SOME ASSEMBLY REQUIRED: OS/2 Device Drivers

A practitioner's guide to development of an asynchronous RS-232 terminal driver for OS/2 in C

AUTHOR: Steven J. Mastrianni

OS/2 device drivers continue to be a limiting factor in the acceptance and use of OS/2. DOS drivers abound, but OS/2 drivers are scarce as hen's teeth -- for a variety of reasons. OS/2 drivers are more complicated than DOS drivers. They've got to handle context switching and priorities and accommodate dual-mode operation (real versus protected) -- issues foreign to many DOS programmers. In this article, I'll describe how to build an asynchronous RS-232 terminal driver for OS/2 in C, complete with interrupt handler and timer support (the code you'll need to build this driver is available on BIX). Once you've seen how that's done, you'll have the basic understanding you need to write OS/2 drivers for other types of devices.

The Nature of the Beast

OS/2 device drivers, like other multitasking drivers, shield applications from the physical characteristics of I/O devices (e.g., timing or I/O port addressing). An application in need of I/O service transmits a request to the OS/2 kernel, which in turn calls a driver. The device driver handles all the hardware details, such as register setup, interrupt handling, and error checking. When the request is complete, the device driver massages the data into a format recognizable by the application. It sends the data or a status indication to the application and notifies the kernel that the request is complete. If the request cannot be handled immediately, the driver may either block the requesting thread or return a Request Not Done status to the kernel. Either way, the driver then relinquishes the CPU so that other threads can run.

DOS device drivers do not have a direct OS/2 counterpart. They are simple single-task polling drivers. Even interrupt drivers under DOS poll until interrupt processing is complete. DOS device drivers support one request at a time, and any subsequent requests from the DOS kernel will cause the system to crash.

In contrast, an OS/2 driver must manage overlapping requests from different processes and threads, and it must therefore be reentrant. It must also handle interrupts from the device and interrupts from a timer handler. In addition, the OS/2 driver must oversee switches from protected mode to real mode. It must accomplish these operations in an efficient manner, allowing other threads to gain access to the CPU, and, most important, it must do all these tasks reliably. Because it operates at ring 0, the OS/2 driver is the only program that has access to critical system functions (e.g., the interrupt system and timer). The driver therefore must be a trusted program, because any error in the driver can cause a fatal system crash.

OS/2 device drivers must also be bimodal, which means they must operate in real mode and protected mode. The interrupts must continue to be processed, and the requests must be completed, even if the user switches from the OS/2 prompt to the DOS compatibility box and back. They must be able to deinstall when requested, releasing any memory used by the driver to OS/2. Additionally, OS/2 drivers may support device monitors, programs that monitor data as it is passed to and from the driver. Fortunately, OS/2 offers a wide range of system services called Device Helper routines, or DevHlps, to provide this functionality.

Tools of the Trade

Designing an OS/2 device driver requires a thorough understanding of the role of a device driver, as well as a solid working knowledge of the OS/2 operating system and design philosophy. Debugging OS/2 drivers can be difficult, even with the proper tools. The OS/2 device driver operates at ring 0 with full access to the system hardware. However, it has almost no access to OS/2 support services, except a handful of DevHlp routines. Many driver failures occur in a real-time context, such as in the midst of interrupt handling. It may be difficult or impossible to find a driver problem using normal debugging techniques. In such cases, it is necessary to visualize the operation of the device driver and OS/2 at the time of the error to help locate the problem.

The most important tool for driver development is the driver debugger. Generally, I use the kernel debugger from Microsoft, which comes with the Device Driver Development Toolkit, or DDK. Several other companies offer good driver development tools. A more complete version of this article in book form and a complete C-callable DevHlp library can be purchased from PSS. PentaSoft offers a C-callable interface to the DevHlp routines. OS Technologies offers a driver debugger that is OS/2

version-independent. And FutureWare offers a driver debugger and a C-callable interface to the DevHlp routines.

I write all my device drivers, including the interrupt and timer handlers, in Microsoft C 6.0. A device driver written in C can be written in approximately half the time it would take to write the same driver with the Microsoft Macro Assembler. In special cases, especially when writing drivers for very fast devices or where performance is extremely critical, it only makes sense to write a few subroutines in assembly language. Mostdrivers, however, work fine when written in C.

