mode CS of entry point
   SEL protDS;                       // protect-mode DS of other DD
} ATTACHAREA;
typedef ATTACHAREA  *PATTACHAREA;

// driver request packet

typedef struct ReqPacket
{
   UCHAR RPlength;                   // request packet length
   UCHAR   RPunit;                   // unit code for block DD only
   UCHAR RPcommand;                  // command code
   USHORT  RPstatus;                 // status word
   UCHAR   RPreserved[4];            // reserved bytes
   ULONG RPqlink;                    // queue linkage
```

```
union {                          // command-specific data
UCHAR   avail[19];
 struct {                        // init
   UCHAR      units;             // number of units
   PFUNCTION DevHlp;            // &DevHlp
   char    *args;             // &args
   UCHAR      drive;             // drive #
   }Init;
 struct {
   UCHAR     units;              // same as input
   PFUNCTION       finalCS;            // final offset, 1st code segment
   PFUNCTION       finalDS;            // final offset, 1st data segment
   PVOID BPBarray;          // &BPB
   } InitExit;

 struct {                        // read, write, write w/verify
   UCHAR      media;             // media descriptor
   PHYSADDR   buffer;            // transfer address
   USHORT     count;             // bytes/sectors
   ULONG      startsector;       // starting sector#
   USHORT     reserved;
   } ReadWrite;

 struct {                        // cached read, write, write w/verify
   UCHAR      media;             // media descriptor
   PHYSADDR   buffer;            // transfer address
   USHORT     count;             // bytes/sectors
   ULONG      startsector;       // starting sector#
   USHORT     reserved;
   } CReadWrite;

 struct {                        // system shutdown
   UCHAR      subcode;           // sub request code
   ULONG      reserved;
   } Shutdown;

 struct {                        // open/close
   USHORT     sysfilenum;        // system file number
   } OpenClose;

 struct {                        // IOCtl
   UCHAR      category;          // category code
   UCHAR      function;          // function code
   PVOID parameters;        // &parameters
   PVOID buffer;            // &buffer
   } IOCtl;

 struct {                        // read, no wait
   UCHAR      char_returned;     // char to return
   } ReadNoWait;

 struct {                        // media check
   UCHAR      media;             // media descriptor
   UCHAR      return_code;       // see #defines
   PVOID prev_volume;       // &previous volume ID
   } MediaCheck;

 struct {                        // build BPB
   UCHAR      media;             // media descriptor
   PVOID buffer;            // 1-sector buffer FAT
   PVOID BPBarray;          // &BPB array
   UCHAR      drive;             // drive #
   } BuildBPB;
```

```
    struct {                        // query partitionalble fixed disks
      UCHAR      count;             // # disks
      ULONG      reserved;
      } Partitionable;

    struct {                        // fixed disk LU map
      ULONG      units;             // units supported
      ULONG      reserved;
      } GetFixedMap;

    struct {                        // get driver capabilities
      UCHAR      reserved[3];
      PVOID capstruct;         // 16:16 pointer to DCS
      PVOID volcharstruct;     // 16:16 pointer to VCS
      } GetDriverCaps;

   } s;                             // command info
} REQPACKET;

typedef REQPACKET  *PREQPACKET;
typedef PREQPACKET  *PPREQPACKET;
typedef PREQPACKET QHEAD;           // Queue Head is &ReqPacket
typedef QHEAD  *PQHEAD;

// Global Info Seg

typedef struct _GINFOSEG
{
    ULONG   time;                // time in seconds
    ULONG   msecs;               // milliseconds
    UCHAR   hour;                // hours
    UCHAR   minutes;             // minutes
    UCHAR   seconds;             // seconds
    UCHAR   hundredths;          // hundredths
    USHORT  timezone;            // minutes from UTC
    USHORT  cusecTimerInterval;  // timter interval, .0001 secs
    UCHAR   day;                 // day of month
    UCHAR   month;               // month, 1-12
    USHORT  year;                // year
    UCHAR   weekday;             // day of week, 0=Sunday, 1=Monday...
    UCHAR   uchMajorVersion;     // major version number
    UCHAR   uchMinorVersion;     // minor version number
    UCHAR   chRevisionLetter;    // rev level
    UCHAR   sgCurrent;           // current foreground session
    UCHAR   sgMax;               // max number of sessions
    UCHAR   cHugeShift;          // shift count for huge elements
    UCHAR   fProtectModeOnly;    // protect mode only
    USHORT  pidForeground;       // pid of last process in foreground
    UCHAR   fDynamicSched;       // dynamic variation flag
    UCHAR   csecMaxWait;         // max wait in seconds
    USHORT  cmsecMinSlice;       // min timeslice in milliseconds
    USHORT  cmsecMaxSlice;       // max timeslice in milliseconds
    USHORT  bootdrive;           // boot drive (0=a, 1=b...)
    UCHAR   amecRAS[32];         // system trace major code flag bits
    UCHAR   csgWindowableVioMax; // max number of VIO sessions
    UCHAR   csgPMMax;            // max number of PM sessions
} GINFOSEG;
typedef GINFOSEG  *PGINFOSEG;

// local info seg

typedef struct _LINFOSEG
```

```
{
    PID     pidCurrent;          // current process id
    PID     pidParent;           // process id of parent
    USHORT  prtyCurrent;         // priroty of current thread
    TID     tidCurrent;          // thread id of current thread
    USHORT  sgCurrent;           // current session id
    UCHAR   rfProcStatus;        // process status
    UCHAR   dummy1;              // reserved
    USHORT  fForeground;         // current process is in foreground
    UCHAR   typeProcess;         // process type
    UCHAR   dummy2;              // reserved
    SEL     selEnvironment;      // selector of environment
    USHORT  offCmdLine;          // command line offset
    USHORT  cbDataSegment;       // length of data segment
    USHORT  cbStack;             // stack size
    USHORT  cbHeap;              // heap size
    USHORT  hmod;                // module handle of application
    SEL     selDS;               // data segment handle of application
} LINFOSEG;

typedef LINFOSEG  *PLINFOSEG;

typedef struct _REGSTACK {       // stack usgae structure
    USHORT  usStruct;            // set to 14 before using
  USHORT  usFlags;               // 0x01 means that the interrupt proc
                                 // enables interrupts. All others resvd
  USHORT  usIRQ;                 // IRQ of interrupt handler
  USHORT  usStackCLI;            // # of stack bytes with interrupts off
  USHORT  usStackSTI;            // # of stack bytes with interrupts on
  USHORT  usStackEOI;            // number of bytes needed after EOI
  USHORT  usNest;                // max number of nested levels
  } REGSTACK;

typedef REGSTACK  *PREGSTACK;

// page list struct

typedef struct _PAGELIST
{
  ULONG pl_Physaddr;
  ULONG pl_cb;
} PAGELIST;
typedef PAGELIST  *PPAGELIST;

// RPstatus bit values

#define RPERR   0x8000              //  error occurred, err in RPstatus
#define RPDEV   0x4000              //  error code defined by driver
#define RPBUSY  0x0200              //  device is busy
#define RPDONE  0x0100              //  driver done with request packet

// error codes returned in RPstatus

#define ERROR_WRITE_PROTECT         0x0000
#define ERROR_BAD_UNIT              0x0001
#define ERROR_NOT_READY             0x0002
#define ERROR_BAD_COMMAND           0x0003
#define ERROR_CRC                   0x0004
#define ERROR_BAD_LENGTH            0x0005
#define ERROR_SEEK                  0x0006
#define ERROR_NOT_DOS_DISK          0x0007
#define ERROR_SECTOR_NOT_FOUND      0x0008
#define ERROR_OUT_OF_PAPER          0x0009
```

```
#define ERROR_WRITE_FAULT           0x000A
#define ERROR_READ_FAULT            0x000B
#define ERROR_GEN_FAILURE           0x000C
#define ERROR_DISK_CHANGE           0x000D
#define ERROR_WRONG_DISK            0x000F
#define ERROR_UNCERTAIN_MEDIA       0x0010
#define ERROR_CHAR_CALL_INTERRUPTED 0x0011
#define ERROR_NO_MONITOR_SUPPORT    0x0012
#define ERROR_INVALID_PARAMETER     0x0013
#define ERROR_DEVICE_IN_USE         0x0014
#define ERROR_QUIET_FAIL            0x0015

// driver request codes  B=block, C=character

#define RPINIT          0x00        //  BC
#define RPMEDIA_CHECK   0x01        //  B
#define RPBUILD_BPB     0x02        //  B
#define RPREAD          0x04        //  BC
#define RPREAD_NO_WAIT  0x05        //   C
#define RPINPUT_STATUS  0x06        //   C
#define RPINPUT_FLUSH   0x07        //   C
#define RPWRITE         0x08        //  BC
#define RPWRITE_VERIFY  0x09        //  BC
#define RPOUTPUT_STATUS 0x0a        //   C
#define RPOUTPUT_FLUSH  0x0b        //   C
#define RPOPEN          0x0d        //  BC
#define RPCLOSE         0x0e        //  BC
#define RPREMOVABLE     0x0f        //  B
#define RPIOCTL         0x10        //  BC
#define RPRESET         0x11        //  B
#define RPGET_DRIVE_MAP 0x12        //  B
#define RPSET_DRIVE_MAP 0x13        //  B
#define RPDEINSTALL     0x14        //   C
#define RPPARTITIONABLE 0x16        //  B
#define RPGET_FIXED_MAP 0x17        //  B
#define RPSHUTDOWN      0x1c        //  BC
#define RPGET_DRIVER_CAPS 0x1d      //  B
#define RPINIT_COMPLETE 0x1f        //  BC

// check for monitor call in DosOpen/DosClose

#define MON_OPEN_STATUS   0x08      // open from DosMonOpen
#define MON_CLOSE_STATUS  0x08      // close from DosMonClose

// media descriptor byte

#define MDB_REMOVABLE     0x04      //  1=removable
#define MDB_EIGHT_SECTORS 0x02      //  1=8 sectors per track
#define MDB_DOUBLE_SIDED  0x01      //  1=double-sided media

// return codes from MediaCheck

#define MC_MEDIA_UNCHANGED 0x01
#define MC_MEDIA_CHANGED   0xFF
#define MC_MEDIA_UNSURE    0x00

// event numbers for SendEvent

#define EVENT_SM_MOUSE   0x00       // session switch via mouse
#define EVENT_CTRLBRK    0x01       // control break
#define EVENT_CTRLC      0x02       // control C
#define EVENT_CTRLNUMLK  0x03       // control num lock
#define EVENT_CTRLPRTSC  0x04       // control printscreen
```

```
#define EVENT_SHFTPRTSC  0x05       // shift printscreen
#define EVENT_SM_KBD     0x06       // session switch hot key

// defines for 1.x movedata function

#define   MOVE_PHYSTOPHYS   0       // move bytes from phys to phys memory
#define   MOVE_PHYSTOVIRT   1       // move bytes from phys to virt memory
#define   MOVE_VIRTTOPHYS   2       // move bytes from virt to phys memory
#define   MOVE_VIRTTOVIRT   3       // move bytes from virt to virt memory

// Micro Channel specific

int GetLIDEntry (USHORT, USHORT, USHORT, PUSHORT);
int FreeLIDEntry (USHORT);
int ABIOSCall (USHORT, USHORT, PVOID);
int ABIOSComm (USHORT, PVOID);
int GetDeviceBlock(USHORT, PVOID);

// special routines

void INT3  (void);
void Enable  (void);
void Disable  (void);
void Abort  (void);
int  SegLimit (SEL, PFUNCTION  *);
int  MoveBytes (PVOID,PVOID,USHORT);
int  MoveData (PVOID, PVOID, USHORT, USHORT);

// system services and misc.

int  GetDOSVar (USHORT, PPVOID);
int  SendEvent (USHORT, USHORT);
void SchedClockAddr (PPVOID);
int  AttachDD (PSTRING, PATTACHAREA);
int  InternalError(PSTRING,USHORT);
int  SaveMessage(PSTRING);
int  ProtToReal(void);
int  RealToProt(void);
int  SetROMVector(USHORT,PFUNCTION,PFUNCTION,PVOID);

// process mgmt

void Yield  (void);
void TCYield  (void);
int  Block  (ULONG, ULONG, USHORT, PVOID);
void Run  (ULONG);
void DevDone  (PREQPACKET);
int  VideoPause(USHORT);

// memory management

int  AllocPhys (ULONG, USHORT, PPHYSADDR);
int  FreePhys (PHYSADDR);
int  VerifyAccess (SEL, OFFSET, USHORT, USHORT);
int  LockSeg  (SEL, USHORT, USHORT, PLHANDLE);
int  UnLockSeg (LHANDLE);

