Device Control

Writing MS-DOS Device Drivers

Marcus Johnson

Marcus Johnson received his B.S. in math from the University of Florida in 1978 and is currently Senior System Engineer for Precision Software in Clearwater, Florida. You may reach him at 6258 99th Circle, Pinellas Park, Florida.

Introduction

This article describes, from my personal experience, the joys of writing MS-DOS device drivers in C. A device driver is the executable code through which the operating system can communicate with an I/O device.

Many of the device drivers you use on your MS-DOS system are already part of the operating system: the basic keyboard/screen (console) driver, the floppy and hard disk drivers, the serial (COM) port driver, and the printer port driver.

The drivers that I have written include a RAM disk driver and an ANSI console driver. They have been compiled under Microsoft C v4.0 and assembled under Microsoft MASM v5.1. The executable binaries were created with Microsoft Link v3.64. Certain modifications will have to be made in order for this to compile under Turbo C.

I wrote much of these drivers in C partly as an exercise and partly to make the code easier to write, understand, and extend. For sheer speed, assembly language is still better. But if you aren't that comfortable in assembly language, what better starting point than the relatively clean, documented, *correct* code produced by your compiler?

The significance of *installable* device drivers, such as provided under MS-DOS, is that you can interface a device to your system that was not originally part of it. The relative ease with which you can write a device driver has led to the proliferation of low-cost peripherals in the MS-DOS environment. Once written, these drivers are installed by simply creating (or adding to) an ASCII text file called *config. sys* in the root directory on your boot disk. For each device, the *config.sys* file contains a line that reads

```
device=filename [options]
```

where filename is the name of the file containing your device driver, and *[options]* are optional instructions for your device driver. Well-known examples of standard drivers include *ansi.sys*, the console driver that allows certain common ANSI escape sequences to be properly interpreted on the screen, and *vdisk.sys*, the disk driver that lets you keep files in RAM.

ansi.sys and *vdisk.sys* represent two of the three device driver types. *ansi.sys* is a driver of the first type, a *character* device driver. It is intended to handle a few bytes of data at a time, and can handle single bytes. *vdisk.sys* is a driver of the second type, a *block* device driver. It handles data in chunks whose units are called blocks or sectors.

The third type, a *clock* device driver, is actually a modified character device driver. It is easy to write. I have not provided an example since I do not have clock hardware to test it with.

Device Driver Format

Device drivers must rigorously follow a specific plan. Each must include a header, a strategy routine, an interrupt routine, and a set of command code routines. The device driver is typically a memory image file, like a *.com* file. The main difference between a device driver and a *.com* file is that the *.com* file starts at offset *0x0100* and the device driver starts at *0x0000*.

The device header is the first part of the file. It contains the following fields:

- Link to next driver in the file (2-byte offset plus 2-byte segment)
- Device attributes (2-byte word)
- Strategy routine (2-byte offset)
- Interrupt routine (2-byte offset)

The header for a character device driver is followed by an 8-byte logical name such as *PRN, CON, or COM1*. This is the name by which the device is known to the system. You use it exactly as you would any other named device. The header for a block device driver is followed by a byte containing the number of units controlled, followed by seven null bytes.

Note that there can be many device drivers within one file, with each driver pointing to the next. The last driver in the file uses *0xFFFF* for the offset and segment of the link to the next driver. Thus, when there is only one device per file, as in my drivers, the link is simply a double word *0xFFFFFFFF*.

The device attributes word contains the following fields:

- bit 15: set if character device, clear if block device
- bit 14: set if I/O control supported
- bit 13: for a block device, set if not IBM format; for a character device, set if output-until-busy call supported
- bit 12: reserved
- bit 11: set if open/close/removable media calls supported
- bits 5-10: reserved

- bit 4: set if *CON* driver
- bit 3: set if current clock device
- bit 2: set if current *NUL* device
- bit 1: set if current standard output device
- bit 0: set if current standard input device

Reserved bits should be zero. Bit 11 has meaning only to block devices and only under MS-DOS version 3 and up. Bits 0-4 only have meaning to character devices.