Anatomy of an OS/2 Device Driver

OS/2 drivers receive requests from the OS/2 kernel. When the driver is originally opened with a DosOpen call, the kernel returns a handle to the program that requested access to the driver. This handle is used for subsequent access to the driver, and the driver name is no longer used (or needed).

When an application makes a call to a driver, the kernel intercepts the call and formats the driver request in a standard driver data structure, called the request packet. The request packet contains the data and pointers that the driver uses to honor the request. In the case of a DosRead or DosWrite, for example, the request packet contains the physical address of the caller's buffer. In the case of an I/O control operation (IOCtl), the request packet contains the virtual address of a data and parameter buffer. Depending on the request, the data in the request packet will change, but the length and format of the request packet's header remain constant. The kernel passes the driver a bimodal pointer to the request packet. This bimodal, or tiled, address is a pointer valid in either protected mode or real mode, because the processor may be in either mode when the driver is called.

How does the kernel know which driver to send the request to? Drivers are loaded by the OS/2 initialization code at boot time, and the kernel keeps a list of the installed drivers by name. Before a driver is used, it must be DosOpened from the application. The DosOpen specifies an ASCII-Z string with the device name as a parameter. The kernel compares this name with its list of installed drivers, and if it finds the name, it calls the Open section of the driver Strategy section to open the device. If that operation succeeds, the kernel returns a handle to the application to use for future driver access. The ASCII-Z name is never used again while the device remains open.

The device handles are usually assigned sequentially, starting with 3 (0, 1, and 2 are claimed by OS/2). However, the handle value should never be assumed. The ASCII-Z device name is located in the device driver header.

The OS/2 Request Packet

An OS/2 device driver consists of a Strategy section and optional Interrupt and Timer sections. The Strategy section receives requests from the kernel in the form of a request packet. The Strategy section verifies the request and, if possible, completes the request and sends the result back to the kernel. If the request cannot be completed immediately, the driver optionally queues up the request to be completed at a later time and starts the I/O operation if necessary. The kernel calls the Strategy section directly by finding its offset address in the device header.

The first entry in the request packet is the request-packet length, filled in by the kernel. The second parameter is the unit code. When a driver supports multiple logical units, the value stored here selects among them. The third field is the command code. The command code is filled in by the kernel. This is the code used by the switch statement in the Strategy section to decode the type of request from the kernel. The next field is the status word returned to the kernel. This field will contain the result of the driver operation along with the Done bit to notify the kernel the request is complete (this is not always the case; the driver may return without the Done bit set). To make things easier, I use a union to access specific types of requests and place the request-packet structures in an include file.

Building the Device Header

A simple OS/2 device driver consists of one code segment and one data segment, although more memory can be allocated if necessary (by means of DevHlp routines). The first data that appears in the data segment must be the device-driver header.

The device-driver header is a fixed-length, link-list structure that contains information for use by the kernel during INIT and normal operation. The first entry in the header is a link pointer to the next device the driver supports. If no other devices are supported, the pointer is set to -1L. This terminates the list of devices supported by this driver. If the driver supports multiple devices, such as a four-port serial board or multiple-disk controller, the link is a far pointer to the next device header.

The next entry in the device header is the attribute word, followed by a one-word offset to the driver Strategy section. Only the offset is necessary, because the driver is written in the small model with a 64-kilobyte code segment and a 64-KB data segment (this is not always true; in special cases, the driver can allocate more code and data space if needed).

The succeeding entry is an offset address to an interdriver communications routine if the driver supports IDC. (The DAW_IDC bit in the device attribute word must also be set; otherwise, the AttachDD call from the other driver will fail.)

The last field is the device name, which must be eight characters in length. Names with fewer than eight characters must be padded with blanks. Remember, any mistake in coding the device-driver header will cause an immediate crash and burn when booting.

Providing a Register Interface to the C Driver

OS/2 device drivers are normally written in C, using the small model, which means 64 KB of data and 64 KB of code (code and data space may be increased in special cases). The driver .SYS file must load the data segment before the code segment. When you write an OS/2 driver in C, you must provide a mechanism for putting the code and data segments in the proper order, and you must also provide a low-level interface to handle device and timer interrupts. Because the device header must be the first item that appears in the data segment, you have to prevent the C compiler from inserting the C start-up code before the device header. You may also have to provide a method of detecting which device is being requested for drivers that support multiple devices. The small assembly language program in listing 4 takes care of these requirements. The _acrtused entry point prevents the C start-up code from being inserted before the driver data segment. The segment-ordering directives ensure that the data segment precedes the code segment.

