ALLAW SALES

Sponsored by Allaw Sales and Fujitsu

FUJITSU

tories. But even this use of TREE is overshadowed by the far better CHKDSK/V, which also lists all the files on your disk. CHKDSK/V displays full path names; TREE/F doesn't. And TREE pads all its listings with unnecessary spaces, which makes it scroll rapidly off your screen. As a bonus, CHKDSK/V adds the standard CHKDSK report detailing the number of files, bytes free, etc. And it displays the hidden files; TREE/F doesn't. Finally, CHKDSK /V is far faster. Chugging through slightly more than 2,000 files on an AT took CHKDSK/V 98 seconds. TREE /F produced an inferior report and took 123 seconds, or 25 per cent longer.

When you copy VTREE.COM into you \BIN directory, the very next thing you should do is type

ERASE \DOS\TREE.COM

Duplicate names

Note that in the above example, the full name of the primitive DOS utility that you just expunged was \DOS\TREE.COM rather than just TREE.COM. That's because you can have different versions of similarly named files in different subdirectories. You can even have similarly named subdirectories; if you wanted to (and you don't) you could have a subdirectory called \DOS and one called \BIN\DOS on the same disk.

For instance, you could rename VTREE.COM to TREE.COM and put it in \BIN. So if you kept the original DOS version in the \DOS subdirectory, your hard disk would then contain files called

\DOS\TREE.COM

(which is the original DOS version) and

\BIN\TREE.COM

(which is the renamed version of the VTREE.COM utility). To run the original DOS tree version, you'd need to type

\DOS\TREE

To run VTREE.COM, which for this example you renamed to TREE.COM, you'd type

```
100 ' Program for creating VTREE.COM -- by Charlie Petzold
110 CLS:PRINT "Checking DATA; please wait..."
120 FOR B=1 TO 32:FOR C=1 TO 16:READ A$:TTL=TTL+VAL("&H"+A$):NEXT
130 READ S:IF S=TTL THEN 150
140 PRINT "DATA ERROR IN LINE";B*10+190;" -- REDO":END
150 TTL=0:NEXT:RESTORE
160 OPEN "VTREE.COM" AS #1 LEN=1:FIELD #1,1 AS D$
170 FOR B=1 TO 32:FOR C=1 TO 16:READ A$
180 LSET D$=CHR$(VAL("&H"+A$)):PUT #1:NEXT:READ DUMMY$:NEXT
190 CLOSE:PRINT "VTREE.COM CREATED"
200 DATA EB,5F,90,00,3A,5C,2A,2E,2A,00,28,43,29,20,43,6F,1112
210 DATA 70,79,72,69,67,68,74,20,43,68,61,72,6C,65,73,20,1545
220 DATA 50,65,74,7A,6F,6C,64,2C,20,31,39,38,35,49,6E,76,1330
230 DATA 61,6C,69,64,20,64,69,73,6B,20,64,72,69,76,65,24,1475
240 DATA 52,65,71,75,69,72,65,73,20,44,4F,53,20,32,2E,30,1286
250 DATA 20,2B,24,00,00,00,5C,2A,2E,2A,00,06,01,3C,03,00,403
260 DATA 00,3C,FF,75,0A,8D,16,2D,01,B4,09,CD,21,CD,20,B4,1495
270 DATA 30,CD,21,3C,02,73,06,8D,16,40,01,EB,EC,A0,5C,00,1420
280 DATA 0A,C0,75,06,B4,19,CD,21,FE,C0,8A,D0,04,40,A2,03,1793
290 DATA 01,FC,8B,16,5D,01,B4,1A,CD,21,8B,1E,54,01,03,DB,1428
300 DATA 80,3E,53,01,00,75,12,C7,87,FC,02,00,00,BA,03,01,1187
310 DATA B9,10,00,B4,4E,CD,21,EB,04,B4,4F,CD,21,73,03,E9,1784
320 DATA DE,00,8B,36,5D,01,80,7C,15,10,75,ED,83,C6,1E,80,1639
330 DATA 3C,2E,74,E5,FF,87,FC,02,8B,0E,54,01,E3,3A,83,BF,1940
340 DATA FC,02,01,74,21,2B,DB,B0,B3,F7,87,FC,02,00,80,74,1901
350 DATA 02,B0,20,E8,FD,00,51,B9,10,00,B0,20,E8,F4,00,E2,1887
360 DATA F9,59,43,43,E2,E1,83,BF,FC,02,01,75,0B,8B,0E,5F,1876
370 DATA 01,B0,C4,E8,DD,00,E2,F9,56,8B,36,5D,01,BF,80,00,1993
380 DATA 8B,D7,B9,2B,00,F3,A4,5E,B4,1A,CD,21,B4,4F,CD,21,2024
390 DATA 72,14,80,3E,95,00,10,75,F3,B0,C2,83,BF,FC,02,01,1796
400 DATA 74,15,B0,C3,EB,11,B0,C4,83,BF,FC,02,01,74,08,B0,2009
410 DATA C0,81,8F,FC,02,00,80,E8,99,00,B0,C4,E8,94,00,B0,2159
420 DATA 20,E8,8F,00,B9,0D,00,8B,3E,5B,01,AC,0A,C0,74,06,1394
430 DATA AA,E8,7F,00,E2,F5,B0,20,E8,78,00,89,0E,5F,01,89,1944
440 DATA 3E,5B,01,FF,06,5B,01,BE,56,01,B9,05,00,F3,A4,FF,1636
450 DATA 06,54,01,C6,06,53,01,00,83,06,5D,01,2B,E9,F2,FE,1382
460 DATA 83,3E,54,01,00,74,4A,F7,87,FC,02,FF,7F,75,0A,B0,1789
470 DATA 0D,E8,3F,00,B0,0A,E8,3A,00,BF,03,01,B9,46,00,B0,1410
480 DATA 00,F2,AE,4F,B9,40,00,B0,5C,FD,F2,AE,F2,AE,47,89,2305
490 DATA 3E,5B,01,FF,06,5B,01,BE,56,01,B9,05,00,FC,F3,A4,1633
500 DATA FF,0E,54,01,C6,06,53,01,01,83,2E,5D,01,2B,E9,A1,1351
510 DATA FE,CD,20,52,8A,D0,B4,02,CD,21,5A,C3,00,00,00,00,1624
```