Bit 4 is an oddity. It is referenced in Ray Duncan's *Advanced MS-DOS* as both a reserved bit and as "special *CON* driver bit, *INT 29H*." Apparently, MS-DOS uses *INT 29H* to output characters via the *CON* driver. It was not until I set the bit and put a replacement for *INT 29H* in my code that my console device driver would work. (A quick tour of my system via *DEBUG* showed that the unadulterated *INT 29H* simply outputs a character in *AL* through the TTY function (*OEH*) of *INT 10H*.)

The strategy routine is a curiosity that, according to Duncan, has no real functionality under the single-user single-tasking MS-DOS we all know, but would have some utility in a multi-user multitasking environment. Its job is to store the request header address, which is in the register pair *ES:BX* on an I/O request.

This request header is the means by which MS-DOS communicates with your device driver. The first 13 bytes of each request header are the same. Later bytes differ depending on the nature of the command. The common portion of the request header contains the following fields:

- Byte 0: Length of the request header
- Byte 1: Unit number (which drive)
- Byte 2: Command code
- Bytes 3-4: Driver's return status word
- Bytes 5-12: Reserved

The command code is used by the interrupt routine to determine which command to execute. The status word is used by the interrupt routine to give back status to MS-DOS. It contains the following fields:

- Bit 15: Set on error
- Bits 10-14: Reserved, should be zero
- Bit 9: Set if busy
- Bit 8: Set if done
- Bits 0-7: Error code if bit 15 set

The error codes returned are:

- 0: Write-protect violation
- 1: Unknown unit
- 2: Drive not ready
- 3: Unknown command
- 4: Data error (bad CRC)
- 5: Bad drive request structure length

- 6: Seek error
- 7: Unknown medium
- 8: Block not found
- 9: Printer out of paper
- 10: Write fault
- 11: Read fault
- 12: General failure
- 13: Reserved
- 14: Reserved
- 15: Invalid media change (MS-DOS versions 3 and up)

Command Code Routines

MS-DOS makes the **driver initialization call** (command code 0) only to install the device driver after the system is booted. It is never called again. Accordingly, it is a common practice among writers of device drivers to place it physically at the end of the device driver code, where it can be abandoned. Its function is to perform any hardware initialization needed. The request header for this command code includes the following additional fields:

- Byte 13: Return number of units initialized
- Bytes 14-17: Return break address (last address in driver)
- Bytes 18-21: Pointer to the character on the *config.sys* line following the "*device*=" (block devices return a 4-byte pointer to the BIOS parameter block array here)
- Byte 22: Drive number (A=0, B=1, etc.) for the first unit of the block driver (MS-DOS version 3 and up)

The BIOS parameter block array contains 2-byte offsets to BIOS parameter blocks, one for each unit supported. The BIOS parameter block describes pertinent information to MS-DOS about each unit controlled. It contains the following fields:

- Bytes 0-1: number of bytes per block
- Byte 2: blocks per allocation unit (must be a power of 2)
- Bytes 3-4: number of reserved blocks (beginning with block 0)
- Byte 5: number of file allocation tables
- Bytes 6-7: maximum number of root directory entries
- Bytes 8-9: total number of blocks
- Byte 10: media descriptor byte
- Bytes 11-12: number of blocks occupied by a single file allocation table

The media descriptor byte describes to MS-DOS what kind of media is in use. The following codes are valid for IBM-format devices:

- 0xF8 fixed disk
- 0xF9 double sided, 15 sectors
- 0xFC single sided, 9 sectors
- 0xFD double sided, 9 sectors
- 0xFE single sided, 8 sectors
- 0xFF double sided, 8 sectors

The **media-check call** (command code 1) is useful for block devices only. (Character devices should simply return *DONE*. I will not repeat this warning for other command codes that you use with only one type of device.) MS-DOS makes this call to determine whether or not the media has been changed. The request header for this command code includes the following additional fields:

- Byte 13: Media descriptor byte, set by MS-DOS
- Byte 14: Media change code, returned by function (-1: media has been changed, 0: don't know whether media has changed, 1: media has not been changed)
- Bytes 15-18: 4-byte pointer to previous volume ID (if open/close/removable media bit in device attributes word was set, the media has been changed, and we're running MS-DOS version 3 or higher)

If we're using a hard disk or a RAM disk, we know that the media cannot be changed, and we always return 1. If the media descriptor byte has changed (a copy of the BIOS parameter block can be found at offset 3 into block 0 of the media, if the format is IBM), or if the volume label has changed (checked under MS-DOS version 3 and up), then we know the media has changed, and we return *-1*. If the media descriptor byte and the volume label match, we don't really know (how many unlabelled disks, identically formatted, do *you* have?), and we return *0*.