Note the _STRAT entry point. How does this get called? Remember, this is the address that is placed in the driver's data-segment device header. The kernel, when making a request to the driver, looks up this address in the device header and makes a far call to it. The assembly language routine then calls the C mainline. Thus, the linkage from the kernel to the driver is established.

Why is there a push 0 at the beginning of the _STRAT routine? That's

the device number. Each device supported by the device driver requires a separate device header, and each device header contains an offset address to its own Strategy section. Using the assembly language interface, the routine pushes the device number on the stack and passes it to the driver Strategy section for service.

The Strategy Section

The Strategy section is nothing more than a big switch statement. Common driver requests, such as DosWrite and DosRead, have standard function and return codes. The driver may ignore any or all of these requests by returning a Done status to the kernel. This tells the kernel that the request has been completed. The status returned to the kernel can also include error information that the kernel returns to the calling program.

Note that in the case of a standard driver function, the kernel will map the error value returned from the driver to one of the standard return codes. It is therefore impossible to pass any special return codes to the application via a standard driver request. If you attempt to do so, the kernel will intercept the special return code and map it to one of the standard return codes. The only way to return a special code to the application is by means of an IOCtl request. IOCtls are used for special driver-defined operations (e.g., port I/O). IOCtls are accessed when the application issues a DosDevIOCtl call with the driver's handle. This flexibility allows the driver writer to customize the device driver to fit any device. For instance, if you had a serial driver that monitored bus traffic and reported the occurrence of one or more special characters, you could use an IOCtl read and pass back the character in the return code.

Listing 5 shows the skeleton of a Strategy section. Note the switch on the request-packet command. Several standard driver functions have command codes predefined in OS/2. The driver writer can act on or ignore any of the requests to the driver. Although it would not make sense, the driver could ignore the Open command, issued by the kernel in response to a DosOpen call. Or, more logically, the driver can refuse to be deinstalled by rejecting a Deinstall request.

The INIT call is made only once, during system loading in response to a DEVICE= in CONFIG.SYS. The call is made in the INIT mode from ring 3, but with I/O privileges. The INIT routine is where you would insert the code to initialize your device, such as configuring a UART or sending a disk to track 0.

The very first thing you must do in the initialization code is to save the DevHlp entry-point address in the driver's data segment. This is the only time the address is valid. It must be saved, or it is lost forever. The address of the DevHlp entry point is passed in the INIT request packet. The initialization code performs two other functions. First, it issues the sign-on message to the screen that the driver is attempting to load. Second, it finds the segment address of the last data and last code item, and it sends them back to OS/2. OS/2 uses the code- and data-segment values to size memory. If a driver fails installation, it must send back zeroes for the CS and DS registers so that OS/2 can use the memory space it occupied.

One of the most common techniques in OS/2 driver design is for the Strategy section to request service from the device and wait for a device or timer interrupt to signal completion of the request. The fragment in listing 6 shows an implementation of this scheme for the Read function of my sample serial communications driver. In this case, the Strategy section starts the I/O and issues a Block DevHlp call, which blocks the calling thread. When the device interrupt signals that the operation is done, the interrupt section runs the blocked thread, completing the request. To protect against the request's never being completed (e.g., in the case of a down device), the Block call can contain a time-out parameter. If the time expires before the completion interrupt occurs, the Strategy section can send the proper error back to the kernel.

Another way to time-out a device is to use the SetTimer DevHlp routine. You can attach a timer handler to the OS/2 system clock and have the handler run the blocked thread after a specified number of ticks.

The commands allowed by the Strategy section are up to the device driver writer. You can process only the commands you wish to act on and let the others simply pass by sending a Done status back to the kernel. You may instead wish to trap the illegal function calls and return an ERROR_BAD_COMMAND message to the kernel. Keep in mind, however, that the kernel frequently issues its own commands to the driver without your knowledge. For example, when the user of the application that opened the driver types a Control-C, the kernel checks the application's list of open drivers and issues a Close request to each one. In general, I've found it easier to ignore all the requests I'm not waiting for and just flag them as done.

In the simplest of drivers, the Strategy section can only contain an

Open, Close, and Read or Write request. In a complicated driver, such as a disk driver, the Strategy section may contain over two dozen standard driver functions and several additional IOCtl calls. IOCtl calls are actually Strategy functions, but they are broken down one step further to provide more detailed or device-specific operations. For instance, a driver might send a list of parameters to an I/O port to initialize it and return the input value of a status port with the status of the initialization.