Fig 2 Charles Petzold's program to create VTREE.COM utility, which produces a graphic representation of a hard disk's hierarchical tree structure

\BIN\TREE

If you were in the root directory and hadn't yet used the PATH command to tell DOS where to look for executable files and you typed

TREE

you wouldn't run either \DOS\TREE or \BIN\TREE; all you'd get is a 'Bad command or file name' message. As discussed above, when you type in a command like TREE at the DOS prompt, COMMAND.COM first checks whether it's an internal command, and if it discovers it's not, checks a specified set of directories (called a PATH) for a file by that name with a .COM, .EXE, or .BAT extension. If \DOS and \BIN aren't yet included in the path, COMMAND.COM won't check in those subdirectories, and won't run either version of TREE.COM.

You can tell COMMAND.COM to check in both of these subdirectories with the command

PATH C:\DOS;C:\BIN

or

PATH C:\BIN;C:\DOS

The difference between these two is that if the top path is active, DOS will look in the \DOS subdirectory before it looks in \BIN. In the second example it

ALLAW SALES
Sponsored by Allaw Sales and Fujitsu
FUJITSU

buy copy-protected software — and back up often.

Subdirectory navigation

It's easy to create new subdirectories and move around inside existing ones if you have the right tools handy and follow a few simple rules.

The first rule is to remember that when you want to move up — toward the root directory — all you have to do is type the simple command

CD ..

(or CD..) to jump you to each successive parent directory. However, when you finally land in the root directory, you can't move up any other levels, so trying to do so will produce an 'Invalid directory' message.

What makes this especially easy is the F3 key. If you're in a subdirectory five levels deep called

LEV1\LEV2\LEV3\LEV4\LEV5

(you will be able to tell this by looking at the C:\LEV1\LEV2\LEV3\LEV4 \LEV5: prompt that your PROMPT $P: command displays) and you want to jump back to the root directory, you can do this the easy way, by typing

**CD **

or you can jump upward a level at a time by typing

CD..

once and then tapping the F3 key four more times. Each time you do, DOS will repeat the earlier command, and since that command is CD.. it will bounce you rapidly rootward.

(Get to know the F3 key, since it's a real labor saver. For instance, if you're creating a lower-level subdirectory with the MD command, and you make a typing mistake and end up creating one that's spelled wrongly, all you have to do is immediately type an R and then hit F3. This will send DOS an RD (Remove Directory) command to eradicate the erroneous one you just created. The syntax of making and removing directories is identical except for the first letter of the command, and once you type in the new first letter, F3 will dredge up the rest.)

To move in the other direction, down from the root directory to LEV5, you could, of course, simply type

CD \LEV1\LEV2\LEV3\LEV4\LEV5

You can't type

CD \LEV5

since that would tell DOS to jump you into a subdirectory called \LEV5 that was just one level down from the root directory. The real name of the \LEV5 subdirectory above is not \LEV5; it's \LEV1\LEV2\LEV3\LEV4\LEV5.

Another way to get there from the root directory is by using the relative version of the CD command to bounce you up one level at a time. Note that since DOS keeps track of each subdirectory by its full path name rather than just its particular branch on the tree, you could have a path like

C:\SHARE\AND\SHARE\ALIKE

since the subdirectory

C:\SHARE

is utterly different from

C:\SHARE\AND\SHARE

One is a single level down from the root directory, while the other is three levels down. However, having similar names like this is confusing and is a bad idea, for an important reason we'll see later.

```
100 'BATMAKR1.BAS
110 'This creates easy subdirectory switcher files
120 'Before you use this, get into DOS and type:
130 '
140 '    chkdsk / v | find "Dir" > tempfile
150 '
160 'For this to work properly, make sure each
170 '    subdirectory has its own unique name.
180 'To switch between subdirectories in DOS, type
190 '    name of the subdirectory WITHOUT the CD\
200 '    prefix, and WITHOUT the long PATHname
210 '    that usually precedes it.
220 'For instance, to switch to \DOS\BIN, just
230 '    type:  BIN
240 ON ERROR GOTO 380
250 ' --- read raw file, truncate left end of each line
260 OPEN "tempfile" FOR INPUT AS #1
270 IF EOF(1) THEN 370 ELSE LINE INPUT #1,A$
280 A$=RIGHT$(A$,LEN(A$)-12):IF A$="\" THEN 270
290 FOR A=LEN(A$) TO 1 STEP -1
300 IF MID$(A$,A,1)<>"\" THEN 320
310 NM$=RIGHT$(A$,LEN(A$)-A)+".BAT":GOTO 330
320 NEXT
330 PRINT "Creating ";NM$;" batch file..."
340 OPEN NM$ FOR OUTPUT AS #2
350 PRINT #2,"CD"+A$;:CLOSE #2
360 GOTO 270
370 CLOSE:KILL "tempfile.":PRINT:LIST 160-230:END
380 IF ERR=53 THEN LIST 120-140 ELSE ON ERROR GOTO 0
```

Fig 3 BATMAKR1.BAS is designed to create individual batch files that let you jump around your subdirectory tree structure by typing in just the subdirectory name without the log pathname that usually precedes it. Before running BATMAKR1, make sure CHKDSK.COM and FIND.EXE are on your disk (or are in subdirectories you've included in your PATH command) and type CHKDSK/V|FIND"Dir">TEMPFILE

ALLAW SALES

Sponsored by Allaw Sales and Fujitsu

FUJITSU

```
:BIN
CD C:\BIN
GOTO END
:ERROR1
ECHO Subdirectory %1 not found.
ECHO Try again.
GOTO END
:ERROR2
ECHO You must enter a subdirectory
ECHO name after %0
:END
```

Both versions require that you have CHKDSK.COM and FIND.EXE on your current directory, or in a subdirectory that you've included in your PATH. Once you've run the CHKDSK/V command mentioned above, run BATMAKR2.BAS to create the long S.BAT file.