The **build-BIOS-parameter-block call** (command code 2) is useful only to block device drivers. MS-DOS makes this call when the media has been legally changed. (Either the media check call has returned "media changed" or it returned "don't know," and there are no buffers to be written to the media.) The routine returns a BIOS parameter block describing the media. Under MS-DOS version 3 and up, it also reads the volume label and saves it. The request header for this command code includes the following additional fields:

- Byte 13: the old media descriptor byte (from MS-DOS)
- Bytes 14-17: a 4-byte pointer to a buffer containing the first file allocation table block. If the non-IBM format bit in the device attributes word is zero, this should not be altered by the driver; otherwise, it may be used as scratch space by the driver. I have no idea what purpose this serves.
- Bytes 18-21: a 4-byte pointer to the new BIOS parameter block, returned by the driver

MS-DOS performs the **I/O-control-read-call** (command code 3) only if the I/O-control bit is set in the device attributes word. It allows application programs to access control information from the driver (what baud rate, etc.). The request header for this command code includes the following additional fields:

- Byte 13: media descriptor byte from MS-DOS
- Bytes 14-17: 4-byte pointer to where to write the information

- Bytes 18–19: count of bytes or blocks to be written; on return, the count of bytes or blocks written.
- Bytes 20–21: the starting block number (block devices only)

The **read call** (command code 4) transfers data from the device to a memory buffer. If an error occurs, the handler must return an error code and report the number of bytes or blocks successfully transferred. The request header for this command code includes the following additional fields:

- Byte 13: media descriptor byte from MS–DOS
- Bytes 14–17: 4–byte pointer to where to write the information
- Bytes 18–19: count of bytes or blocks to be read; on return, count of bytes or blocks successfully read
- Bytes 20–21: starting block number (block devices)
- Bytes 22–25: 4–byte pointer to volume label if error 15 (invalid media change) reported (MS–DOS version 3 and up)

The **non–destructive–read call** (command code 5) is valid only for character devices. Its purpose is to allow MS–DOS to look ahead one character without removing the character from the input buffer. The request header for this command code includes the following additional field:

- Byte 13: the character

The **input–status call** (command code 6) is valid only for character devices. Its purpose is to tell MS–DOS whether or not there are characters in the input buffer. It does so by setting the busy bit in the returned status to indicate if the buffer is empty. An unbuffered character device should return a clear busy bit; otherwise, MS–DOS will hang up, waiting for data in a nonexistent buffer! This call uses no additional fields.

The **flush–input–buffers call** (command code 7) is valid only for character devices. If the device supports buffered input, it should discard the characters in the buffer. This call uses no additional fields.

The **write call** (command code 8) transfers data from the specified memory buffer to the device. If an error occurs, it must return an error code and report the number of bytes or blocks successfully transferred. The request header for this command code includes the following additional fields:

- Byte 13: media descriptor byte from MS–DOS
- Bytes 14–17: 4–byte pointer to where to read the information
- Bytes 18–19: count of bytes or blocks to be written; on return, count of bytes or blocks successfully written
- Bytes 20–21: starting block number (block devices)
- Bytes 22–25: 4–byte pointer to volume label if error 15 (invalid media change) reported (MS–DOS version 3 and up)

The **write–with–verify call** (command code 9) is identical to the write call, except that a read–after–write verify is performed, if possible.

The **output-status call** (command code 10) is used only on character devices. Its purpose is to inform MS-DOS whether the next write request will have to wait for the previous request to complete by returning the busy bit set. This call uses no additional fields.