// address conversion

int  AllocGDTSelector(USHORT, PVOID);
int  PhysToGDTSelector(PHYSADDR, USHORT, SEL, PERRCODE);
int  VirtToPhys (PVOID, PPHYSADDR);
int  PhysToUVirt (PHYSADDR, USHORT, USHORT, PVOID);
int  PhysToVirt (PHYSADDR, USHORT, USHORT, PVOID);
int  UnPhysToVirt (void);

// request packet queue stuff

int  AllocReqPacket (USHORT, PPREQPACKET);
void FreeReqPacket (PREQPACKET);
void PushReqPacket (PQHEAD, PREQPACKET);
void SortReqPacket (PQHEAD, PREQPACKET);
int  PullReqPacket (PQHEAD, PPREQPACKET);
int  PullParticular  (PQHEAD, PREQPACKET);

// driver semaphores

int  SemHandle (LHANDLE, FLAG, PLHANDLE);
int  SemRequest (LHANDLE, ULONG, PERRCODE);
void SemClear (LHANDLE);

// circular character queues

void QueueInit (PCHARQUEUE);
void QueueFlush (PCHARQUEUE);
int  QueueWrite (PCHARQUEUE, UCHAR);
int  QueueRead (PCHARQUEUE, PUCHAR);

// interrupt stuff

int  SetIRQ  (USHORT, PFUNCTION, USHORT);
int  UnSetIRQ (USHORT);
int  EOI  (USHORT);
void ClaimInterrupt(void);
void RefuseInterrupt(void);
int  RegisterStackUsage(PREGSTACK);

// timer stuff

int  SetTimer (PFUNCTION);
int  ResetTimer (PFUNCTION);
int  TickCount (PFUNCTION, USHORT);

// device monitors

int  MonCreate (PSHANDLE, PVOID, PVOID, PERRCODE);
int  Register (SHANDLE, USHORT, PID, PVOID, PFUNCTION, PERRCODE);
int  MonWrite (SHANDLE, PVOID, USHORT, USHORT, ULONG, PERRCODE);
int  MonFlush (SHANDLE, PERRCODE);
int  DeRegister (SHANDLE, PID, PERRCODE);

// 2.x  specfic

int  RegisterPDD(PUCHAR,PFUNCTION);
int  RegisterBeep(PFUNCTION);
int  Beep(USHORT,USHORT);
int  FreeGDTSelector(USHORT);
int  PhysToGDTSel(PHYSADDR,ULONG,SEL,USHORT,PUSHORT);
int  VMLock(LINADDR,ULONG,LINADDR,LINADDR,ULONG,PULONG);
int  VMUnlock(LHANDLE);
int  VMAlloc(LINADDR,ULONG,ULONG,PLINADDR);
int  VMFree(PHYSADDR);
int  VMProcessToGlobal(LINADDR,ULONG,ULONG,PLINADDR);
int  VMGlobalToProcess(LINADDR,ULONG,ULONG,PLINADDR);
int  VirtToLin(PVOID,PLINADDR);
int  LinToGDTSelector(SEL,LINADDR,ULONG);
int  GetDescInfo(SEL,PUSHORT,PULONG,PULONG);
int  LinToPageList(LINADDR,ULONG,LINADDR,PULONG);
int  PageListToLin(ULONG,LINADDR,PLINADDR);
int  PageListToGDTSelector(SEL,ULONG,LINADDR,USHORT,PUSHORT);
int  RegisterTmrDD(PFUNCTION,PPVOID,PPVOID);
int  AllocCtxHook(PFUNCTION,ULONG,PLHANDLE);
int  FreeCtxHook(LHANDLE);
int  ArmCtxHook(ULONG,LHANDLE,ULONG);
int  VMSetMem(LINADDR,ULONG,ULONG);
int  OpenEventSem(LHANDLE);
int  CloseEventSem(LHANDLE);
int  PostEventSem(LHANDLE);
int  ResetEventSem(LHANDLE,LINADDR);
int  DynamicAPI(PVOID,USHORT,USHORT,PUSHORT);

// SMP DevHlps

int  CreateSpinLock(PHSPINLOCK);
int  FreeSpinLock(HSPINLOCK);
int  AcquireSpinLock(HSPINLOCK);
int  ReleaseSpinLock(HSPINLOCK);
int  PortIO(PPORTIO_STRUCT);

int  SetIRQMask(USHORT,USHORT);
int  GetIRQMask(USHORT,PVOID);

// APM

int  APMInit(PVOID);
int  APMDeReg(PVOID);
int  NoError();
int  GetMachine(PVOID);

// these are the only API's available to the driver at Init time

#define APIENTRY

USHORT APIENTRY DosBeep(USHORT, USHORT);
USHORT APIENTRY DosCaseMap(USHORT, PVOID, PVOID);
USHORT APIENTRY DosChgFilePtr(SHANDLE, long, USHORT, PVOID);
USHORT APIENTRY DosClose(SHANDLE);
USHORT APIENTRY DosDelete(PVOID, ULONG);
USHORT APIENTRY DosDevConfig(PVOID, USHORT, USHORT);
USHORT APIENTRY DosDevIOCtl(PVOID, PVOID, USHORT, USHORT, USHORT);
USHORT APIENTRY DosFindClose(SHANDLE);
USHORT APIENTRY DosFindFirst(PVOID, PVOID, USHORT, PVOID,
        USHORT, PVOID, ULONG);
USHORT APIENTRY DosFindNext(SHANDLE, PVOID, USHORT, PVOID);
USHORT APIENTRY DosGetEnv(PVOID, PVOID);
USHORT APIENTRY DosGetMessage(PVOID, USHORT, PVOID, USHORT,
                              USHORT, PVOID, PVOID);
USHORT APIENTRY DosOpen(PVOID, PVOID, PVOID, ULONG,
                        USHORT, USHORT, USHORT, ULONG);
```

```
USHORT APIENTRY DosPutMessage(SHANDLE, USHORT, PVOID);
USHORT APIENTRY DosQCurDir(USHORT, PVOID, PVOID);
USHORT APIENTRY DosQCurDisk(PVOID, PVOID);
USHORT APIENTRY DosQFileInfo(SHANDLE, USHORT, PVOID, USHORT);
USHORT APIENTRY DosQFileMode(PVOID, PVOID, ULONG);
USHORT APIENTRY DosRead(SHANDLE, PVOID, USHORT, PVOID);
USHORT APIENTRY DosWrite(SHANDLE, PVOID, USHORT, PVOID);
USHORT APIENTRY DosCreateSpinLock(PHSPINLOCK);
USHORT APIENTRY DosFreeSpinLock(HSPINLOCK);

// end of DRVLIB.H
```

Figure A-4. Driver library header file, 32-bit.

```
#include "drvlib.h"
#include "digio.h"

int main(PREQPACKET rp);

DEVICEHDR devhdr =
{
  (void  *) 0xFFFFFFFF,     /* link                          */
  (DAW_CHR | DAW_OPN | DAW_LEVEL1),/* attribute word          */
  &main,                    /* &strategy                     */
  0,                        /* &IDC routine                  */
  "DIGIO$  "                /* name/#units                   */
};

PFUNCTION   DevHlp=0;       /* pointer to DevHlp entry point  */
UCHAR       opencount = 0;  /* keeps track of open's          */
USHORT      savepid=0;      /* save thread pid                */
LHANDLE     lock_seg_han;   /* handle for locking appl. seg   */
PHYSADDR    appl_buffer=0;  /* address of caller's buffer     */
ERRCODE     err=0;          /* error return                   */
ULONG       ReadID=0L;      /* current read pointer           */
USHORT      num_rupts=0;    /* count of interrupts            */
USHORT      temp_char;      /* temp character for in-out      */
void         *ptr;          /* temp  pointer                  */
PVOID       appl_ptr=0;     /* pointer to application buffer  */
char        input_char,output_char; /* temp character storage */
char        input_mask;     /* mask for input byte            */

/* messages */
char     CrLf[]= "\r\n";
char     InitMessage1[] = " 8 bit Digital I/O ";
char     InitMessage2[] = " driver installed\r\n";
char     FailMessage[]  = " driver failed to install.\r\n";

/* common entry point for calls to Strategy routines */
int main(PREQPACKET rp)
{
  void  *ptr;
  PLINFOSEG liptr;        /* pointer to global info seg  */
  int i;

  switch(rp->RPcommand)
  {
    case RPINIT:          /* 0x00                        */
      /* init called by kernel in protected mode */
      return Init(rp);

    case RPREAD:                 /* 0x04                      */
      rp->s.ReadWrite.count = 0; /* in case we fail           */
      input_char = inp(DIGIO_INPUT);/* get data               */
      if (PhysToVirt( (ULONG) rp->s.ReadWrite.buffer,
                       1,0,&appl_ptr))
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      if (MoveBytes((PVOID)&input_char,appl_ptr,1))
             return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      rp->s.ReadWrite.count = 1;  /* one byte read            */
      return (RPDONE);

    case RPWRITE:                /* 0x08                       */
      rp->s.ReadWrite.count = 0;
      if (PhysToVirt( (ULONG) rp->s.ReadWrite.buffer,
           1,0,&appl_ptr))
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      if (MoveBytes(appl_ptr,(PVOID)&output_char,1))
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      outp (DIGIO_OUTPUT,output_char); /* send byte          */
      rp->s.ReadWrite.count = 1; /* one byte written         */
      return (RPDONE);

    case RPOPEN:                 /* 0x0d open driver           */
      /* get current process id */
      if (GetDOSVar(2,&ptr))
        return (RPDONE | RPERR | ERROR_BAD_COMMAND);
      /* get process info */
      liptr = *((PLINFOSEG  *) ptr);
      /* if this device never opened, can be opened by anyone*/
      if (opencount == 0)      /* first time this dev opened */
      {
        opencount=1;         /* bump open counter          */
        savepid = liptr->pidCurrent; /* save current PID   */
      }
      else
      {
       if (savepid != liptr->pidCurrent) /* another proc  */
         return (RPDONE | RPERR | ERROR_NOT_READY);/*err*/
       ++opencount;         /* bump counter, same pid     */
      }
      return (RPDONE);

    case RPCLOSE:                /* 0x0e DosClose,ctl-C, kill  */
      /* get process info of caller */
      if (GetDOSVar(2,&ptr))
        return (RPDONE | RPERR | ERROR_BAD_COMMAND);
      /* get process info from os/2 */
      liptr= *((PLINFOSEG  *) ptr); /* ptr to linfoseg    */
      /*
       make sure that process attempting to close this device
       is the one that originally opened it and the device was
       open in the first place.
      */
      if (savepid != liptr->pidCurrent || opencount == 0)
        return (RPDONE | RPERR | ERROR_BAD_COMMAND);
      --opencount;             /* close counts down open cntr*/
      return (RPDONE);        /* return 'done' status       */

    case RPIOCTL:                /* 0x10                       */
      /*
```

```
 The function code in an IOCtl packet has the high bit set
 for the DIGIO$ board. We return all others with the done
 bit set so we don't have to handle things like the 5-48
 code page IOCtl
*/
if (rp->s.IOCtl.category != DIGIO_CAT)/* other IOCtls  */
  return (RPDONE | RPERR | ERROR_BAD_COMMAND);
switch (rp->s.IOCtl.function)
{
  case 0x01:               /* write byte to digio port   */
    /* verify caller owns this buffer area */
    if(VerifyAccess(
      SELECTOROF(rp->s.IOCtl.parameters), /* selector    */
      OFFSETOF(rp->s.IOCtl.parameters),   /* offset      */
      1,                                  /* 1 byte      */
      0)}                                 /* read only   */
      return (RPDONE | RPERR | ERROR_GEN_FAILURE);
    if(MoveBytes(rp->s.IOCtl.parameters,(PVOID)
               &output_char,1))
      return (RPDONE | RPERR | ERROR_GEN_FAILURE);
    outp(DIGIO_OUTPUT,output_char);     /*send to digio*/
    return (RPDONE);

  case 0x02:               /* read byte w/wait from port */
    /* verify caller owns this buffer area */
    if(VerifyAccess{
      SELECTOROF(rp->s.IOCtl.buffer), /* selector        */
      OFFSETOF(rp->s.IOCtl.buffer),   /* offset          */
      1,                              /* 1 bytes)        */
      0))                             /* read only       */
      return (RPDONE | RPERR | ERROR_GEN_FAILURE);