If you enter just the name of the batch file you just created, S, with no subdirectory after it, the

IF %1@==@ GOTO ERROR2

line will jump to the ERROR2 error message. The %0 in this message is a special replaceable parameter that prints the name of the batch file itself in place of the %0. If you change the name of the batch file to something like SWITCH.BAT, this device will handle the new name.

BATMAKR2 automatically creates both a lowercase and an uppercase test. If you entered

S DOS

or

S dos

either would jump the program to the :DOS label. The line immediately following the label switches to the /DOS subdirectory and then jumps the program to the :END label so it exits. There are other faster ways to exit, such a having the batch file execute another short batch file, but the delay isn't all that bad on a RAMdisk, and you really shouldn't run this on anything else.

If you enter a subdirectory name that's not in the list of tests at the beginning of the program, you'll jump to the :ERROR1 label, which uses the %1 replaceable parameter to tell you it couldn't find the directory you specified.

BATMAKR1.BAS in Fig 3 is shorter and creates shorter files that work far faster than the long S. BAT. After you run it, to change to \BIN you'd just have to type BIN.

These programs don't offer any fancy way to jump back to the root directory. After all, CD\ isn't that hard to type. And if you're really rabid about, you can always create a ROOT.BAT batch file that executes this for you.

But how do you know what directories are on your disk? Simple. Just redirect the output of VTREE into a file called VTREE.PIC with the command

VTREE>VTREE.PIC

and then create a small batch file call V.BAT:

```
100 'BATMAKR2.BAS
110 'This creates easy subdirectory switcher files
120 ' (And puts them all in one very long file.)
130 'Before you use this, get into DOS and type:
140 '
150 '    chkdsk / v | find "Dir" > tempfile
160 '
170 'For this to work properly, make sure each
180 '    subdirectory has its own unique name.
190 'To switch between subdirectories in DOS, type
200 '    S and then the name of the subdirectory
210 '    WITHOUT the "CD\" prefix, and WITHOUT the
220 '    long PATHname that usually precedes it.
230 'For instance, to switch to \DOS\BIN, type:
240 '    S BIN
250 'DON'T run S.BAT on a floppy disk.  For best
260 '    results, run it on a RAMdisk you've PATHed to.
270 '
280 DIM B$(300),C$(300),F$(300)
290 ON ERROR GOTO 660
300 ' --- read raw file, truncate left end of each line ---
310 OPEN "tempfile" FOR INPUT AS #1
320 IF EOF(1) THEN 430 ELSE LINE INPUT #1,A$
330 B$(K)=RIGHT$(A$,LEN(A$)-10):IF B$(K)="\" THEN 320
340 FOR A=LEN(B$(K)) TO 1 STEP -1
350 IF MID$(B$(K),A,1)="\" THEN C$(K)=RIGHT$(B$(K),LEN(B$(K))-A):GOTO 380
360 NEXT
370 ' --- create lowercase version of each test ---
380 FOR D=1 TO LEN(C$(K))
390 F$(K)=F$(K)+CHR$(ASC(MID$(C$(K),D,1)) OR 32)
400 NEXT
410 K=K+1:GOTO 320
420 ' --- write upper- and lowercase tests to S.BAT ---
430 OPEN "S.BAT" FOR OUTPUT AS #2
440 PRINT #2,"ECHO OFF"
450 PRINT #2,"IF %1@==@ GOTO ERROR2"
460 FOR A=1 TO K-1
470 PRINT #2,"IF %1==";C$(A);" goto ";C$(A)
480 PRINT #2,"IF %1==";F$(A);" goto ";C$(A)
490 NEXT
500 PRINT #2,"GOTO ERROR1"
510 ' --- write actual CD instructions to S.BAT ---
520 FOR A=1 TO K-1
530 PRINT #2,":"+C$(A)
540 PRINT #2,"CD"+CHR$(32)+B$(A)
550 PRINT #2,"GOTO END"
560 NEXT
570 ' --- write error-handling and ending routines to S.BAT ---
580 PRINT #2,":ERROR1"
590 PRINT #2,"ECHO Subdirectory %1 not found.  Try again."
600 PRINT #2,"GOTO END"
610 PRINT #2,":ERROR2"
620 PRINT #2,"ECHO You must enter a subdirectory name after %0"
630 PRINT #2,":END"
640 ' --- cleanup and error routine ---
650 CLOSE:KILL "tempfile.":PRINT:LIST 170-260:END
660 IF ERR=53 THEN LIST 130-150 ELSE ON ERROR GOTO 0
```

Fig 4 BATMAKR2.BAS is designed to create one master S.BAT batch file to switch subdirectories by typing in the subdirectory name after S. Note the difference from BATMKR1.BAS, which creates small individual files. Run S.BAT from a RAMdisk for best performance. Before running BATMAKR2, make sure CHKDSK.COM and FIND.EXE are on your disk (or are in subdirectories you've included in your PATH command) and type CHKDSK/V|FIND"Dir">TEMPFILE

ALLAW SALES

Sponsored by Allaw Sales and Fujitsu

FUJITSU

COPY CON V.BAT
BROWSE VTREE.PIC

Hit the Enter key after each line, and when finished, hit the F6 function key and then the Enter key one more time.