A Sampler of Standard Driver Functions

INIT (code 0x00). This function is called by the kernel during driver installation at boot time. The INIT section should initialize your device, such as setting the baud rate, parity, stop bits, and so forth on a serial port or checking to see if the device is installed by issuing a status request to the device controller. This INIT function is called in a special mode in ring 3 with some ring 0 capabilities.

The driver may turn off interrupts, but they must be turned back on before returning to the kernel. The INIT code may perform direct port I/O without protection violations. Usually, the driver writer will allocate buffers and data storage during initialization, to be sure the driver will work when installed. Because the initialization is being performed in ring 3, the system can check to make sure the buffer and storage allocations are valid and the segments are owned by the driver. If not, the driver can remove itself from memory, freeing up any previously allocated space for other system components or another driver. Because initialization is done only once during system boot-up, it is not critical to optimize the section. Do all your initializations here, as it may be time-prohibitive or even impossible to do initialization during normal driver operation.

Media Check (code 0x01). This function is called by the kernel prior to disk access, and it is therefore valid only for block devices. The kernel passes the driver the media ID byte corresponding to the type of disk it expects to find in the selected drive.

BuildBPB (code 0x02). When the block driver gets a Build Bios Parameter Block call, it must return a pointer to the BPB that describes the mass-storage device.

Read (code 0x04). The application calls the Read section by issuing a DosRead with the handle obtained during the DosOpen. The Read routine may return one character at a time, but more often it returns a buffer full of data. How the Read function works is up to the driver writer.

The driver returns the count of characters read and stores the received data in the data segment of the application. Read returns a standard driver return code.

Nondestructive Read (code 0x05). In response to this request, the driver must get the first character in the driver buffer and return it to the caller. If no character is present, the driver must return immediately with the proper error bits and Done bit set.

Input Status (code 0x06). The driver must clear the Busy bit in the request packet if one or more characters are in the driver's buffer, or set it if no characters are present. This is a Peek function to determine the presence of data.

Flush Input Buffer(s) (code 0x07). This function should flush any receiver queues or buffers and return a Done status to the kernel.

Write (code 0x08). This is a standard driver request called by the application as a result of a DosWrite call. The application passes to the driver the address of data to write (usually in the application's data segment) and the count of characters to write. The driver writes the data and returns the status to the application along with the number of characters that were actually written. Write returns a standard driver return code.

Write with Verify (code 0x09). The driver writes data as in the Write function code above, but it verifies that the data was written correctly.

Output Status (code 0x0a). The driver must set the Busy bit in the request packet if an operation is in progress, or clear it if the transmitter is free.

Output Flush (code 0x0b). The driver must flush the output queues and buffers and return a Done status to the kernel.

Device Open (code 0x0d). This function is called as a result of the application issuing a DosOpen call. The kernel makes note of the DosOpen request, and if it is successful (done with no errors) the kernel sends back a handle to the application to use for subsequent driver service. The driver writer can use this section to initialize a device, flush any buffers, reset the buffer pointer, initialize the character queues, or anything necessary for a clean starting

operation.

Device Close (code 0x0e). This function is called as a result of the application doing a DosClose with the correct driver handle. It's a good idea to make sure the application closing the driver is the same one that opened it, so save the process ID of the application that opened the driver and make sure the closing PID is the same. If not, reject it as a bogus request. You should make all your devices quiescent at this time.

Removable Media (code 0x0f). The driver receives this request when an application generates an IOCtl call to category 8, function 0x20. Instead of calling the IOCtl, the kernel issues this request. The driver must set the Busy bit of the request-packet status if the media is nonremovable, or clear it if it is removable.

Generic IOCtl (code 0x10). This is a special type of function call. It is very flexible, as the data passed to the driver is stored in two buffers owned by the caller. These buffers may contain any type of data; the format is up to the driver writer.

The first and second parameters of an IOCtl are the address of the application program's data buffer and parameter buffer, respectively. The parameter buffer might contain a list of USHORTs, UCHARs, or pointers. The data buffer parameter might be a data buffer address in the application program, where the driver would store data from the device.