    /* lock the segment down temp */
    if(LockSeg(
      SELECTOROF(rp->s.IOCtl.buffer), /* selector        */
      1,                              /* lock forever    */
      0,                              /* wait for seg loc*/
      (PLHANDLE) &lock_seg_han))      /* handle returned */
         return (RPDONE | RPERR | ERROR_GEN_FAILURE);
    if(MoveBytes(rp->s.IOCtl.parameters,(PVOID)
                &input_mask,1))
      return (RPDONE | RPERR | ERROR_GEN_FAILURE);

    /* wait for switch to be pressed */
    ReadID = (ULONG)rp;             /* block ID        */
    if (Block(ReadID,-1L,0,&err))
        if (err == 2)
          return(RPDONE | RPERR
                | ERROR_CHAR_CALL_INTERRUPTED);
    /* move data to users buffer */
    if(MoveBytes((PVOID)&input_char,rp->s.IOCtl.buffer,
                1))
      return(RPDONE | RPERR | ERROR_GEN_FAILURE);

    /* unlock segment */
    if(UnLockSeg(lock_seg_han))
      return(RPDONE | RPERR | ERROR_GEN_FAILURE);
    return (RPDONE);

  case 0x03:               /* read byte immed digio port */
    /* verify caller owns this buffer area */
    if(VerifyAccess(
      SELECTOROF(rp->s.IOCtl.buffer), /* selector        */
```

```
            OFFSETOF(rp->s.IOCtl.buffer),   /* offset          */
            4,                              /* 4 bytes         */
            0))                             /* read only       */
             return (RPDONE | RPERR | ERROR_GEN_FAILURE);

          input_char = inp(DIGIO_INPUT);  /* get data        */
          if(MoveBytes((PVOID)&input_char,rp->s.IOCtl.buffer,1))
            return(RPDONE | RPERR | ERROR_GEN_FAILURE);

          return (RPDONE);

        default:
            return(RPDONE | RPERR | ERROR_GEN_FAILURE);
        }

    /* don't allow deinstall */
    case RPDEINSTALL:            /* 0x14                       */
        return(RPDONE | RPERR | ERROR_BAD_COMMAND);

    /* all other commands are flagged as bad */
    default:
        return(RPDONE | RPERR | ERROR_BAD_COMMAND);
    }
}

timr_handler()
{
  if (ReadID != 0) {
    /* read data from port */
    input_char = inp(DIGIO_INPUT );/* get data               */
    if ((input_char && input_mask) !=0) {
      Run (ReadID);
      ReadID=0L;
      }
  }
}

/* Device Initialization Routine */

int Init(PREQPACKET rp)
{
  /* store DevHlp entry point */
  DevHlp = rp->s.Init.DevHlp;
  /* install timer handler */
  if(SetTimer((PFUNCTION)timr_handler)) {
    /* if we failed, effectively deinstall driver with cs+ds=0 */
    DosPutMessage(1, 8, devhdr.DHname);
    DosPutMessage(1,strlen(FailMessage),FailMessage);
    rp->s.InitExit.finalCS = (OFFSET) 0;
    rp->s.InitExit.finalDS = (OFFSET) 0;
    return (RPDONE | RPERR | ERROR_GEN_FAILURE);
    }
  /* configure 8255 parallel chip */
  outp (DIGIO_CONFIG,0x91);
  /* output initialization message */
  DosPutMessage(1, 2, CrLf);
  DosPutMessage(1, 8, devhdr.DHname);
  DosPutMessage(1, strlen(InitMessage1), InitMessage1);
  DosPutMessage(1, strlen(InitMessage2), InitMessage2);

  /* send back our code and data end values to os/2 */
  if (SegLimit(HIUSHORT((void  *) Init),
      &rp->s.InitExit.finalCS) || SegLimit(HIUSHORT((void  *)
      InitMessage2), &rp->s.InitExit.finalDS))
        Abort();
    return(RPDONE);
}
```

Figure A-5. Simple OS/2 parallel device driver, 32-bit version.

```
#include "drvlib.h"
#include "mmap.h"

extern void near STRATEGY();        // name of strat rout. in DDSTART

DEVICEHDR devhdr = {
  (void far *) 0xFFFFFFFF,    // link
  (DAW_CHR | DAW_OPN | DAW_LEVEL1),// attribute
  (OFF) STRATEGY,             // &strategy
  (OFF) 0,                    // &IDCroutine
  "MMAP$   "
};

FPFUNCTION  DevHlp=0;         // storage area for DevHlp calls
LHANDLE     lock_seg_han;     // handle for locking appl. segment
PHYSADDR    appl_buffer=0;    // address of caller's buffer
PREQPACKET  p=0L;             // pointer to request packet
ERRCODE     err=0;            // error return
void        far *ptr;         // temp far pointer
USHORT      i,j;              // general counters
PHYSADDR    board_address;    // base board address
USHORT      opencount;        // count of DosOpens
USHORT      savepid;          // save the caller's PID
USHORT      cntr = 0;         // misc counter
USHORT      bus = 0;          // default ISA bus
REQBLK      ABIOS_r_blk;      // ABIOS request block
LIDBLK      ABIOS_l_blk;      // ABIOS LID block
USHORT      lid_blk_size;     // size of LID block
CARD        card[MAX_NUM_SLOTS+1]; // array for IDs
CARD        *pcard;           // pointer to card array
USHORT      matches = 0;      // match flag for card ID
POS_STRUCT  pos_struct;       // struct to get POS reg
ADDR_STRUCT addr_struct;      // struct for passing addresses
USHORT      chunk1,chunk2;    // temp variables for address calc
char     arguments[64]={0};   // save command line args in dgroup
char     NoMatchMsg[]  = " no match for selected Micro Channel
                           card ID found.\r\n";
char     MainMsgMCA[]  = "\r\nOS/2 Micro Channel memory-mapped
                          driver installed.\r\n";
char     MainMsgISA[]  = "\r\nOS/2 ISA bus memory-mapped driver
                          installed.\r\n";
```

```
// prototypes
int      hex2bin(char c);
USHORT   get_POS();
UCHAR    get_pos_data();
UCHAR    nget_pos_data();

// common entry point for calls to Strategy routines

int main(PREQPACKET rp )
{
  void far *ptr;
  int far *pptr;
  PLINFOSEG liptr;                // pointer to local info seg
  int i;
  ULONG addr;
  USHORT in_data;

  switch(rp->RPcommand)
  {
    case RPINIT:                    // 0x00
      // init called by kernel in protected mode ring 3 with IOPL
      return Init(rp);

    case RPOPEN:                    // 0x0d
      // get current processes id
      if (GetDOSVar(2,&ptr))
        return (RPDONE | RPERR | ERROR_BAD_COMMAND);
      // get process info
      liptr = *((PLINFOSEG far *) ptr);
      // if this device never opened, can be opened by any process
      if (opencount == 0)    // first time this device opened
      {
        opencount=1;                 // set open counter
        savepid = liptr->pidCurrent; // save current process id
      }
      else
      {
        if (savepid != liptr->pidCurrent) // proc tried to open
          return (RPDONE | RPERR | RPBUSY ); // so return error
        ++opencount;            // bump counter, same pid
      }
      return (RPDONE);

    case RPCLOSE:                   // 0x0e
      // get process info of caller
      if (GetDOSVar(2,&ptr))
        return (RPDONE | RPERR | ERROR_BAD_COMMAND); // no info
      // get process info from os/2
      liptr= *((PLINFOSEG far *) ptr); // ptr to process info seg
      //
      // make sure that process attempting to close this device
      // one that originally opened it and the device was open in
      // first place.
      //
      if (savepid != liptr->pidCurrent || opencount == 0)
        return (RPDONE | RPERR | ERROR_BAD_COMMAND);
      // if an LDT selector was allocated, free it
      PhysToUVirt(board_address,0x8000,2,
                 (FARPOINTER)&addr_struct.mapped_addr);
      --opencount;             // close counts down open counter
      return (RPDONE);         // return 'done' status to caller
```

```
case RPREAD:                   // 0x04
  return(RPDONE);

case RPWRITE:                  // 0x08
    return (RPDONE);

case RPIOCTL:                  // 0x10
  if (rp->s.IOCtl.category != OUR_CAT) // only our category
    return (RPDONE);

  switch (rp->s.IOCtl.function)
  {
    // this IOCtl returns the bus type. If Micro Channel
     // the return is 0xff01. If ISA, the return is ff00

    case 0x01:                 // check if MCA or ISA
      return (RPDONE | RPERR | bus);

     // this IOCtl maps adapter mem to an LDT selector:offset,
     // and sends it to the application for direct reads
   // and writes

   case 0x02:                  // send memmapped addr to app
     // verify caller owns this buffer area
     if(VerifyAccess(
        SELECTOROF(rp->s.IOCtl.buffer), // selector
        OFFSETOF(rp->s.IOCtl.buffer),   // offset
        8,                              // 8 bytes
        1) )                            // read write
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);
     // lock the segment down temp
     if(LockSeg(
        SELECTOROF(rp->s.IOCtl.buffer), // selector
        0,                              // lock < 2 sec
        0,                              // wait for seg lock
        (PLHANDLE) &lock_seg_han))      // handle returned
            return (RPDONE | RPERR | ERROR_GEN_FAILURE);

     // map the board address to an LDT entry

     if ( PhysToUVirt(board_address,0x8000,1,
            (FARPOINTER)&addr_struct.mapped_addr))
       return (RPDONE | RPERR | ERROR_GEN_FAILURE);

     // move data to users buffer
     if(MoveBytes(
       &addr_struct,                   // source
       rp->s.IOCtl.buffer,             // dest
       8))                             // 8 bytes
       return (RPDONE | RPERR | ERROR_GEN_FAILURE);

     // unlock segment
     if(UnLockSeg(lock_seg_han))
       return(RPDONE | RPERR | ERROR_GEN_FAILURE);
      return (RPDONE);

    // this IOCtl demonstrates how an application gets the
     // contents of a Micro Channel Adapter's POS registers

    case 0x03:                 // get pos reg data
      // verify caller owns this buffer area
      if(VerifyAccess(
        SELECTOROF(rp->s.IOCtl.buffer), // selector
        OFFSETOF(rp->s.IOCtl.buffer),   // offset
        6,                              // 6 bytes
        1) )                            // read write
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);

      // lock the segment down temp
      if(LockSeg(
        SELECTOROF(rp->s.IOCtl.buffer), // selector
        0,                              // lock < 2 sec
        0,                              // wait for seg lock
        (PLHANDLE) &lock_seg_han))      // handle returned
            return (RPDONE | RPERR | ERROR_GEN_FAILURE);

        // move slot data to driver buffer
      if(MoveBytes(
        (FARPOINTER) appl_buffer,       // source
        &pos_struct,                    // for pos data
        6))                             // 6 bytes
            return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      pos_struct.data =
            get_pos_data(pos_struct.slot,pos_struct.reg);
      // move POS reg data to users buffer
      if(MoveBytes(
        &pos_struct,                    // for pos data
        (FARPOINTER) appl_buffer,       // source
        6))                             // 6 bytes
         return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      // unlock segment
      if(UnLockSeg(lock_seg_han))
        return(RPDONE | RPERR | ERROR_GEN_FAILURE);
      return (RPDONE);

     // this IOCtl is the same as 0x02, except the
     // user virtual address is mapped to a lin address in the
     // process address range and then sent to the app. This
     // saves the SelToFlat and FlatToSel each time the ptr is
     // referenced.

    case 0x04:         // 32-bit memory-mapped addr to app
      // verify caller owns this buffer area
      if(VerifyAccess(
        SELECTOROF(rp->s.IOCtl.buffer), // selector
        OFFSETOF(rp->s.IOCtl.buffer),   // offset
        8,                              // 8 bytes
        1) )                            // read write
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      // lock the segment down temp
      if(LockSeg(
        SELECTOROF(rp->s.IOCtl.buffer), // selector
        0,                              // lock < 2 sec
        0,                              // wait for seg lock
        (PLHANDLE) &lock_seg_han))      // handle returned
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      // map the board address to an LDT entry
      if ( PhysToUVirt(board_address,0x8000,1,
               (FARPOINTER) &addr_struct.mapped_addr))
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      // now convert it to a linear address
      if (VirtToLin((FARPOINTER)addr_struct.mapped_addr,
                (PLINADDR)&addr_struct.mapped_addr))
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);
      // move data to users buffer
      if(MoveBytes(
            &addr_struct,                   // source
            rp->s.IOCtl.buffer,             // dest
            8))                             // 8 bytes
            return (RPDONE | RPERR | ERROR_GEN_FAILURE);