Redirect the output of VTREE into VTREE.PIC every time you create a new subdirectory or remove an existing one. (If you want, you can create another batch file, called UPDATE.BAT, that does this for you and even puts the VTREE.PIC output file into the proper subdirectory.) Then, assuming BROWSE.COM and V.BAT are in a subdirectory that you've included in your PATH, each time you type

V

you'll see an instant graphic representation of your subdirectory tree structure. You can use the cursor and PgUp/PgDn keys to move around in the tree. Hitting Esc will return you to DOS, where you can switch to the target subdirectory by using one of the two BATMAKR methods described above.

If you don't have BROWSE.COM handy and your subdirectory tree is fairly short, you could substitute the command

TYPE VTREE.PIC | MORE

for the line BROWSE VTREE.PK

An even better adaptation of this method is to use SideKick's notepad as a window that display the VTREE.PIC file as the default. Store VTREE.PIC in you \BIN subdirectory. Bring up SideKick's main menu, and type F7 or S for the Setup menu. Type in \BIN\VTREE.PIC as the new Notefile name and hit F2 to save this as the default. Then whenever you pop up SideKick and select the notepad, the graphic representation will jump onto the screen. For best results, hit QG, which turns on the graphics line characters that connect the subdirectories.

Charles Petzold has written three very short utilities, called UP.COM, DOWN.COM and NEXT.COM, that can move you effortlessly around your subdirectory tree. To create these, run the CD.BAS program in Fig 5.

UP.COM is a lot like the command CD.. except that if you keep tapping CD.. you'll eventually get to the root directory and receive the 'Invalid directory' message mentioned earlier. When UP.COM reaches the root directory it just sits there silently.

```
100 ' CD.BAS -- makes C. Petzold's NEXT.COM, DOWN.COM and UP.COM
110 CLS:PRINT "Checking DATA; please wait..."
120 DIM S(12):FOR A=1 TO 12:READ S(A):R=R+S(A):NEXT
130 IF R<>17789 THEN PRINT "ERROR IN LINE 260 -- REDO ":END
140 FOR B=1 TO 12:FOR C=1 TO 16:READ A$:T=T+VAL("&H"+A$):NEXT
150 IF S(B)<>T THEN PRINT "ERROR LINE";B*10+260;" -- REDO":END
160 T=0:NEXT:RESTORE 270
170 OPEN "NEXT.COM" AS #1 LEN=1:FIELD #1,1 AS D$
180 FOR B=1 TO 129:READ A$:LSET D$=CHR$(VAL("&H"+A$)):PUT #1
190 NEXT:CLOSE:PRINT "NEXT.COM CREATED"
200 OPEN "DOWN.COM" AS #1 LEN=1:FIELD #1,1 AS D$
210 FOR B=1 TO 44:READ A$:LSET D$=CHR$(VAL("&H"+A$)):PUT #1
220 NEXT:CLOSE:PRINT "DOWN.COM CREATED"
230 OPEN "UP.COM" AS #1 LEN=1:FIELD #1,1 AS D$
240 FOR B=1 TO 15:READ A$:LSET D$=CHR$(VAL("&H"+A$)):PUT #1
250 NEXT:CLOSE:PRINT "UP.COM CREATED"
260 DATA 934,1655,1762,1501,1530,1326,1391,1902,1195,1758,1839,996
270 DATA EB,0D,90,2E,2E,00,2A,2E,2A,00,81,01,00,00,00,BE
280 DATA 81,01,2A,D2,B4,47,CD,21,80,3E,81,01,00,74,60,FC
290 DATA 2B,C9,AC,0A,C0,74,0D,41,3C,5C,75,F6,2B,C9,89,36
300 DATA 0A,01,EB,EE,89,0E,0C,01,BA,03,01,B4,3B,CD,21,BA
310 DATA 06,01,B9,10,00,B4,4E,CD,21,72,34,B4,4F,F6,06,95
320 DATA 00,10,74,F3,80,3E,9E,00,2E,74,EC,80,3E,0E,01,00
330 DATA 75,16,BE,9E,00,8B,3E,0A,01,8B,0E,0C,01,F3,A6,75
340 DATA D6,C6,06,0E,01,01,EB,CF,BA,9E,00,B4,3B,CD,21,CD
350 DATA 20,EB,05,90,2A,2E,2A,00,BA,03,01,B9,10,00,B4,4E
360 DATA CD,21,72,17,B4,4F,F6,06,95,00,10,74,F3,80,3E,9E
370 DATA 00,2E,74,EC,BA,9E,00,B4,3B,CD,21,CD,20,EB,04,90
380 DATA 2E,2E,00,BA,03,01,B4,3B,CD,21,CD,20,00,00,00,00
```

Fig 5 Charles Petzold's program to create NEXT.COM, DOWN.COM, and UP.COM utilities, which let you navigate easily through your subdirectories

DOWN.COM takes you in the other direction, away from the root. NEXT.COM moves you sideways. Try them. You'll like them. NEXT is especially useful when you type it in the first time and then just lean on the F3 and Enter keys to meander up and down the branches of your subdirectory tree.

Finding your way

While these utilities will make it a breeze to find any subdirectory and jump into it, they don't help you find files in your subdirectories.

