Double Duty Directory Display

DDIR is an assembly language utility that displays normal directory information in two columns and alphabetizes the listing.

The more you need DIR to list large numbers of files, the more obvious its basic limitation becomes. By using only 39 display columns per line—a concession to terminals that few, if any, PC owners use—DIR wastes half its 80-column display space and can show you only 25 files at once. The only provision DOS makes for putting more files on the screen is the DIR/W option, and this leaves out the important size, date, and time information.

DDIR, which stands for "Double Directory," is an assembly language utility designed to make your directory listings more useful. It displays all the normal directory information in two columns, thus giving you more directory information per screen. It also automatically alphabetizes the listing and pauses after every full screen.

DDIR.COM is not intended as a wholesale replacement for DIR.COM. Rather, it simply intercepts the normal DIR information on its way to the display, sorts it, and puts it in two columns instead of one. Thus, DDIR will accept any drive, path, or filename parameter that would be accepted by DIR alone. The program requires PC-DOS 2.0 or above and, as you'll see, takes advantage of some of the more esoteric features of DOS.

Getting DDIR

The easiest way to get a copy of DDIR is to download it via modem from the PC Interactive Reader Service at (212) 696-0360. If your communications software supports the Xmodem protocol, you can download DDIR.COM directly. If not, you can still download DDIR.ASC, RE-

1985/No. 20

Name it as DDIR.BAS, load it into BASICA, and run it. This will create the DDIR.COM file ready for use.

If you have the IBM Macro Assembler, you may prefer instead to download DDIR.ASM and assemble it yourself. It's in the .COM file format, so you'll have to run EXE2BIN after LINK. Even if you don't have the Macro Assembler, however, getting the DDIR.ASM file will be helpful in following the discussion of the program below.

The Big Picture

To understand how DDIR operates, it's a good idea to step back and look at the big PC-DOS picture. Any disk formatted with the /S option contains the three files that comprise PC-DOS. Two of these are hidden files; the third, COMMAND.COM, is visible in a directory listing.

The two hidden files are called IBMBIO.COM and IBMDOS.COM. In PC-DOS 2.0 and above, IBMBIO is basically a series of device drivers. These drivers provide a means for PC-DOS to communicate with the hardware of the PC, including the display, the keyboard, the disk drives, and the printer. In many cases, these IBMBIO.COM device drivers use the ROM BIOS interrupts.

IBMDOS.COM contains the code needed to execute the DOS function calls. You use IBMDOS.COM every time you issue an Interrupt 21h in an assembly language program. If a particular function call needs to use a hardware device, it then calls a device driver routine in IBMBIO

.COM. Thus, there is a hierarchical structure to DOS: programs issue DOS calls to IBMDOS.COM, IBMDOS.COM issues device driver calls to IBMBIO.COM, the device drivers issue interrupts to the ROM BIOS, and the ROM BIOS talks to the hardware.

So where does COMMAND.COM fit in? COMMAND.COM is the program that is running when no other program is running. It is COMMAND.COM that asks for the date and time when you boot up, displays the DOS prompt, reads in what you type at the DOS prompt, and searches to see if what you typed in is an internal command (DIR, COPY, ERASE, etc.). If so, it executes that command, often using lots of Interrupt 21h DOS function calls. If the typed command is not an internal command, then COMMAND.COM searches the current directory for a .COM, .EXE, or .BAT file of that name, and then uses any directory paths set by PATH to do further searches. It is COMMAND.COM that loads in the file, and it is COMMAND.COM that takes over when the program is done.

COMMAND.COM is also responsible for executing batch file programs, including AUTOEXEC.BAT, when DOS is first loaded. COMMAND.COM also takes over during critical hardware errors and issues the much-loved "Abort, Retry, Ignore?" message.

As if to complicate matters, COMMAND.COM divides itself into two pieces when it is first loaded into memory. The "resident" part of COMMAND .COM, about 3K bytes, sits in the lower end of memory above the other two DOS files. The "transient" part of COMMAND.COM—the bulk of the program—resides up at the very top of user memory.