The **flush-output-buffers call** (command code 11) is used only on character devices. If the output is buffered, the driver should discard the data in the buffer. This call uses no additional fields.

MS-DOS makes the **I/O-control-write call** (command code 12) only if the I/O-control bit is set in the device attributes word. It allows application programs to pass control information to the driver (what baud rate, etc.). The request header for this command code includes the following additional fields:

- Byte 13: media descriptor byte from MS-DOS
- Bytes 14-17: 4-byte pointer to where to read the information
- Bytes 18-19: a count of bytes or blocks to be read; on return, the count of bytes or blocks read
- Bytes 20-21: the starting block number (block devices only)

The **open call** (command code 13) is available only for MS-DOS version 3 and up. MS-DOS makes this call only if the open/close/removable media bit is set in the device attributes word. This call can be used to tell a character device to send an initializing control string, as to a printer. It can be used on block devices to control local buffering schemes. Note that the predefined handles for the CON, AUX, and PRN devices are always open. This call uses no additional fields.

The **close call** (command code 14) is available only for MS-DOS version 3 and up. MS-DOS makes this call only if the open/close/removable media bit is set in the device attributes word. This call can be used to tell a character device to send a terminating control string, as to a printer. It can be used on block devices to control local buffering schemes. Note that the predefined handles for the CON, AUX, and PRN devices are never closed. This call uses no additional fields.

The **removable-media call** (command code 15) is available only for MS-DOS version 3 and up, and only for block devices where the open/close/removable media bit is set in the device attributes word. If the media is removable, the function returns the busy bit set. This call uses no additional fields.

The **output-until-busy call** (command code 16) is available only for MS-DOS version 3 and up, and is called only if the output-until-busy bit is set in the device attributes word. It only pertains to character devices. This call is an optimization designed for use with print spoolers. It causes data to be written from the specified buffer to the device until the device is busy. It is not an error, therefore, for the driver to report back fewer bytes written than were specified. The request header for this command code includes the following fields after the standard request header:

- Bytes 14–17: 4-byte pointer to the buffer from which data is to be written
- Bytes 18–19: count of bytes to be written; on return, the number of bytes written.

Designing a Device Driver

Designing a device driver is a relatively simple task, since so much of the design is dictated to you. You know that you must have a strategy routine and an interrupt routine that must perform certain well-defined functions. The only real design decisions are how you choose to implement these functions. What tasks must be performed in order to implement the functions? What approaches will you use? Note that some calls only exist under MS-DOS versions 3 and up, or act differently under those versions. Will you use those calls, will you restrict yourself from using them, or (tricky, but best) will you write code that finds out the MS-DOS version and acts accordingly?

Coding the device driver is an entirely different matter, and, except maybe for debugging, the most challenging. Those of us who write C code for a living are not normally concerned with the underlying implementation of our code in machine language. We might employ some tricks we have learned about how C is typically implemented — using shifts to divide or multiply by a power of 2, for example — to get us a bit more speed, but by and large we ignore the machine interface.

In the world of the device driver, you are forced to think about what you're really doing at the machine level. If you look at my code, you'll find that I hardly ever pass parameters from one function to another. I don't use local variables. Everything's done with global variables. Look at *w_putc* in the console driver — it just cries out to be broken down into smaller functions. But it isn't, although it was originally written that way. The reason? You have no stack to speak of, perhaps 40 or 50 bytes. C passes parameters on the stack, two bytes for each word. C also keeps local variables on the stack, two bytes for each word again. Each function call eats up at least four bytes of stack as well. (My C compiler insists on starting every function by pushing the BP register, preparatory to building a stack frame for the local variables, whether or not there are any local variables.). All these contributions add up.

What I ended up doing was learning more assembly language then I ever meant to. In the early stages, I used the *-Fc* flag in my compilations to generate a merged assembly/C listing. That allowed me to examine the code that the compiler generated from the C I had written. In particular, I had to learn about how *far* pointers are implemented to come up with the *(char far * far *)* cast used in the *ansi_init* code to (correctly) load the *INT 29* vector. I learned a few more things, too, but I will discuss those a little later.