You can, of course, create a small batch file call FFIND.BAT:

ECHO OFF
IF %1@==@ GOTO ERROR
CHKDSK / V | FIND '%1'
GOTO END
:ERROR
ECHO You didn't specify a filespec
:END

This short file will launch CHKDSK/V into uncovering every file on your hard disk and filter out every filename that doesn't contain the string of characters that you specified. If you enter

FFIND BAS

FFIND.BAT will print a list of every file that ends in a .BAS extension, as well as any file with the letters 'BAS' anywhere else in the filename, such as BASCOM.LIB or BASEBALL.BAT

But FFIND.BAT is slow, especially on a nearly full hard disk, since it has to pipe hundreds or thousands of filenames through a filter and create temporary files while it does so.

A better choice is to type in the WHERE.BAS program in Fig 6, which will create a file called WHERE.COM. To use WHERE.COM you must follow it with a legal DOS filespec. While FFIND.BAT lets you get away with entering fragments of filenames, WHERE.COM insists on using full and legal filenames

WHERE COMMAND.COM

or wildcards, as in

ALLAW SALES

Sponsored by Allaw Sales and Fujitsu

FUJITSU

```
100 ' Program for creating WHERE.COM
110 CLS:PRINT "Checking DATA; please wait..."
120 FOR B=1 TO 27:FOR C=1 TO 16:READ A$:TTL=TTL+VAL("&H"+A$):NEXT
130 READ S:IF S=TTL THEN 150
140 PRINT "DATA ERROR IN LINE";B*10+190;" -- REDO":END
150 TTL=0:NEXT:RESTORE
160 OPEN "WHERE.COM" AS #1 LEN=1:FIELD #1,1 AS D$
170 FOR B=1 TO 27:FOR C=1 TO 16:READ A$
180 LSET D$=CHR$(VAL("&H"+A$)):PUT #1:NEXT:READ DUMMY$:NEXT
190 CLOSE:PRINT "WHERE.COM CREATED"
200 DATA FC,BF,79,02,BE,81,00,AC,3C,0D,74,1E,3C,20,76,F7,1733
210 DATA 80,3E,5C,00,00,74,06,AC,AC,3C,20,76,06,AA,AC,3C,1366
220 DATA 20,77,FA,A0,5C,00,0A,C0,75,06,B4,19,CD,21,FE,C0,1867
230 DATA 00,06,27,02,BA,76,02,BB,2A,02,E8,16,00,80,3E,86,1162
240 DATA 02,FF,75,0D,BB,02,00,B9,1A,00,B4,40,BA,87,02,CD,1559
250 DATA 21,CD,20,52,BE,79,02,E8,86,00,33,C9,E8,60,00,72,1725
260 DATA 0D,E8,85,00,E8,6D,00,72,05,E8,7D,00,EB,F6,5A,52,1848
270 DATA BE,23,02,E8,6A,00,B9,10,00,E8,43,00,72,3F,8B,F2,1623
280 DATA F6,44,15,10,75,0D,E8,4B,00,72,32,8B,F2,F6,44,15,1668
290 DATA 10,74,F3,80,7C,1E,2E,74,ED,57,53,8B,F2,83,C6,1E,1966
300 DATA 8B,FB,AC,AA,0A,C0,75,FA,8B,DF,AA,C6,47,FF,5C,E8,2681
310 DATA A1,FF,5B,5F,C6,07,00,B4,1A,CD,21,EB,C9,5A,C3,51,2053
320 DATA 83,C2,2C,B4,1A,CD,21,8B,EA,B4,4E,BA,27,02,CD,21,1909
330 DATA 8B,D5,59,C3,8B,EA,B4,4F,BA,27,02,CD,21,8B,D5,C3,2280
340 DATA 8B,FB,AC,AA,0A,C0,75,FA,C3,8B,EA,80,7E,1E,2E,74,2315
350 DATA 22,BA,27,02,32,C0,A2,86,02,86,07,97,E8,15,00,97,1497
360 DATA 88,07,8B,D5,83,C2,1E,E8,0A,00,B4,09,BA,9F,02,CD,1833
370 DATA 21,8B,D5,C3,8B,F2,B4,02,AC,8A,D0,CD,21,AC,0A,C0,2273
380 DATA 75,F7,C3,2A,2E,2A,00,40,3A,5C,00,00,00,00,00,00,903
390 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,0
400 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,0
410 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,0
420 DATA 00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,0
430 DATA 00,00,00,00,00,00,00,00,00,2A,2E,2A,00,00,00,00,130
440 DATA 00,00,00,00,00,00,FF,4E,6F,20,6D,61,74,63,68,69,1106
450 DATA 6E,67,20,66,69,6C,65,73,20,66,6F,75,6E,64,2E,0D,1407
460 DATA 0A,24,00,00,00,00,00,00,00,00,00,00,00,00,00,00,46
```

Fig 6 Program to create WHERE.COM file finder

process of adding to and editing down your files each day ends up sowing little file fragments more or less at random over the surface of your disk.

You should periodically copy all your files to a backup medium (and get rid of the duplicates, .BAK versions, and dead data in the process), reformat your hard disk, and then copy everything back. You'll notice an immediate improvement in speed. When you do this, put the subdirectories that you PATH to at the very beginning of your directory by making sure they're the first ones you copy to the newly formatted disk.

One final pearl of wisdom is obvious but bears repeating. Think before you FORMAT. Even though the latest versions of DOS make you type in a Y and then hit the Enter key before letting it go ahead and wipe everything out, late at night you may misinterpret the question or hit a Y when you mean N, or have some aberrant and lethal combination of JOIN, APPEND, and SUBST bubbling away under the surface that steers an innocent floppy request into a jolt of panic. A few seconds into the formatting process the hard disk FATs and directories get zeroed out, any attempt at resurrection is only a best guess. It is possible to bring much of your data back to life with a utility like Mace's or Norton's, especially if you let Mace park a copy of your FAT ahead of time. But don't tempt fate.