          // unlock segment
          if(UnLockSeg(lock_seg_han))
            return(RPDONE | RPERR | ERROR_GEN_FAILURE);
          return (RPDONE);
      } // switch (rp->s.IOCtl.function

    case RPDEINSTALL:               // 0x14
      return(RPDONE | RPERR | ERROR_BAD_COMMAND);

   // all other commands are ignored
    default:
      return(RPDONE);
   }
}

int  hex2bin(char c)
{
 if(c < 0x3a)
  return (c - 48);
 else
  return (( c & 0xdf) - 55);
}

USHORT get_POS(USHORT slot_num,
               USHORT far *card_ID,
               UCHAR far *pos_regs)
{
USHORT rc, i, lid;

  if (GetLIDEntry(0x10, 0, 1, &lid)) // get LID for POS
    return (1);

  // Get the size of the LID request block
  ABIOS_l_blk.f_parms.req_blk_len = sizeof(struct lid_block_def);
  ABIOS_l_blk.f_parms.LID = lid;
  ABIOS_l_blk.f_parms.unit = 0;;
  ABIOS_l_blk.f_parms.function = GET_LID_BLOCK_SIZE;
  ABIOS_l_blk.f_parms.ret_code = 0x5a5a;
  ABIOS_l_blk.f_parms.time_out = 0;
  if (ABIOSCall(lid,0,(void far *)&ABIOS_l_blk))
    return (1);
  lid_blk_size = ABIOS_l_blk.s_parms.blk_size; // Get block size
  // Fill POS regs and card ID with FF in case this does not work
  *card_ID = 0xFFFF;
  for (i=0; i<NUM_POS_BYTES; i++) { pos_regs[i] = 0x00; };
  // Get the POS registers and card ID for the commanded slot
  ABIOS_r_blk.f_parms.req_blk_len = lid_blk_size;
  ABIOS_r_blk.f_parms.LID = lid;
  ABIOS_r_blk.f_parms.unit = 0;;
  ABIOS_r_blk.f_parms.function = READ_POS_REGS_CARD;
  ABIOS_r_blk.f_parms.ret_code = 0x5a5a;
  ABIOS_r_blk.f_parms.time_out = 0;
  ABIOS_r_blk.s_parms.slot_num = (UCHAR)slot_num & 0x0F;
  ABIOS_r_blk.s_parms.pos_buf = (void far *)pos_regs;
  ABIOS_r_blk.s_parms.card_ID = 0xFFFF;
  if (ABIOSCall(lid,0,(void far *)&ABIOS_r_blk))
    rc = 1;
  else
  {                                     // Else
   *card_ID = ABIOS_r_blk.s_parms.card_ID;   // Set card ID value
    rc = 0;
  }
  FreeLIDEntry(lid);
  return(rc);
}

UCHAR get_pos_data (int slot, int reg)
{
  UCHAR pos;
  CARD *cptr;

  cptr = &card[slot-1];            // set ptr to card array
  if (reg == 0)                    // card ID
     pos = LOUSHORT(cptr->card_ID);
  else
    if ( reg == 1)
      pos = HIUSHORT(cptr->card_ID);
  else
     pos = cptr->pos_regs[reg-2];  // POS data register
  return (pos);
}

// Device Initialization Routine

int Init(PREQPACKET rp)
{
  USHORT lid;
  register char far *p;

  // store DevHlp entry point
  DevHlp = rp->s.Init.DevHlp;  // save DevHlp entry point
  if (!(GetLIDEntry(0x10, 0, 1, &lid)))
  {                            // get LID for POS
     FreeLIDEntry(lid);
     // Micro Channel (tm) setup section
     bus = 1;                      // MCA bus
     //    Get POS data and card ID for each of 8 possible slots
     for (i=0;i <= MAX_NUM_SLOTS; i++)
        get_POS(i+1,(FARPOINTER)&card[i].card_ID,
                    (FARPOINTER)card[i].pos_regs);
     matches = 0;
     for (i=0, pcard = card; i <= MAX_NUM_SLOTS; i++, pcard++)
     {
       if (pcard->card_ID == TARGET_ID)
       {
         matches = 1;
         break;
       }
     }
     if (matches == 0) {           // at least one board found
       DosPutMessage(1, 8, devhdr.DHname);
       DosPutMessage(1,strlen(NoMatchMsg),NoMatchMsg);
       rp->s.InitExit.finalCS = (OFF) 0;
       rp->s.InitExit.finalDS = (OFF) 0;
        return (RPDONE | RPERR | ERROR_BAD_COMMAND);
       }
     // calculate the board address from the POS regs
     board_address = ((unsigned long) get_pos_data(i+1, 4)
        << 16) |((unsigned long)(get_pos_data(i+1, 3) & 1)
        << 15);
  }
  else
  // ISA bus setup
  {
    bus = 0;                      // ISA bus

    // get parameters, port addr and base mem addr
    for (p = rp->s.Init.args; *p && *p != ' ';++p);
    for (; *p == ' '; ++p); // skip blanks following name
    if (*p)
    {
      board_address=0;           // i/o port address
      for (; *p != '\0'; ++p)    // get board address
      board_address = (board_address << 4) + (hex2bin(*p));
      addr_struct.board_addr = board_address;
    }
  }
  if (bus)
     DosPutMessage(1,strlen(MainMsgMCA),MainMsgMCA);
  else
     DosPutMessage(1,strlen(MainMsgISA),MainMsgISA);

  // send back our cs and ds end values to os/2
  if (SegLimit(HIUSHORT((void far *) Init),
              &rp->s.InitExit.finalCS) ||
               SegLimit(HIUSHORT((void far *) MainMsgISA),
               &rp->s.InitExit.finalDS))
     Abort();
  Beep(200,500);
  Beep(200,500);
  Beep(250,500);
  Beep(300,500);
  Beep(250,500);
  Beep(300,500);
  return (RPDONE);
}
```

Figure A-6. Memory-mapped device driver, 16-bit version.

```
#include <stdio.h>
#include <string.h>
//
//   OS/2 Device Driver for memory mapped I/O
//
//                Steve Mastrianni
//                15 Great Oak Lane
//                Unionville, CT 06085
//                (203) 693-0404 voice
//                (203) 693-9042 data
//                CI$ 71501,1652
//                BIX smastrianni
//                Internet 6099225@mcimail.com
//
//   This driver is loaded in the config.sys file with the DEVICE=
//   statement. For ISA configuration, the first parameter to the "DEVICE="
//   is the board base memory address in hex.
//
//   This driver also returns a boolean to the calling application to
//   inform it of the bus type (Micro Channel or ISA).
```

```
//
//   All numbers are in hex. For MCA configuration, the board address
//   is read from the board POS regs. The POS regs data is specific for
//   each adapter, so the address calculations here may not work with
//   your specific adapter. Refer to the hardware tech reference for the
//   particular adapter to determine where and how the address appears
//   in the POS registers.
//
//
//   This driver allows the application I/O to run in Ring 2 with IOPL.
//   The CONFIG.SYS files *must* contain the IOPL=YES statement.
//
//   This driver supports 4 IOCtls, Category 0x90.
//
//   IOCtl 0x01 test for MCA or ISA bus
//   IOCtl 0x02 gets and returns a selector to fabricated board memory
//   IOCtl 0x03 gets the value of a selected POS register
//   IOCtl 0x04 gets the board address that the driver found
//
//   The driver is made by using the make file mmap.mak.

#include "drvlib.h"
#include "mmap.h"

extern void  STRATEGY();        // name of strat rout. in DDSTART

DEVICEHDR devhdr = {
        (void  *) 0xFFFFFFFF,    // link
        (DAW_CHR | DAW_OPN | DAW_LEVEL1),// attribute
        (OFF) STRATEGY,              // &strategy
        (OFF) 0,                     // &IDCroutine
        "MMAP$   "
};

FPFUNCTION  DevHlp=0;                // storage area for DevHlp calls
LHANDLE     lock_seg_han;            // handle for locking appl. segment
PHYSADDR    appl_buffer=0;           // address of caller's buffer
PREQPACKET  p=0L;                    // pointer to request packet
ERRCODE     err=0;                   // error return
void         *ptr;                // temp  pointer
USHORT      i,j;                     // general counters
PHYSADDR    board_address;           // base board address
USHORT      opencount;               // count of DosOpens
USHORT      savepid;                 // save the caller's PID
USHORT      cntr = 0;                // misc counter
USHORT      bus = 0;                 // default ISA bus
REQBLK      ABIOS_r_blk;             // ABIOS request block
LIDBLK      ABIOS_l_blk;             // ABIOS LID block
USHORT      lid_blk_size;            // size of LID block
CARD        card[MAX_NUM_SLOTS+1];  // array for IDs and POS reg values
CARD        *pcard;                  // pointer to card array
USHORT      matches = 0;             // match flag for card ID
POS_STRUCT  pos_struct;              // struct to get POS reg
ADDR_STRUCT addr_struct;             // struct for passing addresses
USHORT      chunk1,chunk2;           // temp variables for address calc

char     arguments[64]={0};          // save command line args in dgroup
char     NoMatchMsg[]  = " no match for selected Micro Channel card ID
found.\r\n";
char     MainMsgMCA[]  = "\r\nOS/2 Micro Channel memory-mapped driver
installed.\r\n";
char     MainMsgISA[]  = "\r\nOS/2 ISA bus memory-mapped driver installed.\r\n";
```

```
// prototypes

int      hex2bin(char c);
USHORT   get_POS();
UCHAR    get_pos_data();
UCHAR    nget_pos_data();

// common entry point for calls to Strategy routines

int main(PREQPACKET rp )
{
    void  *ptr;
    int  *pptr;
    PLINFOSEG liptr;                // pointer to local info seg
    int i;
    ULONG addr;
    USHORT in_data;

    switch(rp->RPcommand)
    {
    case RPINIT:                    // 0x00

        // init called by kernel in protected mode ring 3 with IOPL

        return Init(rp);

    case RPOPEN:                    // 0x0d

        // get current processes id

        if (GetDOSVar(2,&ptr))
            return (RPDONE | RPERR | ERROR_BAD_COMMAND);

        // get process info

        liptr = *((PLINFOSEG  *) ptr);

        // if this device never opened, can be opened by any process

        if (opencount == 0)    // first time this device opened
        {
            opencount=1;                 // set open counter
            savepid = liptr->pidCurrent; // save current process id
        }
        else
            {
            if (savepid != liptr->pidCurrent) // another proc tried to open
                return (RPDONE | RPERR | RPBUSY ); // so return error
            ++opencount;            // bump counter, same pid
        }
        return (RPDONE);

    case RPCLOSE:                   // 0x0e

        // get process info of caller

        if (GetDOSVar(2,&ptr))
            return (RPDONE | RPERR | ERROR_BAD_COMMAND); // no info

        // get process info from os/2

        liptr= *((PLINFOSEG  *) ptr); // ptr to process info seg
        //
        // make sure that process attempting to close this device
        // one that originally opened it and the device was open in
        // first place.
        //

        if (savepid != liptr->pidCurrent || opencount == 0)
            return (RPDONE | RPERR | ERROR_BAD_COMMAND);

        // if an LDT selector was allocated, free it

        PhysToUVirt(board_address,0x8000,2,
                   (FARPOINTER)&addr_struct.mapped_addr);

        --opencount;                // close counts down open counter
        return (RPDONE);            // return 'done' status to caller

    case RPREAD:                    // 0x04

        return(RPDONE);

    case RPWRITE:                   // 0x08

        return (RPDONE);

    case RPIOCTL:                   // 0x10

        if (rp->s.IOCtl.category != OUR_CAT) // only our category
            return (RPDONE);

        switch (rp->s.IOCtl.function)
        {

                 // this IOCtl returns the bus type. If the type is Micro Channel
                 // the return is 0xff01. If ISA, the return is ff00

        case 0x01:                  // check if MCA or ISA
            return (RPDONE | RPERR | bus);

                 // this IOCtl maps an adapter memory to an LDT selector:offset,
                 // and sends it to the application for direct application reads
                 // and writes

        case 0x02:                  // send memory-mapped addr to app

            // verify caller owns this buffer area

            if(VerifyAccess(
            SELECTOROF(rp->s.IOCtl.buffer), // selector
            OFFSETOF(rp->s.IOCtl.buffer),   // offset
            8,                              // 8 bytes
            1) )                            // read write
                return (RPDONE | RPERR | ERROR_GEN_FAILURE);