Unfortunately, when you're working in a high-level language, you sometimes "can't get there from here." How do you get the compiler to load certain specific registers and then make a BIOS call? What statement generates a return-from-interrupt opcode? You need to preserve the machine state by pushing all the locally-used registers, and then popping them back off the stack when you're done. What function will do that? If your compiler allows in-line assembler code, great. But that's cheating, it isn't standard C. Thus the assembler interface.

I broke the assembly code for the drivers into two files, *main.asm* and *vars.inc,* plus *raw.asm* for the console driver and *bpb.inc* and *rdisk.asm* for the block driver. *raw.asm* performs functions that you just can't do in standard C. It handles all the BIOS calls, the reading and writing of I/O ports, the interrupt handlers. *bpb.inc* defines the standard BIOS parameter block for the RAM disk. *rdisk.asm* sets up the boot block, file allocation table blocks, and the first directory block, complete with clever volume label. *main.asm* handles the startup code. Except for the device header, it is pretty much identical for both drivers. *vars.inc* sets up the global variables used.

vars.inc allocates the variables because my C compiler wants to put them in a segment that gets loaded higher in memory than the program code. This behavior defeats the practice of putting the initialization code physically last and passing its own address back as the end of the driver. Also, the assembly language routines and the C routines could never agree (as I discovered by examining the code with *DEBUG*) as to where the variables were in memory until I put them in the assembly language .*CODE* segment portion.

Other Lessons

In putting the code together, I learned about a few more switches for the compiler that I had never used before, by examining the merged assembly/source files. I didn't want to use any code from the compiler's library. I had no idea what the library code did internally, and I couldn't risk putting unknown code into the drivers. Nor could I afford the additional stack usage. Yet there were calls to a stack-checking routine in every C function. Fortunately, there is a command-line switch to disable such stack probes.

A more serious problem was that my C code was incorrectly pulling fields out of the request header, which I had set up as a structure. The problem was that the compiler aligns the structure fields on *int* boundaries to minimize access time to the fields. Unfortunately, I don't have access to the source code for MS-DOS to make its request header similarly aligned. I did discover, however, that there is yet another command-line switch to force tight packing of structures.

One final trick I had to play was to fool the linker into not loading the C library functions. Even though no reference is made in the source code, the compiler adds to the object file a reference to a routine called *_acrtused*. As it turns out, this is the startup code that processes the DOS command line, initializes the data area for memory allocation, and calls *main*. I could not get rid of the references in the C object, so I named the interrupt routine in *main.asm _acrtused* and made it a public name.

Creating the final executable was simple. Using the Microsoft linker, I simply made sure that *main.obj* is the first file in the command line and that *init.obj* is the last. Object modules are linked together in the order they are found in the command line. The linker complains of no stack segment, as I expected, but this is a warning, not an error. Finally, the executable *main.exe* is converted to *main.bin* by *exe2bin.* The file is now ready for calling in your *config.sys* file.

Debugging the device driver is not simple. In its final form, it is ill-suited for standard debugging tools. Its first bytes, containing the link to the next device in the driver, are not executable. I found that the best way to debug the driver was to test each of the interrupt functions as they were written, attaching stubs to them for testing. Once each of the functions was debugged, I was ready to tie them into the *main.asm* interrupt routine.

As Duncan recommends, I copied the test version onto a floppy and booted from there. For the first three evenings of test, everything I did gave the same result: the drive would be accessed, then everything would get real quiet, with the *A:* drive light shining steadily. Finally, as I explained earlier, I looked at the code with *DEBUG* and discovered the discrepancies between where the strategy routine was placing the pointer to the request header and where the C routines were looking for it. That problem solved, I booted successfully, and the drivers tested out to my specs.

I am deeply indebted to the following sources of knowledge while producing this article: Ray Duncan's Advanced MS-DOS *and Peter Norton*'s The Programmer's Guide to the IBM PC. *These volumes are an indispensable part of my library, and in great danger of falling apart from use*.

Listing 1

Listing 2

Listing 3

Listing 4

Listing 5

Listing 6

Listing 7

Listing 8