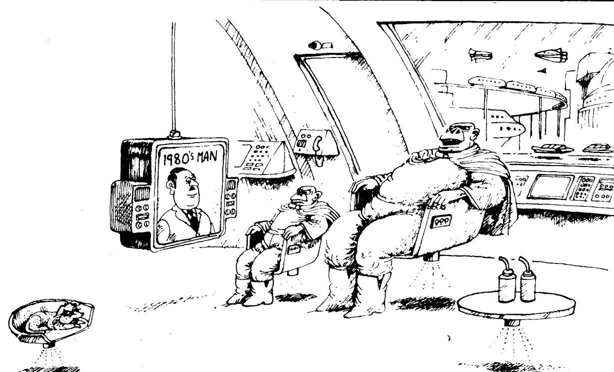
If you're working on something time sensitive and critically important, stop

'One final pearl of wisdom is obvious but bears repeating. Think before you FORMAT.'

frequently while you're working and make a working copy on a floppy. It is possible to corrupt a hard disk if your writing to it and the electricity commission decides that moment would be a good one to switch generators. You can set up a batch file to automate the process. Otherwise you might end up spending the rest of the evening patching together little shards of your work that you've fished out of the magnetic murk.

If you notice that performance is degrading, or hear the percussive rhythm of repeated read retries, run Norton's DISKTEST program. This takes a few minutes, but can ferret out developing programs and zap out bad sectors better than DOS can. And if the Norton program reports grief, back up everything pronto and dive down to your dealer. When hard disks start whimpering, they go downhill very fast. Hard disk problems never just go away.

END



'It's nature's way of adapting, son.
Now that computers do all our brainwork we don't need such big heads.'

```
(* (C) Copyright 1986, D. M. Armstrong-Allen *)
{$R+}
PROGRAM self;
TYPE
  s64 = STRING[64];            (* length of a filename w/path *)

  FUNCTION Self : s64;         (* Requires MS/PC-DOS v3.00 or later *)
  VAR
    envseg : Integer ABSOLUTE CSeg : $002C;
    i      : Integer;
    temp   : s64;
  BEGIN
    i := 0;
    temp := '';
    (* Read thru the environment until we get to the end, *)
    (* i.e. two null bytes.                                *)
    WHILE MemW[envseg:i] <> 0 DO
      i := i+1;
    i := i+4;                   (* Skip the two null bytes and word count *)
    (* Get the d:\path\filename.ext as passed to the EXEC function. *)
    WHILE Mem[envseg:i] <> 0 DO
      BEGIN
        temp := temp+Chr(Mem[envseg:i]);
        i := i+1;
      END;
    Self := temp;
  END;

BEGIN
  WriteLn('This program is named "', Self, '".');
END.
```

Fig 9 A program that can locate itself on disk

```
{$R+}
PROGRAM DOSVersion;
VAR
  X,Y : integer;
PROCEDURE Dos_Version(VAR Maj, Min : integer);
TYPE
  Registers  = Record
                 CASE Integer Of
                   1 : ( AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : Integer );
                   2 : ( AL,AH,BL,BH,CL,CH,DL,DH : Byte );
                 END;
VAR
  R : Registers;
BEGIN
  WITH R DO
    BEGIN
      AH := $30;
      MSDOS(R);
      IF AL = 0 THEN BEGIN Maj := 1; Min := 0; END
      ELSE BEGIN Maj := AL; Min := AH; END;
    END;
END;

BEGIN
  Dos_Version(X,Y);
  WriteLn('DOS version ',X,'.',Y);
END.
```

Fig 10 A simple routine to check the current DOS version

DOS environment and search it yourself. Under DOS 3.x, you can run any program simply by spelling out its full pathname. By combining these two methods with the simple test for the current DOS version used in the DOS_Version procedure that I've included in Fig 10, you could construct a nearly foolproof procedure for locating your program's data files or overlay files, no matter where the program was called from — PS

QuickBASIC 2.0 and the EGA

I've recently bought Microsoft's QuickBASIC 2.0 for the IBM PC and have been converting some of my older Basic programs using graphics to work with the Enhanced Graphics Adaptor (EGA) high-resolution modes. These programs now use SCREEN 9, which offers 640- by 350-pixel resolution with 16 colours.

However, the way that tiling works with the EGA really has me confused. It's not explained at all in manual, and I've had to do it by trial and error. Can you shed some light on this subject?

Further, the BSAVE and BLOAD commands don't seem to work correctly with the EGA. How can I get them to work?

Greg Griffith

The 600-page QuickBASIC 2.0 manual looks complete, except when you need to find something in it. It's amazing how much is not in there at all. EGA graphics certainly deserves a more ex-

```
DefInt A-Z
Def FnRand(X) = Int (X * Rnd)
Screen 9
For I = 1 to 25
        Line (FnRand(640),FnRand(350))-(FnRand(640),FnRand(350)),FnRand(16),BF
Next I
Call EgaBsave ("SAVESCRN", 0, 80 * 350)
Cls
Call EgaBload ("SAVESCRN")
End

Sub EgaBsave (FileName$, Offset, Length) Static
        Def Seg = &HA000
        Out &H3CE,4 : Out &H3CF,0
        Bsave FileName$ + ".BLU", Offset, Length
        Out &H3CE,4 : Out &H3CF,1
        Bsave FileName$ + ".GRN", Offset, Length
        Out &H3CE,4 : Out &H3CF,2
        Bsave FileName$ + ".RED", Offset, Length
        Out &H3CE,4 : Out &H3CF,3
        Bsave FileName$ + ".INT", Offset, Length
        Out &H3CE,4 : Out &H3CF,0
End Sub

Sub EgaBload (FileName$) Static
        Out &H3C4,2 : Out &H3C5,1 : Bload FileName$ + ".BLU"
        Out &H3C4,2 : Out &H3C5,2 : Bload FileName$ + ".GRN"
        Out &H3C4,2 : Out &H3C5,4 : Bload FileName$ + ".RED"
        Out &H3C4,2 : Out &H3C5,8 : Bload FileName$ + ".INT"
        Out &h3C4,2 : Out &H3C5,&H0F
End Sub
```