            // lock the segment down temp

            if(LockSeg(
            SELECTOROF(rp->s.IOCtl.buffer), // selector
            0,                              // lock < 2 sec
            0,                              // wait for seg lock
            (PLHANDLE) &lock_seg_han))      // handle returned
                return (RPDONE | RPERR | ERROR_GEN_FAILURE);
```

```
   // map the board address to an LDT entry

   if ( PhysToUVirt(board_address,0x8000,1,
                (FARPOINTER)&addr_struct.mapped_addr))
       return (RPDONE | RPERR | ERROR_GEN_FAILURE);

   // move data to users buffer

   if(MoveBytes(
   &addr_struct,                    // source
   rp->s.IOCtl.buffer,              // dest
   8))                              // 8 bytes
       return (RPDONE | RPERR | ERROR_GEN_FAILURE);

   // unlock segment

   if(UnLockSeg(lock_seg_han))
       return(RPDONE | RPERR | ERROR_GEN_FAILURE);

   return (RPDONE);

         // this IOCtl demonstrates how an application program can get the
         // contents of a Micro Channel Adapter's POS registers

case 0x03:                  // get pos reg data

    // verify caller owns this buffer area

    if(VerifyAccess(
    SELECTOROF(rp->s.IOCtl.buffer), // selector
    OFFSETOF(rp->s.IOCtl.buffer),   // offset
    6,                              // 6 bytes
    1) )                            // read write
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);

    // lock the segment down temp

    if(LockSeg(
    SELECTOROF(rp->s.IOCtl.buffer), // selector
    0,                              // lock < 2 sec
    0,                              // wait for seg lock
    (PLHANDLE) &lock_seg_han))      // handle returned
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);

    // move slot data to driver buffer

    if(MoveBytes(
    (FARPOINTER) appl_buffer,       // source
    &pos_struct,                    // for pos data
    6))                             // 6 bytes
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);

    pos_struct.data = get_pos_data(pos_struct.slot,pos_struct.reg);

    // move POS reg data to users buffer

    if(MoveBytes(
    &pos_struct,                    // for pos data
    (FARPOINTER) appl_buffer,       // source
    6))                             // 6 bytes
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);

    // unlock segment
    if(UnLockSeg(lock_seg_han))

        return(RPDONE | RPERR | ERROR_GEN_FAILURE);

    return (RPDONE);

         // this IOCtl is essentially the same as 0x02, except the
         // user virtual address is mapped to a li address in the
         // process address range and then sent to the application. This
         // save the SelToFlat and FlatToSel each time the pointer is
         // referenced.

case 0x04:                 // 32-bit memory-mapped addr to app

    // verify caller owns this buffer area

    if(VerifyAccess(
    SELECTOROF(rp->s.IOCtl.buffer), // selector
    OFFSETOF(rp->s.IOCtl.buffer),   // offset
    8,                              // 8 bytes
    1) )                            // read write
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);

    // lock the segment down temp

    if(LockSeg(
    SELECTOROF(rp->s.IOCtl.buffer), // selector
    0,                              // lock < 2 sec
    0,                              // wait for seg lock
    (PLHANDLE) &lock_seg_han))      // handle returned
        return (RPDONE | RPERR | ERROR_GEN_FAILURE);

   // map the board address to an LDT entry

   if ( PhysToUVirt(board_address,0x8000,1,
                (FARPOINTER) &addr_struct.mapped_addr))
       return (RPDONE | RPERR | ERROR_GEN_FAILURE);

             // now convert it to a li address

             if (VirtToLin((FARPOINTER)addr_struct.mapped_addr,

(PLINADDR)&addr_struct.mapped_addr))
       return (RPDONE | RPERR | ERROR_GEN_FAILURE);

   // move data to users buffer

   if(MoveBytes(
   &addr_struct,                   // source
   rp->s.IOCtl.buffer,             // dest
   8))                             // 8 bytes
       return (RPDONE | RPERR | ERROR_GEN_FAILURE);

   // unlock segment

   if(UnLockSeg(lock_seg_han))
       return(RPDONE | RPERR | ERROR_GEN_FAILURE);

   return (RPDONE);

} // switch (rp->s.IOCtl.function
    case RPDEINSTALL:               // 0x14

        return(RPDONE | RPERR | ERROR_BAD_COMMAND);

        // all other commands are ignored

    default:
        return(RPDONE);

    }
}

int  hex2bin(char c)
{
 if(c < 0x3a)
  return (c - 48);
 else
  return (( c & 0xdf) - 55);
}

USHORT get_POS(USHORT slot_num,USHORT  *card_ID,UCHAR  *pos_regs)
{
USHORT rc, i, lid;

    if (GetLIDEntry(0x10, 0, 1, &lid)) // get LID for POS
        return (1);

    // Get the size of the LID request block

    ABIOS_l_blk.f_parms.req_blk_len = sizeof(struct lid_block_def);
    ABIOS_l_blk.f_parms.LID = lid;
    ABIOS_l_blk.f_parms.unit = 0;;
    ABIOS_l_blk.f_parms.function = GET_LID_BLOCK_SIZE;
    ABIOS_l_blk.f_parms.ret_code = 0x5a5a;
    ABIOS_l_blk.f_parms.time_out = 0;

    if (ABIOSCall(lid,0,(void  *)&ABIOS_l_blk))
        return (1);

    lid_blk_size = ABIOS_l_blk.s_parms.blk_size; // Get the block size

    // Fill POS regs and card ID with FF in case this does not work

    *card_ID = 0xFFFF;
    for (i=0; i<NUM_POS_BYTES; i++) { pos_regs[i] = 0x00; };

    // Get the POS registers and card ID for the commanded slot

    ABIOS_r_blk.f_parms.req_blk_len = lid_blk_size;
    ABIOS_r_blk.f_parms.LID = lid;
    ABIOS_r_blk.f_parms.unit = 0;;
    ABIOS_r_blk.f_parms.function = READ_POS_REGS_CARD;
    ABIOS_r_blk.f_parms.ret_code = 0x5a5a;
    ABIOS_r_blk.f_parms.time_out = 0;

    ABIOS_r_blk.s_parms.slot_num = (UCHAR)slot_num & 0x0F;
    ABIOS_r_blk.s_parms.pos_buf = (void  *)pos_regs;
    ABIOS_r_blk.s_parms.card_ID = 0xFFFF;

    if (ABIOSCall(lid,0,(void  *)&ABIOS_r_blk))
       rc = 1;
     else {                                       // Else
       *card_ID = ABIOS_r_blk.s_parms.card_ID;   //    Set the card ID value
       rc = 0;
      }
    FreeLIDEntry(lid);
    return(rc);

}

UCHAR get_pos_data (int slot, int reg)
{
   UCHAR pos;
   CARD *cptr;

   cptr = &card[slot-1];            // set pointer to beg of card array
   if (reg == 0)                    // card ID
      pos = LOUSHORT(cptr->card_ID);
   else
     if ( reg == 1)
      pos = HIUSHORT(cptr->card_ID);
   else
      pos = cptr->pos_regs[reg-2];  // POS data register
   return (pos);
}

// Device Initialization Routine

int Init(PREQPACKET rp)
{
    USHORT lid;

    register char  *p;

    // store DevHlp entry point

    DevHlp = rp->s.Init.DevHlp;  // save DevHlp entry point

    if (!(GetLIDEntry(0x10, 0, 1, &lid))) { // get LID for POS
       FreeLIDEntry(lid);

  // Micro Channel (tm) setup section

  bus = 1;                      // MCA bus

      //    Get the POS data and card ID for each of 8 possible slots

      for (i=0;i <= MAX_NUM_SLOTS; i++)
         get_POS(i+1,(FARPOINTER)&card[i].card_ID,(FARPOINTER)card[i].pos_regs);

      matches = 0;
      for (i=0, pcard = card; i <= MAX_NUM_SLOTS; i++, pcard++) {
         if (pcard->card_ID == TARGET_ID) {
            matches = 1;
            break;
            }
         }

      if (matches == 0) {           // at least one board found
   DosPutMessage(1, 8, devhdr.DHname);
   DosPutMessage(1,strlen(NoMatchMsg),NoMatchMsg);
     rp->s.InitExit.finalCS = (OFF) 0;
   rp->s.InitExit.finalDS = (OFF) 0;
   return (RPDONE | RPERR | ERROR_BAD_COMMAND);
   }
```

```
      // calculate the board address from the POS regs

    board_address = ((unsigned long) get_pos_data(i+1, 4) << 16) |
    ((unsigned long)(get_pos_data(i+1, 3) & 1) << 15);
  }

  else

  // ISA bus setup

  {
  bus = 0;                      // ISA bus

  // get parameters, IRQ (not used yet), port addr and base mem addr

  for (p = rp->s.Init.args; *p && *p != ' ';++p);// skip driver name
  for (; *p == ' '; ++p);       // skip blanks following driver name
  if (*p)
  {
   board_address=0;           // i/o port address
   for (; *p != '\0'; ++p)    // get board address
       board_address = (board_address << 4) + (hex2bin(*p));
   addr_struct.board_addr = board_address;
   }
  }

  if (bus)
         DosPutMessage(1,strlen(MainMsgMCA),MainMsgMCA);
      else
         DosPutMessage(1,strlen(MainMsgISA),MainMsgISA);

  // send back our cs and ds end values to os/2

  if (SegLimit(HIUSHORT((void  *) Init), &rp->s.InitExit.finalCS) ||
     SegLimit(HIUSHORT((void  *) MainMsgISA), &rp->s.InitExit.finalDS))
       Abort();

  Beep(200,500);
  Beep(200,500);
  Beep(250,500);
  Beep(300,500);
  Beep(250,500);
  Beep(300,500);

  return (RPDONE);

}
```

Figure A-7. Memory-mapped driver, 32-bit version.

```
/* file sample.c
   sample OS/2 serial device driver
*/

#include "drvlib.h"
#include "uart.h"
#include "serial.h"

extern void near STRAT(); /* name of strat rout.*/
extern void near TIM_HNDLR(); /* timer handler  */
extern int  near INT_HNDLR(); /* interrupt hand */

DEVICEHDR devhdr = {
 (void far *) 0xFFFFFFFF,  /* link              */
 (DAW_CHR | DAW_OPN | DAW_LEVEL1),/* attribute  */
 (OFF) STRAT,              /* &strategy         */
 (OFF) 0,                  /* &IDCroutine       */
 "DEVICE1 "
  };

CHARQUEUE   rx_queue;      /* receiver queue    */
CHARQUEUE   tx_queue;      /* transmitter queue */
FPFUNCTION  DevHlp=0;      /* for DevHlp calls  */
LHANDLE     lock_seg_han;  /* handle for locking*/
PHYSADDR    appl_buffer=0; /* address of caller */
PREQPACKET  p=0L;          /* Request Packet ptr*/
ERRCODE     err=0;         /* error return      */
void        far *ptr;      /* temp far pointer  */
DEVICEHDR   *hptr;         /* pointer to Device */
USHORT      i;             /* general counter   */
UARTREGS    uart_regs;     /* uart registers    */
ULONG       WriteID=0L;    /* ID for write Block*/
ULONG       ReadID=0L;     /* ID for read Block */
PREQPACKET  ThisReadRP=0L; /* for read Request  */
PREQPACKET  ThisWriteRP=0L;/* for write Request */
char        inchar,outchar;/* temp chars        */
USHORT      baud_rate;     /* current baud rate */
unsigned    int savepid;   /* PID of driver own */
UCHAR       opencount;     /* number of times   */
ULONG       tickcount;     /* for timeouts      */
unsigned    int com_error_word; /* UART status  */
USHORT      port;          /* port variable     */
USHORT      temp_bank;     /* holds UART bank   */
QUEUE       rqueue;        /* receive queue info*/

void near init();
void near enable_write();
void near disable_write();
void near set_dlab();
void near reset_dlab();
void near config_82050();

char   IntFailMsg[] = " interrupt handler failed to install.\r\n";
char   MainMsg[] = " OS/2 Serial Device Driver V1.0 installed.\r\n";

/* common entry point to strat routines */

int main(PREQPACKET rp, int dev )
{
   void far *ptr;
   int far *pptr;
   PLINFOSEG liptr;    /* pointer to local info */
   int i;
   ULONG addr;

   switch(rp->RPcommand)
    {
    case RPINIT:       /* 0x00                  */

        /* init called by kernel in prot mode */

        return Init(rp,dev);
```

```
case RPOPEN:       /* 0x0d                  */

    /* get current processes id */

    if (GetDOSVar(2,&ptr))
        return (RPDONE|RPERR|ERROR_BAD_COMMAND);

    /* get process info */

    liptr = *((PLINFOSEG far *) ptr);