Why the split? The reason is that the transient part of COMMAND.COM interprets and executes the DOS internal commands and does batch file processing. These facilities are not needed when other programs are running. So, by putting itself at the top of memory, the transient COMMAND.COM allows itself to be overwritten by other programs if they need the space.

When a program exits, control is returned to the resident part of COMMAND .COM. This then does a simple sum check of the memory area normally occupied by the transient COMMAND.COM. If it doesn't check, then the transient part was at least partly overwritten and is now invalid. Finding this, the resident part reloads the transient COMMAND.COM back into memory. This reloading is the reason that an exit from a large program often causes a disk access. Some programs—like Lotus's *1-2-3* and many compilers—always use the top area, so reloading the transient COMMAND.COM is always necessary.

I mentioned earlier that our double-column directory program DDIR will invoke the DIR command, which is located in the transient part of COMMAND.COM. But when running another program (such as DDIR), the transient COMMAND.COM cannot be trusted to be valid. So how can we proceed?

Loading Secondary Command Processors

If you search hard in the PC-DOS manuals for Versions 2.0 and above, you will find a short, 1½-page section on loading "secondary command processors." Look in Appendix F in the DOS 2.0 manual, page 1-11 in the DOS 2.1 manual, chapter 7 of the DOS 2.1 technical reference manual (which omits the crucial /c parameter), and chapter 7 of the DOS 3.0 technical reference manual (which gets it right).

What these sources say is that you can execute any DOS internal command (or any .COM, .EXE, or .BAT program for that matter) from within an assembly language program by loading in a secondary copy of COMMAND.COM and passing to it a parameter containing the command or program name. COMMAND.COM will then do all the messy stuff and return control back to your program.

Thus, you can actually load in a secondary version of COMMAND.COM right on the DOS command level. Compare the size of the available memory reported by a simple CHKDSK with that reported by

```
COMMAND /c CHKDSK
```

The difference is the amount of memory taken up by the second resident portion of COMMAND.COM. When a /c and a command are given after COMMAND, as in this example, COMMAND.COM then executes the parameter (CHKDSK). It then terminates, just like any other program, by returning control to the previously executing program—which in this case is the primary COMMAND.COM.

Loading a second version of COMMAND.COM from within an assembly language program is somewhat complicated, but since you may need to do it with programs of your own, it's worth explaining. There are several steps involved.

First, you must make sure there is enough memory for DOS to load the second COMMAND.COM. When a program first begins executing, all available memory is allocated to it. Thus, some of this memory must be freed up. Before the memory is freed up, the stack pointer may have to be moved from the area of memory being freed, so the stack doesn't get destroyed.

Second, you must determine where COMMAND.COM is currently stored. On a two-floppy system this is generally drive A:; on a PC-XT it will generally be drive C:; on my system, on the other hand, it's in the subdirectory DOS31 of drive D: of my Bernoulli Box. In other words, you can't just guess where COMMAND.COM actually resides. It may be different on different systems.

This is where the "environment" provided by DOS 2.0 and above comes in handy. In DOS 2.0 and above, programs have access to an environment, which is an area in memory with strings that show the current DOS prompt, any path you may have set with the PATH command, and the drive, directory, and filename of your current command processor.

If you'd like to see your current environment, just issue the SET command. One of the lines printed out will begin with "COMSPEC=" and will show the drive, directory, and filename of the command processor COMMAND.COM. Any program can get at its environment by accessing the memory beginning at the segment address stored in offset 002Ch of the Program Segment Prefix.

When your program finds where COMMAND.COM is stored, it can perform a PC-DOS EXEC call, namely, function call 4Bh of Interrupt 21h. The EXEC call can load in any .COM or .EXE file and execute it, with minimum fuss by the programmer. It requires a block that contains parameters to pass to the program, but that's it. In setting up DDIR, the parameters to pass with the EXEC call will be the DIR command we want COMMAND.COM to execute. And, of course, when we tell COMMAND.COM to do its normal DIR, we will be watching for screen output so we can sort the listing, put it in an economical two columns instead of the wasteful one, and pause after every full screen.