IOCtls can extend the range of status information that drivers can convey to applications. Suppose, for example, a driver needed to report to an application that the data was in ASCII or binary format, or that a parity error was detected while receiving it. Here an IOCtl would be the answer. The reason? The kernel massages return codes from standard function calls to fit within the standard error definitions. The IOCtl, however, will pass back codes to the application exactly as they were set in the driver. In several drivers that I have written, the DosRead and DosWrite sections of the Strategy routine are commented out and never used. I use IOCtls for the reads and writes to allow the driver to communicate directly with the application without interference from the kernel.

PrepareForSysShutdown. This function tells the device driver it should post any open buffers to their devices before the system powers down.

This occurs when you select Shutdown from the Desktop window.

The Interrupt Section

When OS/2 calls your interrupt handler, it does so with interrupts disabled, so any extended time spent in the interrupt handler could cause performance problems. When activated in response to the receipt of data, the interrupt handler must store the data and exit quickly. In the case of character devices, the OS/2 DevHlp library supports fast reads and writes to circular character queues. For block devices, interrupt handling is fast because the interrupt is usually caused by a DMA completion or disk seek completion. For block devices, data is ordinarily transferred to the user buffer using DMA, eliminating the need to transfer data during the interrupt processing. On a DMA transfer, the driver can exit once the DMA controller starts so that other threads can run. When the DMA completes, it generates a DMA completion interrupt that activates the driver's interrupt handler.

The interrupt handler routine is not difficult to write or understand, but it can be very difficult to debug. Errors that occur in the interrupt handler frequently appear only in a real-time context, when the interrupt handler is active in response to a hardware interrupt. You can't do a printf() from the interrupt routine or inspect variables with an application debugger, such as CodeView. You must use the OS/2 KDB (Kernel Debugger) supplied with the DDK or a similar debugger. Even with the KDB, a breakpoint will halt the program, and further interrupts may pass undetected while you decide what to type next. Because of this pause in execution, you lose the real-time context of the program, which may be the root of the original problem. In the end, there's no substitute for the ability to visualize the correct operation of the interrupt handler.

The Timer Handler

In an OS/2 driver, you can hook the system timer interrupt with a call to the DevHlp library SetTimer function. You pass OS/2 a near pointer to your timer handler, and for each system timer tick, OS/2 calls your timer handler routine and any other timer handler that had been previously registered.

If no data appears within one or two 32-millisecond time ticks, the driver assumes that data input has stopped or at least paused. If a valid Read request is pending, it sends back the data to the blocked Strategy section by issuing a Run request with the same ID used to block the requesting thread. The Strategy section becomes unblocked, gets the data from the receiver queue, and sends the data to the

application's data buffer.

Do You Really Need a Device Driver?
Maybe not. OS/2 1.x allows programs with I/O Privilege (IOPL) enabled to do direct register I/O to a device. If the device is a parallel card or digital switch, a driver may not be necessary. You can set or clear bits using IN and OUT instructions, and as long as the device is not time critical, such a method will be sufficient.

Yet devices that generate interrupts, require asynchronous service, or operate in a time-critical environment must use a device driver. Take a serial device, for example. It would be difficult or impossible to read data from the device using the IOPL method. By definition, asynchronous data may come in at any time. Because OS/2 may be running another thread at the time the data appears, your chances of missing data are excellent. But an interrupt driver could continue to read and buffer the incoming data until the OS/2 scheduler ran your thread.

Optionally, you can allow interrupts to preempt the current running thread and run your thread immediately. You need not wait for the scheduler to run it. This sort of preemptive multitasking sets OS/2 apart from other multitasking systems, like Unix. In Unix, the currently running program retains the CPU until it exhausts its time slice. It cannot be preempted based on an event, such as a device interrupt. That's why OS/2 is my choice for time-critical applications.

Steven J. Mastrianni is an independent consultant in South Windsor, Connecticut, who specializes in OS/2 device drivers. You can contact him on BIX as "smastrianni."

COMPANIES MENTIONED

FutureWare, Inc.
78 Temple Ave., Suite 15
Hackensack, NJ 07601
(201) 343-3921

Microsoft Corp.
One Microsoft Way
Redmond, WA
(800) 227-6444

OS Technologies
532 Longley Rd.
Groton, MA 01450
(508) 448-9653

PentaSoft
17541 Stone Ave. N
Seattle, WA 98133
(206) 546-0470

PSS Corp.
290 Brookfield St.
South Windsor, CT 06074
(203) 644-4764