    /* if this device never opened */

    if (opencount == 0) /* 1st time dev op'd*/
    {
        ThisReadRP=0L;
        ThisWriteRP=0L;
        opencount=1;  /* set open counter   */
        savepid = liptr->pidCurrent; /* PID */
        QueueInit(&rx_queue);/* init driver */
        QueueInit(&tx_queue);
    }
    else
        {
        if (savepid != liptr->pidCurrent)
            return (RPDONE | RPERR | RPBUSY );
        ++opencount;       /* bump counter  */
    }
    return (RPDONE);

case RPCLOSE:       /* 0x0e                 */

    /* get process info of caller */

    if (GetDOSVar(2,&ptr))
        return (RPDONE|RPERR|ERROR_BAD_COMMAND); /* no info  */

    /* get process info from os/2 */

    liptr= *((PLINFOSEG far *) ptr); /* PID */
    if (savepid != liptr->pidCurrent ||
       opencount == 0)
    return (RPDONE|RPERR|ERROR_BAD_COMMAND);
    --opencount;   /* close counts down open*/

    if (ThisReadRP !=0 && opencount == 0) {
        Run((ULONG) ThisReadRP); /* dangling*/
        ThisReadRP=0L;
    }
    return (RPDONE);     /* return 'done'   */

case RPREAD:             /* 0x04            */

    /*  Try to read a character */

    ThisReadRP = rp;
    if (opencount == 0)/* drvr was closed   */
    {
        rp->s.ReadWrite.count = 0;  /* EOF  */
        return(RPDONE);
    }
    com_error_word=0;/* start off no errors */
    ReadID = (ULONG) rp;
    if (Block(ReadID, -1L, 0, &err))
        if (err == 2)       /* interrupted  */
           return(RPDONE|RPERR|ERROR_CHAR_CALL_INTERRUPTED);

    if (rx_queue.qcount == 0) {
       rp->s.ReadWrite.count=0;
       return (RPDONE|RPERR|ERROR_NOT_READY);
       }

    i=0;
    do {
      if (MoveData((FARPOINTER)&inchar,
      (FARPOINTER) (rp->s.ReadWrite.buffer+i),
      1,
                   MOVE_VIRTTOPHYS))
        return(RPDONE|RPERR|ERROR_GEN_FAILURE);
        }
    while (++i < rp->s.ReadWrite.count
        && !QueueRead(&rx_queue,&inchar));
    rp->s.ReadWrite.count = i;
    QueueInit(&rx_queue);
    return(rp->RPstatus);

case RPWRITE:        /* 0x08                */

    ThisWriteRP = rp;

    /* transfer characters from user buffer */

    addr=rp->s.ReadWrite.buffer;/* get addr */
    for (i = rp->s.ReadWrite.count; i; --i,++addr)
    {
      if (MoveData((FARPOINTER)addr,
         (FARPOINTER)&outchar,
                          1,
                          MOVE_PHYSTOVIRT))
            return (RPDONE|RPERR|ERROR_GEN_FAILURE);

      if (QueueWrite(&tx_queue,outchar))
          return (RPDONE|RPERR|ERROR_GEN_FAILURE);
      }
    WriteID = (ULONG) rp;
    enable_write();

    if (Block(WriteID, -1L, 0, &err))
        if (err == 2)   /* interrupted      */
            return(RPDONE|RPERR|ERROR_CHAR_CALL_INTERRUPTED);

    tickcount=MIN_TIMEOUT; /* reset timeout */
    QueueInit(&tx_queue);
    return (rp->RPstatus);

case RPINPUT_FLUSH:      /* 0x07            */

    QueueFlush(&rx_queue);
    return (RPDONE);

case RPOUTPUT_FLUSH:     /* 0x0b            */

    QueueFlush(&tx_queue);
    return (RPDONE);

case RPIOCTL:            /* 0x10            */
```

```
if (!((rp->s.IOCtl.category == SAMPLE_CAT)
   || (rp->s.IOCtl.category == 0x01)))
      return (RPDONE);

switch (rp->s.IOCtl.function)
{
case 0x41:    /* set baud rate          */
/* set baud rate to 1.2, 2.4, 9.6, 19.2 */
/* verify caller owns the buffer area   */

if(VerifyAccess(
 SELECTOROF(rp->s.IOCtl.parameters),
 OFFSETOF(rp->s.IOCtl.parameters),
 2,            /* two bytes            */
 1) )          /* read/write           */
        return (RPDONE|RPERR|ERROR_GEN_FAILURE);

 /* lock the segment down temp */

 if(LockSeg(
 SELECTOROF(rp->s.IOCtl.parameters),
 0,          /* lock for < 2 sec      */
 0,          /* wait for seg lock     */
 (PLHANDLE) &lock_seg_han)) /* handle */
     return (RPDONE|RPERR|ERROR_GEN_FAILURE);

 /* get physical address of buffer */
 if (VirtToPhys(
 (FARPOINTER) rp->s.IOCtl.parameters,
 (FARPOINTER) &appl_buffer))
     return (RPDONE|RPERR|ERROR_GEN_FAILURE);

 /* move data to local driver buffer */

 if(MoveData(
 (FARPOINTER) appl_buffer,  /* source             */
 (FARPOINTER)&baud_rate,    /* destination        */
 2,                         /* 2 bytes            */
             MOVE_PHYSTOVIRT))
        return (RPDONE|RPERR|ERROR_GEN_FAILURE);

 if (UnPhysToVirt()) /* release selector*/
        return(RPDONE|RPERR|ERROR_GEN_FAILURE);

 /* unlock segment */

 if(UnLockSeg(lock_seg_han))
      return(RPDONE|RPERR|ERROR_GEN_FAILURE);

 switch (baud_rate)
    {
    case 1200:

        uart_regs.Bal=0xe0;
        uart_regs.Bah=0x01;
        break;

    case 2400:

        uart_regs.Bal=0xf0;
        uart_regs.Bah=0x00;
        break;
```

```
            case 9600:

                uart_regs.Bal=0x3c;
                uart_regs.Bah=0x00;
                break;

            case 19200:

                uart_regs.Bal=0x1e;
                uart_regs.Bah=0x00;
                break;

            case 38400:

                uart_regs.Bal=0x0f;
                uart_regs.Bah=0x00;
                break;

error:
            return (RPDONE|RPERR|ERROR_BAD_COMMAND);

            }
            init();      /* reconfigure uart    */
            return (RPDONE);

        case 0x68:       /* get number of chars */

            /* verify caller owns the buffer    */

            if(VerifyAccess(
            SELECTOROF(rp->s.IOCtl.buffer),
            OFFSETOF(rp->s.IOCtl.buffer),
            4,            /* 4 bytes            */
            1) )          /* read/write         */
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* lock the segment down temp */

            if(LockSeg(
            SELECTOROF(rp->s.IOCtl.buffer),
            0,           /* lock for < 2 sec    */
            0,           /* wait for seg lock   */
            (PLHANDLE) &lock_seg_han)) /* handle*/
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* get physical address of buffer */

            if (VirtToPhys(
            (FARPOINTER) rp->s.IOCtl.buffer,
            (FARPOINTER) &appl_buffer))
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            rqueue.cch=rx_queue.qcount;
            rqueue.cb=rx_queue.qsize;

            /* move data to local driver buffer */

            if(MoveData(
            (FARPOINTER)&rx_queue,  /* source   */
            (FARPOINTER) appl_buffer, /* dest   */
            4,           /* 4 bytes             */
                              MOVE_PHYSTOVIRT))
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            if (UnPhysToVirt())
                return(RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* unlock segment */

            if(UnLockSeg(lock_seg_han))
                return(RPDONE|RPERR|ERROR_GEN_FAILURE);

            return (RPDONE);

        case 0x6d:    /* get COM error info     */

            /* verify caller owns the buffer    */

            if(VerifyAccess(
            SELECTOROF(rp->s.IOCtl.buffer),
            OFFSETOF(rp->s.IOCtl.buffer),
            2,        /* two bytes              */
            1) )      /* read/write             */
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* lock the segment down temp */

            if(LockSeg(
            SELECTOROF(rp->s.IOCtl.buffer),
            0,        /* lock for < 2 sec       */
            0,        /* wait for seg lock      */
            (PLHANDLE) &lock_seg_han)) /* handle*/
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* get physical address of buffer */

            if (VirtToPhys(
            (FARPOINTER) rp->s.IOCtl.buffer,
            (FARPOINTER) &appl_buffer))
               return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* move data to application buffer */

            if(MoveData(
            (FARPOINTER)&com_error_word, /* source */
            (FARPOINTER) appl_buffer,    /* dest   */
            2,            /* 2 bytes               */
                            MOVE_VIRTTOPHYS))
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            if (UnPhysToVirt())
                return(RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* unlock segment */

            if(UnLockSeg(lock_seg_han))
                return(RPDONE|RPERR|ERROR_GEN_FAILURE);

            return (RPDONE);

        default:
            return(RPDONE|RPERR|ERROR_GEN_FAILURE);
        }

        /* don't allow deinstall */
```

```
    case RPDEINSTALL:  /* 0x14                  */
        return(RPDONE|RPERR|ERROR_BAD_COMMAND);

    /* all other commands are ignored */

    default:
        return(RPDONE);

    }
}

void enable_write()

/* enable write interrupts on uart */

{
    int  port;
    int  reg_val;

    port=UART_PORT_ADDRESS;
    reg_val=inp(port+2) & 0x60;
    set_bank(00);
    outp((port+1),inp(port+1) | 0x12);
    outp((port+2),reg_val);

}
void disable_write()

/* turn off write interrupts on uart */

{
    int  port;
    int  reg_val;

    port=UART_PORT_ADDRESS;
    reg_val=inp(port+2) & 0x60;
    set_bank(00);
    outp((port+1),inp(port+1) & 0xed);
    outp((port+2),reg_val);

}

void init ()

/* intializes software and configures 82050 */

{
    config_82050 ();    /* Configure 82050      */
    set_bank(01);
}

void config_82050()

/*  Configure the 82050      */

{
    int  port;
    int inval;

    Disable();                          /* disable interrupts    */
    port=UART_PORT_ADDRESS;
    /* set stick bit */

    set_bank(01);                       /* stick bit            */
    outp((port+7),0x10);                /* reset port           */
    outp ((port+1), uart_regs.Txf);     /* stick bit            */

    set_bank (02);                      /* general config       */
    outp ((port + 4), uart_regs.Imd);   /*auto rupt             */
    outp ((port + 7), uart_regs.Rmd);
    outp ((port + 5), uart_regs.Acr1);  /* cntl-z               */
    outp ((port + 3), uart_regs.Tmd);   /* no 9 bit             */
    outp ((port + 1), uart_regs.Fmd);   /* rx fifo              */
    outp ((port + 6), uart_regs.Rie);   /* enable               */

    set_bank (03);                      /* modemconfiguration   */

    outp ((port + 0), uart_regs.Clcf);  /* clock                */
    set_dlab (03);                      /*                      */
    outp ((port + 0), uart_regs.Bbl);   /* BRGB lsb             */
    outp ((port + 1), uart_regs.Bbh);   /* BRGB msb             */
    reset_dlab (03);                    /*                      */
    outp ((port + 3), uart_regs.Bbcf);  /* BRGB                 */
    outp ((port + 6), uart_regs.Tmie);  /* timer b              */

    set_bank (00);                      /* general cfg          */
    outp ((port + 1), uart_regs.Ger);   /* enable               */
    outp ((port + 3), uart_regs.Lcr);   /* 8 bit                */
    outp ((port + 7), uart_regs.Acr0);  /* CR                   */
    outp ((port + 4), uart_regs.Mcr_0); /* no DTR               */
    set_dlab (00);                      /*                      */
    outp ((port + 0), uart_regs.Bal);   /* BRGA lsb             */
    outp ((port + 1), uart_regs.Bah);   /* BRGA msb             */
    reset_dlab (00);
    set_bank(01);

    Enable();                           /* turn on              */
}

void set_dlab (bank)

/*  Set DLAB bit to allow access to divisior registers  */

int bank;
{
    int  inval;
    int  port;

    port=UART_PORT_ADDRESS;
    set_bank (00);
    inval=inp(port +3);
    inval =inval | 0x80;                 /* set dlab in LCR     */
    outp ((port+3),inval);
    set_bank (bank);
}

getsrc()

{
    int   v,src;
    int   port;

    port=UART_PORT_ADDRESS;             /* get base address     */
    v=inp(port+2);                      /* get data             */
    src=v & 0x0e;                          /* mask bits            */
    src=src/2;                             /* divide by 2          */
    return(src);                           /* and pass it back     */
}

set_bank(bank_num)