From Theory to Practice

From this point on you'll find it helpful if you have obtained (by modem or mail) a copy of the actual assembly language listing DDIR.ASM. That way, in addition to following the steps involved, you can see how they are translated into workable code.

As a first step, memory to accommodate 528 files (11 screens of 48 files each) is blanked out, and the stack pointer is set to 256 bytes beyond that area. Then the DOS SETBLOCK function call 4Ah is issued, which shrinks down the memory allocated for ES (i.e., the DDIR program) to BX paragraphs. Now there should be plenty of room for COMMAND.COM.

Next the environment must be searched for the drive and directory of the command processor, COMMAND.COM. The segment address of the environment is located in offset 002Ch of the program segment prefix, and ES must be set to that segment.

DI is set to point to the beginning of the environment. When the "COMSPEC=" is found, it is followed by the drive, directory, and name of the command processor and is terminated by a byte of zeros. It is thus an ASCIIZ string—exactly what we need to execute the COMMAND.COM file.

We must then prepare a parameter block for the EXEC call. This block requires a segment address of the environment, which we already know, plus addresses of the file control blocks (FCB) and an unformatted parameter that will be placed in the Program Segment Prefix when COMMAND.COM is loaded and executed.

At this point, we save the current vector address of Interrupt 21h and reset it to a routine within the DDIR program. We are going to intercept Interrupt 21h so we can fiddle around with the DIR screen output.

The unformatted parameter area for

Loading a second version of COMMAND.COM from within an assembly program is complicated, but you may need to do it.

COMMAND.COM will be similar to the unformatted parameter in DDIR's own Program Segment Prefix, but with a few additions. To take an example, if you have issued the command

```
DDIR \MYWORK\*.TXT
```

then the "OldParameter" at offset 80h in the Program Segment Prefix will contain the number of bytes in the parameter. (In this case it is 14 or 0Eh, counting the first space.) Offset 81h will contain a space, and \MYWORK*.TXT will begin at offset 82h.

The parameter to pass to COMMAND .COM must be

```
/CDIR \MYWORK\*.TXT
```

This parameter is five characters longer than the parameter actually passed to DDIR. So it's just a matter of adjusting the number of bytes and inserting a /C and a DIR.

Now we are ready to load COMMAND.COM. The EXEC function call (4Bh) requires ES:BX to be set to the parameter block, DS:DX to be set to the ASCIIZ string of the program being loaded (which we found by searching the environment), and AL to be set to zero for a program load.

The EXEC call loads COMMAND .COM and passes the parameter to it for the DIR command. COMMAND.COM

SEGMENTED MEMORY Since you can now pop into memory at the touch of a key, you may want to have at least a general idea of where you are and what you're looking at. So, first, let us briefly discuss the memory addressing scheme used by the Intel processors in your machine.

The 8086/8088 microprocessor in an IBM PC can directly address 1MB (1024K) of memory. In hex digits, the addressing range of the processor is 0 to FFFFFh. The value FFFFFh, however, requires 20 bits in binary. That's 4 bits more than will fit in the 16-bit registers of the 8088. (The 80286 and 80386 chips default to 8088 emulation mode and so have the same initial addressing configuration.)

To provide access to the full 20-bit range of a 1MB address space, therefore, Intel uses a two-register scheme to identify each memory location. When used as a pair, these registers are said to contain the segment and offset parts of the address. By adding these registers in an adjusted manner, the actual (or absolute) address of a memory location is obtained.

The 8086/8088 addressing scheme adjusts the registers before adding them by assuming that the segment register always has a zero following its value. Thus, if the segment register contains 1000h and the offset register contains 0001h, the absolute address is 10000h + 0001h = 10001h. If the segment register contains 2345h and the offset register contains 5432h, the absolute address is 23450h + 5432h = 28882h.