Fig 11 This QuickBASIC 2.0 program demonstrates the use of two subroutines to save and load high-resolution EGA graphics displays to disk

tensive discussion. The organisation of video memory in the EGA high-resolution modes is fundamentally different from the Colour Graphics Adaptor (CGA). In the CGA medium-resolution mode (320 by 200 pixels with four colours), each pair of 2 bits represented 1 pixel. That pixel could be one of four different colours.

The EGA high-resolution video memory is organised as four colour planes. There are separate planes for blue, green, red, and intensity. Each bit in each plan corresponds to 1 pixel. In the PAINT statement, the first 4 bytes in the tiling string are for these four colour planes. The first byte represents the bit patterns for the blue plane, the second byte for the green plane, the third byte for red, and the fourth byte for the intensity plane.

For instance, the statement

PAINT (X,Y), CHR$(&HC8)+ CHR$(&HC1)+CHR$(&H8f)+ CHR$(&H81)

(Although shown on three lines so as to fit within the column, the statement above must be entered as a single line — Ed) means to use pixel patterns of 11001000 (or &HC8) for blue, 11000001 (or &HC1) for green, 10001111 (or &H8F) for red, and 10000001 (or &H81) for intensity. These 4 bytes define the colours of a set of 8 pixels. The colours will be combined with one another so that the pixels from left to right are:

Pixel	*Colour*
1	*High-intensity white (all on)*
2	*Cyan (blue and green)*
3	*Background (all off)*
4	*Background (all off)*
5	*Magenta (blue and red)*
6	*Red*
7	*Red*
8	*Yellow (green, red, and intensity)*

If you use the PALETTE statement, these colours will be different. If you use more than 4 bytes, the second set of 4 bytes will apply to the next scan line.

The BSAVE and BLOAD statements are more of a problem, but this again results from the organisation of EGA high-resolution video memory into colour planes.

The four colour planes of the EGA all occupy the same address, starting at A000:0000. When reading from the memory space, only one of the colour planes may be active. It is necessary to switch between the applicable colour planes when doing a BSAVE and BLOAD and save each of the four colour planes separately in different files. This is done by manipulating output ports on the EGA.

Fig 11 shows a QuickBASIC 2.0 program with two subroutines called EgaBsave and EgaBload that will do this for you. The program draws some random rectangles on the screen and calls EgaBsave. This has three parameters:

EgaBsave(filespec,offset,length)

Filespec *is a filename without an extension. It can have a path.* Offset *will be 0 when you're working with the first video page.* Length *is the number of pixels on the display divided by 8. EgaBsave switches through the four colour planes for reading the display and does a BSAVE on each one while appending the extension BLU, GRN, RED, and INT to the filename.*

After that, this demonstration program clears the screen and loads the four colour planes back in using EgaBload. EgaBload requires only a filespec. It switches through the four colour planes for writing to the display and BLOADs each of the four files. Particularly if you're loading these files from a floppy diskette, you'll clearly see that each of the four colour planes is loaded separately.

Note that manipulating the EGA registers in a program that uses the EGA may have some unforeseen consequences, but it appears that the Basic routines leave the EGA in a normal state and don't seem to be adversely affected when you mess with the registers — CP

Screen slider

The short Apple II machine-language program in Fig 12 lets you slide a line of 40-column text off the screen to the right, accompanied by sound. To use the routine in your own Applesoft Basic programs, simply include lines 20-50 near the beginning of your program.

Then do a VTAB n (where *n* is in the range from 1 to 24) to select the line you want to slide and CALL A. The address A can be any convenient location with 29 unused contiguous bytes.

The 88 in line 50 controls the speed of the slide. Larger values produce a slower slide, and smaller values

```
10 PRINT  CHR$ (21): REM   GO TO 40 COLUMN MODE
20 A = 768: REM   MACHINE LANGUAGE ADDRESS = $300
30 FOR I = A TO A + 28: READ D: POKE I,D: NEXT : REM  INSTALL ML
40 DATA 162,40,160,39,136,177,40,200,145,40,136,208,247,169
50 DATA 160,145,40,44,48,192,169,88,32,168,252,202,208,230,96
60 A$ = "ABCDEFGHIJKLMNOPQRSTUVW": REM  FILL SCREEN
70 FOR V = 0 TO 39: PRINT A$;: NEXT : REM    FILL SCREEN
80 FOR V = 1 TO 23: VTAB V: CALL A: NEXT : REM SLIDE LINES
90 GOTO 70: REM  KEEP IT UP!
```

Fig 12 Screen slider program

```
N NEWKEYS.COM
A
JMP     013A            ; Jmp Initialize
DW      0,0
CMP     AH,00           ; NewInt16:
JZ      0115            ; Jmp GetKey
CMP     AH,01
JZ      0121            ; Jmp GetStatus
CS:
JMP     FAR [0102]      ; Jmp OldInt16
MOV     AH,10           ; GetKey:
PUSHF
CS:
CALL    FAR [0102]      ; Call OldInt16
CALL    0131            ; Call FixUp
IRET
MOV     AH,11           ; GetStatus:
PUSHF
CS:
CALL    FAR [0102]      ; Call OldInt16
JZ      012E

CALL    0131            ; Call FixUp
RETF    0002
CMP     AL,E0           ; FixUp:
JNZ     0139
SUB     AL,AL
CMP     AL,01
RET
MOV     AX,3516         ; Initialize:
INT     21              ; Get OldInt16
MOV     [0102],BX       ; Save it
MOV     [0104],ES
MOV     DX,0106
MOV     AX,2516
INT     21              ; Set NewInt16
MOV     DX,013A
INT     27              ; Stay Resident

R CX
54
W
Q
```

Fig 6 This DEBUG script file creates a remain-resident program called NEWKEYS.COM that lets DOS use the new keyboard codes defined for the IBM enhanced keyboard

You were misinformed — the codes for the F11 and F12 keys are actually 133 and 134. However, don't rush off and try them just yet. There's a catch.