/* set bank of 82050 uart */

int  bank_num;

{
    int reg_val;
    int   port;

    reg_val=bank_num*0x20;                 /* select bank numb */
    port=UART_PORT_ADDRESS;                /* get real port    */
    outp(port+gir_addr,reg_val);           /* output      */
}

void reset_dlab (bank)

/*  Reset DLAB bit of LCR   */

int bank;

{
    int  inval;
    int  port;

    port=UART_PORT_ADDRESS;
    set_bank (00);
    inval=inp (port +3);
    inval = (inval & 0x7f);                /* dlab = 0 in LCR      */
    outp ((port+3),inval);
    set_bank (bank);
}

/* 82050 interrupt handler */

void interrupt_handler ()
{
    int  rupt_dev;
    int  source;
    int  cmd_b;
    int  st_b;
    int  port;
    int  temp;
    int  rxlevel;


    port=UART_PORT_ADDRESS;
    outp((port+2),0x20);                   /* switch to bank 1     */
    source = getsrc ();                    /* get vector           */
    switch (source)
    {

    /* optional timer service routine */

    case timer :

        st_b=inp (port+3);                 /* dec transmit count   */
        if ( ThisReadRP == 0)              /* nobody waiting       */
```

```
            break;
        ThisReadRP->RPstatus=(RPDONE|RPERR|ERROR_NOT_READY);
        Run ((ULONG)  ThisWriteRP);     /* run thread           */
        ThisWriteRP=0;
        break;

    case txm   :
    case txf   :

        /* spurious write interrupt */

        if ( ThisWriteRP == 0) {
            temp=inp(port+2);
            break;
        }

        /* keep transmitting until no data left */

        if  (!(QueueRead(&tx_queue,&outchar)))
        {
             outp((port), outchar);
             tickcount=MIN_TIMEOUT;
             break;
        }

        /* done writing, run blocked thread     */

        tickcount=MIN_TIMEOUT;
        disable_write();
        ThisWriteRP->RPstatus = (RPDONE);
        Run ((ULONG)  ThisWriteRP);
        ThisWriteRP=0;
        break;

    case ccr   :

        /* control character, treat as normal   */

        inchar=inp(port+5);

    case rxf   :

        /* rx fifo service routine */

        if ( ThisReadRP == 0)
            inchar=inp (port); /* get character */
        else
        {
        temp=inp(port+4);
        rxlevel=(temp & 0x70) / 0x10;

         /* empty out chip FIFO */

         while (rxlevel !=0) {

           inchar=inp (port); /* get character */
           rxlevel--;
           tickcount=MIN_TIMEOUT;

           /* write input data to queue */

           if(QueueWrite(&rx_queue,inchar))
```

```
             /* error, queue must be full */

             {
             ThisReadRP->RPstatus=(RPDONE|RPERR|ERROR_GEN_FAILURE);
             Run ((ULONG) ThisReadRP);
             ThisReadRP=0;
             break;
             }
           com_error_word |= inp(port+5);

        } /* while rxlevel */
     } /* else */
  } /* switch (source) */
  EOI(5);
}
void timer_handler()
{
  if (ThisReadRP == 0)
        return;

  tickcount--;
  if(tickcount == 0)  {
    ThisReadRP->RPstatus=(RPDONE);
    Run ((ULONG) ThisReadRP);
    ThisReadRP=0L;
    tickcount=MIN_TIMEOUT;
    }
}

/* Device Initialization Routine */

int Init(PREQPACKET rp, int dev)
{
    register char far *p;

    /* store DevHlp entry point */

    DevHlp = rp->s.Init.DevHlp;

    /* install interrupt hook in vector */

    if (SetTimer((PFUNCTION)TIM_HNDLR))
                goto fail;

    rx_queue.qsize=QUEUE_SIZE;
    tx_queue.qsize=QUEUE_SIZE; /* init queue    */
    init();                    /* init the port */
    tickcount=MIN_TIMEOUT;     /* set timeout   */

    if(SetIRQ(5,(PFUNCTION)INT_HNDLR,0)) {

     /* if we failed, deinstall driver cs+ds=0 */
fail:
     DosPutMessage(1, 8, devhdr.DHname);
     DosPutMessage (1,strlen(IntFailMsg),IntFailMsg);
     rp->s.InitExit.finalCS = (OFF) 0;
     rp->s.InitExit.finalDS = (OFF) 0;
     return (RPDONE | RPERR | ERROR_BAD_COMMAND);
     }

/* output initialization message */

DosPutMessage(1, 8, devhdr.DHname);
DosPutMessage(1, strlen(MainMsg), MainMsg);

/* send back our cs and ds values to os/2 */

if (SegLimit(HIUSHORT((void far *) Init),&rp->s.InitExit.finalCS)
    || SegLimit(HIUSHORT((void far *) MainMsg),
    &rp->s.InitExit.finalDS))
     Abort();
   return(RPDONE);
}
```

Figure A-8. Serial device driver, 16-bit version.

```
/* file sample.c
   sample OS/2 serial device driver
*/

#include "drvlib.h"
#include "uart.h"
#include "serial.h"
#include <stdio.h>

int main (PREQPACKET rp, int d);

DEVICEHDR devhdr = {
 (void  *) 0xFFFFFFFF,         /* link              */
 (DAW_CHR | DAW_OPN | DAW_LEVEL1),/* attribute         */
 &main,                           /* &strategy         */
 0,                               /* &IDCroutine       */
 "DEVICE1 "
  };

CHARQUEUE   rx_queue;      /* receiver queue    */
CHARQUEUE   tx_queue;      /* transmitter queue */
PFUNCTION  DevHlp=0;      /* for DevHlp calls  */
LHANDLE     lock_seg_han;  /* handle for locking*/
PHYSADDR    appl_buffer=0; /* address of caller */
PREQPACKET  p=0L;          /* Request Packet ptr*/
ERRCODE     err=0;         /* error return      */
void         *ptr;      /* temp  pointer  */
DEVICEHDR   *hptr;         /* pointer to Device */
USHORT      i;             /* general counter   */
UARTREGS    uart_regs;     /* uart registers    */
ULONG       WriteID=0L;    /* ID for write Block*/
ULONG       ReadID=0L;     /* ID for read Block */
PREQPACKET  ThisReadRP=0L; /* for read Request  */
PREQPACKET  ThisWriteRP=0L;/* for write Request */
char        inchar,outchar;/* temp chars        */
USHORT      baud_rate;     /* current baud rate */
unsigned    int savepid;   /* PID of driver own */
UCHAR       opencount;     /* number of times   */
ULONG       tickcount;     /* for timeouts      */
unsigned    int com_error_word; /* UART status  */
USHORT      port;          /* port variable     */
USHORT      temp_bank;     /* holds UART bank   */
QUEUE       rqueue;        /* receive queue info*/
```

```
void  init();
void  enable_write();
void  disable_write();
void  set_dlab();
void  reset_dlab();
void  config_82050();

char   IntFailMsg[] = " interrupt handler failed to install.\r\n";
char   MainMsg[] = " OS/2 Serial Device Driver V1.0 installed.\r\n";

/* common entry point to strat routines */

int main(PREQPACKET rp, int dev )
{
   void  *ptr;
   int  *pptr;
   PLINFOSEG liptr;    /* pointer to local info */
   int i;
   ULONG addr;

   switch(rp->RPcommand)
    {
    case RPINIT:       /* 0x00                  */

        /* init called by kernel in prot mode */

        return Init(rp,dev);

    case RPOPEN:       /* 0x0d                  */

        /* get current processes id */

        if (GetDOSVar(2,&ptr))
            return (RPDONE|RPERR|ERROR_BAD_COMMAND);

        /* get process info */

        liptr = *((PLINFOSEG  *) ptr);

        /* if this device never opened */

        if (opencount == 0) /* 1st time dev op'd*/
        {
            ThisReadRP=0L;
            ThisWriteRP=0L;
            opencount=1;  /* set open counter   */
            savepid = liptr->pidCurrent; /* PID */
            QueueInit(&rx_queue);/* init driver */
            QueueInit(&tx_queue);
        }
        else
            {
            if (savepid != liptr->pidCurrent)
                return (RPDONE | RPERR | RPBUSY );
            ++opencount;       /* bump counter  */
        }
        return (RPDONE);

    case RPCLOSE:        /* 0x0e                */

        /* get process info of caller */

        if (GetDOSVar(2,&ptr))
            return (RPDONE|RPERR|ERROR_BAD_COMMAND); /* no info  */

        /* get process info from os/2 */

        liptr= *((PLINFOSEG  *) ptr); /* PID */
        if (savepid != liptr->pidCurrent ||
           opencount == 0)
        return (RPDONE|RPERR|ERROR_BAD_COMMAND);
        --opencount;   /* close counts down open*/

        if (ThisReadRP !=0 && opencount == 0) {
            Run((ULONG) ThisReadRP); /* dangling*/
            ThisReadRP=0L;
        }
        return (RPDONE);     /* return 'done'   */

    case RPREAD:             /* 0x04            */

        /*  Try to read a character */

        ThisReadRP = rp;
        if (opencount == 0)/* drvr was closed   */
        {
            rp->s.ReadWrite.count = 0;  /* EOF  */
            return(RPDONE);
        }
        com_error_word=0;/* start off no errors */
        ReadID = (ULONG) rp;
        if (Block(ReadID, -1L, 0, &err))
            if (err == 2)       /* interrupted  */
               return(RPDONE|RPERR|ERROR_CHAR_CALL_INTERRUPTED);

        if (rx_queue.qcount == 0) {
           rp->s.ReadWrite.count=0;
           return (RPDONE|RPERR|ERROR_NOT_READY);
           }

        i=0;
        do {
          if (MoveData((PVOID)&inchar,
          (PVOID) (rp->s.ReadWrite.buffer+i),
          1,
                       MOVE_VIRTTOPHYS))
            return(RPDONE|RPERR|ERROR_GEN_FAILURE);
            }
        while (++i < rp->s.ReadWrite.count
            && !QueueRead(&rx_queue,&inchar));
        rp->s.ReadWrite.count = i;
        QueueInit(&rx_queue);
        return(rp->RPstatus);

    case RPWRITE:        /* 0x08                */

        ThisWriteRP = rp;

        /* transfer characters from user buffer */

        addr=rp->s.ReadWrite.buffer;/* get addr */
        for (i = rp->s.ReadWrite.count; i; --i,++addr)
        {
          if (MoveData((PVOID)addr,
             (PVOID)&outchar,
                              1,
                               MOVE_PHYSTOVIRT))
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

          if (QueueWrite(&tx_queue,outchar))
              return (RPDONE|RPERR|ERROR_GEN_FAILURE);
          }
        WriteID = (ULONG) rp;
        enable_write();

        if (Block(WriteID, -1L, 0, &err))
            if (err == 2)   /* interrupted      */
                return(RPDONE|RPERR|ERROR_CHAR_CALL_INTERRUPTED);

        tickcount=MIN_TIMEOUT; /* reset timeout */
        QueueInit(&tx_queue);
        return (rp->RPstatus);

    case RPINPUT_FLUSH:      /* 0x07            */

        QueueFlush(&rx_queue);
        return (RPDONE);

    case RPOUTPUT_FLUSH:     /* 0x0b            */

        QueueFlush(&tx_queue);
        return (RPDONE);

    case RPIOCTL:            /* 0x10            */

        if (!((rp->s.IOCtl.category == SAMPLE_CAT)
           || (rp->s.IOCtl.category == 0x01)))
              return (RPDONE);

        switch (rp->s.IOCtl.function)
        {
        case 0x41:    /* set baud rate          */
        /* set baud rate to 1.2, 2.4, 9.6, 19.2 */
        /* verify caller owns the buffer area   */

        if(VerifyAccess(
         SELECTOROF(rp->s.IOCtl.parameters),
         OFFSETOF(rp->s.IOCtl.parameters),
         2,             /* two bytes            */
         1) )           /* read/write           */
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

         /* lock the segment down temp */

         if(LockSeg(
         SELECTOROF(rp->s.IOCtl.parameters),
         0,          /* lock for < 2 sec      */
         0,          /* wait for seg lock     */
         (PLHANDLE) &lock_seg_han)) /* handle */
             return (RPDONE|RPERR|ERROR_GEN_FAILURE);

         /* get physical address of buffer */
         if (VirtToPhys(
         (PVOID) rp->s.IOCtl.parameters,
         (PVOID) &appl_buffer))
             return (RPDONE|RPERR|ERROR_GEN_FAILURE);