The conventional way of expressing memory addresses in terms of segment and offset values is to separate the two values with a colon. For example, 1000:0100h indicates the memory location addressed with a segment of 1000h and an offset of 0100h. This is the absolute memory location 10100h.

One of the more interesting side effects of this principle is that any one absolute address can be described by different segment:offset combinations. For example, the location 10100h can be addressed as 1005:0050h, 1001:0090h, 0FFF:0110h, and 4,093 other possible combinations.

A RAM ROAD MAP The PC's memory organization is shown in the table in Figure 2. While not as detailed a map as you could find elsewhere, it does provide enough information to begin your snooping. When your curiosity gets the better of you and you want a more comprehensive memory map, try one of the many popular PC books, such as Peter Norton's *Inside the IBM PC*.

Looking at the map, you'll immediately see that certain areas, such as the interrupt vectors table, are reserved for system use. Other areas are reserved for screen memory and system ROM. The user memory area is located between the DOS data and screen memory areas. The BIOS data areas, DOS system files, and file buffers are all located in this area, as are any programs loaded by the user.

THE RAMVIEW PROGRAM The assembly listing of RAMVIEW is well commented, which should make it fairly self-explanatory for programmers. Before elaborating on several routines, we'd like to make a couple of comments regarding the way in which the program is written.

Writing a useful program that is small enough to print in *PC Magazine* is quite a challenge. A lot of effort is made to keep the code size down. In writing commercial code, program size isn't quite as important. For this reason, we used a programming technique writing RAMVIEW that we generally avoid: passing values to a procedure in registers and not on the stack. This involves more overhead for the programmer, because he has to keep track of which registers must be preserved across calls. While faster and more compact, using registers can easily produce procedures that conflict with other procedures. While both the ROM BIOS and DOS use registers to pass their arguments, OS/2 passes all parameters on the stack.

General Memory Map of PC/XT/AT

Segment:Offset	Usage
FFFF:FFFF	
	System ROM area
F000:0000	
	ROM expansion PS/2 BIOS ROM
E000:0000	
	ROM expansion EMS window
D000:0000	
	ROM expansion area Hard disk adapters EGA ROM
C000:0000	
	Reserved
B000:C000	
	CGA screen area
B000:8000	
	Reserved (used by HGC)
B000:1000	
	MDA screen area
B000:0000	
	EGA scratch area
A000:0000	
	User memory area COMMAND.COM (transient) User programs TSRs COMMAND.COM (resident) Installable device drivers DOS buffers, control areas IBMDOS.COM IBMBIO.COM
0000:0700	
	DOS data area
0000:0500	
	BIOS data area
0000:0400	
	Interrupt vector area
0000:0000	

Figure 2: *Memory on a PC is allocated in sections according to purpose.*

MEMORY RESIDENCY RAMVIEW was a relatively easy TSR to program. Because it uses very few system resources and performs no disk access, it doesn't need much code to avoid interfering with most other programs. The keyboard interrupt 9 is used to check for the hotkey and implement the pop-up logic. The BIOS video interrupt 10h is checked to avoid popping up when it is unsafe to do so.

An interrupt 9 is generated by a dedicated microprocessor in the keyboard each time a key is pressed or released. Because RAMVIEW is invoked by a keystroke, this is the logical interrupt to use to pop up. Whenever interrupt 9 occurs, the code hooked in by RAMVIEW looks for the selected hotkey. If the hotkey and the proper shift keys are pressed, RAMVIEW performs the necessary housecleaning tasks of clearing the hotkey from the keyboard microprocessor and disabling interrupts to ensure that it can't be disturbed unexpectedly.

When RAMVIEW gets control, it checks a flag byte (labeled BUSY) that it uses to signal either that it is already invoked or that it is unsafe to pop up. If the BUSY byte has a nonzero value, RAMVIEW exits with an IRET. This has the same effect as if the hotkey was never pressed.

If the BUSY byte is zero, RAMVIEW sets that byte to nonzero to indicate that it's