         /* move data to local driver buffer */
```

```
         if(MoveData(
         (PVOID) appl_buffer,  /* source             */
         (PVOID)&baud_rate,    /* destination        */
         2,                         /* 2 bytes            */
                      MOVE_PHYSTOVIRT))
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

         if (UnPhysToVirt()) /* release selector*/
                return(RPDONE|RPERR|ERROR_GEN_FAILURE);

         /* unlock segment */

         if(UnLockSeg(lock_seg_han))
              return(RPDONE|RPERR|ERROR_GEN_FAILURE);

         switch (baud_rate)
            {
            case 1200:

                uart_regs.Bal=0xe0;
                uart_regs.Bah=0x01;
                break;

            case 2400:

                uart_regs.Bal=0xf0;
                uart_regs.Bah=0x00;
                break;

            case 9600:

                uart_regs.Bal=0x3c;
                uart_regs.Bah=0x00;
                break;

            case 19200:

                uart_regs.Bal=0x1e;
                uart_regs.Bah=0x00;
                break;

            case 38400:

                uart_regs.Bal=0x0f;
                uart_regs.Bah=0x00;
                break;

error:
            return (RPDONE|RPERR|ERROR_BAD_COMMAND);

            }
            init();      /* reconfigure uart    */
            return (RPDONE);

        case 0x68:       /* get number of chars */

            /* verify caller owns the buffer    */

            if(VerifyAccess(
            SELECTOROF(rp->s.IOCtl.buffer),
            OFFSETOF(rp->s.IOCtl.buffer),
            4,            /* 4 bytes            */
            1) )          /* read/write         */
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* lock the segment down temp */

            if(LockSeg(
            SELECTOROF(rp->s.IOCtl.buffer),
            0,           /* lock for < 2 sec    */
            0,           /* wait for seg lock   */
            (PLHANDLE) &lock_seg_han)) /* handle*/
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* get physical address of buffer */

            if (VirtToPhys(
            (PVOID) rp->s.IOCtl.buffer,
            (PVOID) &appl_buffer))
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            rqueue.cch=rx_queue.qcount;
            rqueue.cb=rx_queue.qsize;

            /* move data to local driver buffer */

            if(MoveData(
            (PVOID)&rx_queue,  /* source   */
            (PVOID) appl_buffer, /* dest   */
            4,           /* 4 bytes             */
                             MOVE_PHYSTOVIRT))
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            if (UnPhysToVirt())
                return(RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* unlock segment */

            if(UnLockSeg(lock_seg_han))
                return(RPDONE|RPERR|ERROR_GEN_FAILURE);

            return (RPDONE);

        case 0x6d:    /* get COM error info     */

            /* verify caller owns the buffer    */

            if(VerifyAccess(
            SELECTOROF(rp->s.IOCtl.buffer),
            OFFSETOF(rp->s.IOCtl.buffer),
            2,        /* two bytes              */
            1) )      /* read/write             */
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* lock the segment down temp */

            if(LockSeg(
            SELECTOROF(rp->s.IOCtl.buffer),
            0,        /* lock for < 2 sec       */
            0,        /* wait for seg lock      */
            (PLHANDLE) &lock_seg_han)) /* handle*/
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* get physical address of buffer */

            if (VirtToPhys(
            (PVOID) rp->s.IOCtl.buffer,
            (PVOID) &appl_buffer))
               return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* move data to application buffer */

            if(MoveData(
            (PVOID)&com_error_word, /* source */
            (PVOID) appl_buffer,    /* dest   */
            2,            /* 2 bytes               */
                             MOVE_VIRTTOPHYS))
                return (RPDONE|RPERR|ERROR_GEN_FAILURE);

            if (UnPhysToVirt())
                return(RPDONE|RPERR|ERROR_GEN_FAILURE);

            /* unlock segment */

            if(UnLockSeg(lock_seg_han))
                return(RPDONE|RPERR|ERROR_GEN_FAILURE);

            return (RPDONE);

        default:
            return(RPDONE|RPERR|ERROR_GEN_FAILURE);
        }

        /* don't allow deinstall */

    case RPDEINSTALL:  /* 0x14                  */
        return(RPDONE|RPERR|ERROR_BAD_COMMAND);

    /* all other commands are ignored */

    default:
        return(RPDONE);

    }
}

void enable_write()

/* enable write interrupts on uart */

{
    int  port;
    int  reg_val;

    port=UART_PORT_ADDRESS;
    reg_val=inp(port+2) & 0x60;
    set_bank(00);
    outp((port+1),inp(port+1) | 0x12);
    outp((port+2),reg_val);

}
void disable_write()

/* turn off write interrupts on uart */

{
    int  port;
    int  reg_val;
    port=UART_PORT_ADDRESS;
    reg_val=inp(port+2) & 0x60;
    set_bank(00);
    outp((port+1),inp(port+1) & 0xed);
    outp((port+2),reg_val);

}

void init ()

/* intializes software and configures 82050 */

{
    config_82050 ();    /* Configure 82050      */
    set_bank(01);
}

void config_82050()

/*  Configure the 82050      */

{
    int  port;
    int inval;

    Disable();                          /* disable interrupts    */
    port=UART_PORT_ADDRESS;

    /* set stick bit */

    set_bank(01);                       /* stick bit            */
    outp((port+7),0x10);                /* reset port           */
    outp ((port+1), uart_regs.Txf);     /* stick bit            */

    set_bank (02);                      /* general config       */
    outp ((port + 4), uart_regs.Imd);   /*auto rupt             */
    outp ((port + 7), uart_regs.Rmd);
    outp ((port + 5), uart_regs.Acr1);  /* cntl-z               */
    outp ((port + 3), uart_regs.Tmd);   /* no 9 bit             */
    outp ((port + 1), uart_regs.Fmd);   /* rx fifo              */
    outp ((port + 6), uart_regs.Rie);   /* enable               */

    set_bank (03);                      /* modemconfiguration   */

    outp ((port + 0), uart_regs.Clcf);  /* clock                */
    set_dlab (03);                      /*                      */
    outp ((port + 0), uart_regs.Bbl);   /* BRGB lsb             */
    outp ((port + 1), uart_regs.Bbh);   /* BRGB msb             */
    reset_dlab (03);                    /*                      */
    outp ((port + 3), uart_regs.Bbcf);  /* BRGB                 */
    outp ((port + 6), uart_regs.Tmie);  /* timer b              */

    set_bank (00);                      /* general cfg          */
    outp ((port + 1), uart_regs.Ger);   /* enable               */
    outp ((port + 3), uart_regs.Lcr);   /* 8 bit                */
    outp ((port + 7), uart_regs.Acr0);  /* CR                   */
    outp ((port + 4), uart_regs.Mcr_0); /* no DTR               */
    set_dlab (00);                      /*                      */
    outp ((port + 0), uart_regs.Bal);   /* BRGA lsb             */
    outp ((port + 1), uart_regs.Bah);   /* BRGA msb             */
    reset_dlab (00);
    set_bank(01);
    Enable();                           /* turn on              */
}

void set_dlab (bank)

/*  Set DLAB bit to allow access to divisior registers  */

int bank;
{
    int  inval;
    int  port;

    port=UART_PORT_ADDRESS;
    set_bank (00);
    inval=inp(port +3);
    inval =inval | 0x80;                 /* set dlab in LCR     */
    outp ((port+3),inval);
    set_bank (bank);
}

getsrc()

{
    int   v,src;
    int   port;

    port=UART_PORT_ADDRESS;             /* get base address     */
    v=inp(port+2);                      /* get data             */
    src=v & 0x0e;                       /* mask bits            */
    src=src/2;                          /* divide by 2          */
    return(src);                        /* and pass it back     */
}

set_bank(bank_num)

/* set bank of 82050 uart */

int  bank_num;

{
    int reg_val;
    int   port;

    reg_val=bank_num*0x20;              /* select bank numb */
    port=UART_PORT_ADDRESS;             /* get real port    */
    outp(port+gir_addr,reg_val);        /* output      */
}

void reset_dlab (bank)

/*  Reset DLAB bit of LCR   */

int bank;

{
    int  inval;
    int  port;

    port=UART_PORT_ADDRESS;
    set_bank (00);
    inval=inp (port +3);
    inval = (inval & 0x7f);             /* dlab = 0 in LCR      */
    outp ((port+3),inval);
    set_bank (bank);
}

/* 82050 interrupt handler */

void interrupt_handler ()
{
    int  rupt_dev;
    int  source;
    int  cmd_b;
    int  st_b;
    int  port;
    int  temp;
    int  rxlevel;


    port=UART_PORT_ADDRESS;
    outp((port+2),0x20);                /* switch to bank 1     */
    source = getsrc ();                 /* get vector           */
    switch (source)
    {

    /* optional timer service routine */

    case timer :

        st_b=inp (port+3);              /* dec transmit count   */
        if ( ThisReadRP == 0)           /* nobody waiting       */
            break;
        ThisReadRP->RPstatus=(RPDONE|RPERR|ERROR_NOT_READY);
        Run ((ULONG)  ThisWriteRP);     /* run thread           */
        ThisWriteRP=0;
        break;

    case txm   :
    case txf   :

        /* spurious write interrupt */

        if ( ThisWriteRP == 0) {
            temp=inp(port+2);
            break;
        }

        /* keep transmitting until no data left */

        if  (!(QueueRead(&tx_queue,&outchar)))
        {
             outp((port), outchar);
             tickcount=MIN_TIMEOUT;
             break;
        }

        /* done writing, run blocked thread     */

        tickcount=MIN_TIMEOUT;
        disable_write();
        ThisWriteRP->RPstatus = (RPDONE);
        Run ((ULONG)  ThisWriteRP);
        ThisWriteRP=0;
        break;

    case ccr   :
        /* control character, treat as normal   */

        inchar=inp(port+5);

    case rxf   :

        /* rx fifo service routine */

        if ( ThisReadRP == 0)
            inchar=inp (port); /* get character */
        else
        {
        temp=inp(port+4);
        rxlevel=(temp & 0x70) / 0x10;

         /* empty out chip FIFO */

         while (rxlevel !=0) {

           inchar=inp (port); /* get character */
           rxlevel--;
           tickcount=MIN_TIMEOUT;

           /* write input data to queue */

           if(QueueWrite(&rx_queue,inchar))

             /* error, queue must be full */

             {
             ThisReadRP->RPstatus=(RPDONE|RPERR|ERROR_GEN_FAILURE);
             Run ((ULONG) ThisReadRP);
             ThisReadRP=0;
             break;
             }
           com_error_word |= inp(port+5);

        } /* while rxlevel */
     } /* else */
  } /* switch (source) */
  EOI(5);
}
void timer_handler()
{
  if (ThisReadRP == 0)
        return;

  tickcount--;
  if(tickcount == 0)  {
    ThisReadRP->RPstatus=(RPDONE);
    Run ((ULONG) ThisReadRP);
    ThisReadRP=0L;
    tickcount=MIN_TIMEOUT;
    }
}

/* Device Initialization Routine */

int Init(PREQPACKET rp, int dev)
{
    register char  *p;
    /* store DevHlp entry point */

    DevHlp = rp->s.Init.DevHlp;

    /* install interrupt hook in vector */

    if (SetTimer((PFUNCTION)timer_handler))
                goto fail;

    rx_queue.qsize=QUEUE_SIZE;
    tx_queue.qsize=QUEUE_SIZE; /* init queue    */
    init();                    /* init the port */
    tickcount=MIN_TIMEOUT;     /* set timeout   */

    if(SetIRQ(5,(PFUNCTION)interrupt_handler,0)) {

     /* if we failed, deinstall driver cs+ds=0 */
fail:
     DosPutMessage(1, 8, devhdr.DHname);
     DosPutMessage (1,strlen(IntFailMsg),IntFailMsg);
     rp->s.InitExit.finalCS = (OFFSET) 0;
     rp->s.InitExit.finalDS = (OFFSET) 0;
     return (RPDONE | RPERR | ERROR_BAD_COMMAND);
     }

/* output initialization message */

DosPutMessage(1, 8, devhdr.DHname);
DosPutMessage(1, strlen(MainMsg), MainMsg);

/* send back our cs and ds values to os/2 */

if (SegLimit(HIUSHORT((void  *) Init),&rp->s.InitExit.finalCS)
    || SegLimit(HIUSHORT((void  *) MainMsg),
    &rp->s.InitExit.finalDS))
     Abort();
   return(RPDONE);
}
```

Figure A-9. Serial device driver, 32-bit version.