When IBM designed the BIOS support for the enhanced keyboard, they added over 30 new extended keyboard codes, starting at 133. However, they did not make these keyboard codes available to programs through the normal BIOS keyboard interface. To do so would have created incompatibilities with some existing programs. For instance, some keyboard macro programs define their own extended keys and these may conflict with the new IBM codes.

DOS (and most programs) get keyboard information from the BIOS through interrupt 16h, function calls 0, 1 and 2. For the enhanced keyboard, IBM defined new function calls numbered 10h, 11h and 12h that duplicated 0, 1 and 2, except that the new calls also return the new extended keyboard codes in addition to the old ones.

Fig 6 shows a DEBUG script for a NEWKEYS.COM program you can create that allows DOS access to the new codes and thus allows you to use these new keys with ANSI.SYS. You can create NEWKEYS.COM by typing the lines shown into a file called NEWKEYS.SCR. (You don't need to type the semicolons or the comments that follow them.) Then execute

```
DEBUG<NEWKEYS.SCR
```

This creates NEWKEYS.COM. NEWKEYS.COM is a terminate-and-stay-resident program, so it need be loaded only once during your PC session. Like most TSRs, it may have some compatibility problems with other programs. If everything seems to work okay once you load it, then you're probably in good shape.

When NEWKEYS is loaded, you can use the extra keyboard codes for ANSI.SYS redefinitions. The new codes are shown in Fig 7. (The old codes can be obtained from the IBM Basic manual.) For instance, the ANSI sequence for redefining the F11 key to do a DIR command is

```
<Esc>[0;133;"DIR";13p
```

— CP.

Password

The program below lets you keep others out of the programs and files on your Apple DOS 3.3 disk. Use POKE 21503,0 to disable the catalog on a DOS 3.3 disk if the password is not correct. To change the password, change lines 30, 40 and 55 from TRIPLE BOGIE to the password desired. Just save it under the boot program's name.

Mike Horak

```
10  HOME : PRINT "WHAT IS
    THE CODE WORD?"
20  INPUT A$
30  HOME : IF A$ < > "TRIPLE
    BOGIE" THEN POKE - 21503,0
40  IF A$ = "TRIPLE BOGIE"
    THEN PRINT "HELLO MIKE"
50  PRINT : PRINT : PRINT :
    PRINT : PRINT : PRINT :
    PRINT
55  IF A$ < > "TRIPLE BOGIE"
    THEN ONERR GOTO 70
60  PRINT CHR$ (4);"CATALOG"
70  NEW
```

Extended Code	Key	Extended Code	Key
133	F11	149	Ctrl /
134	F12	150	Ctrl *
135	Shift F11	151	Alt Home
136	Shift F12	152	Alt Up-Arrow
137	Ctrl F11	153	Alt Page-Up
138	Ctrl F12	155	Alt Left-Arrow
139	Alt F11	157	Alt Right-Arrow
140	Alt F12	159	Alt End
141	Ctrl Up-Arrow	160	Alt Down-Arrow
142	Ctrl -	161	Alt Page-Down
143	Ctrl 5	162	Alt Insert
144	Ctrl +	163	Alt Delete
145	Ctrl Down-Arrow	164	Alt /
146	Ctrl Insert	165	Alt Tab
147	Ctrl Delete	166	Alt Enter
148	Ctrl Tab		

Fig 7 IBM's new extended keyboard codes for the enhanced keyboard

IIGS on-screen clock

The following utility program reads the IIGS clock and puts the time and date in variable T$, which you can display or manipulate by using MID$, LEFT$, etc. Other strings can be concatenated with T$. Just don't try to redefine T$, and be sure that it's the first statement in your program. If you run the program, this statement won't look right because it now holds the time/date last displayed, which tokenises into Applesoft commands such as COLOR=. You can also poke the time and date directly to the screen. Try changing 9,8 to 50,6. Doing so sends the output to $0632 on the screen page instead of $0809 in the Applesoft program. You can also change the format with open-apple/Control/Esc as you're running the program.
David Hill

```
10  T$ = "MO/DY/YR HR/MI/SE
    SM": HOME : FOR I=0 TO 19:
    READ J: POKE 768 + I,J:
    NEXT : DATA
    24,251,194,48,244,0,0,244,9,
    8,162,3,15,34,0,0,225,56,251,96
20  CALL 768: VTAB 5: HTAB 5:
    PRINT T$:KB = PEEK
    (49152): IF KB < 128 THEN 20
```

Getting to point mode

When you edit a formula in 1-2-3, it's often easier to re-enter cell references by pointing to them rather than by typing cell addresses. This is especially true if you can't tell what the addresses are because the cells are off the screen. But once you have hit F2 and are in Edit mode, how do you then get into Point mode?

If you want to change the last cell reference in the formula, backspace over it so that the formula ends with an arithmetic operator (+,*,etc.), a comma, or an open parenthesis. Now, when you hit the Up or Down Arrow key, the cursor will move to the next cell and you'll be in Point mode. To move the pointer right or left, you still have to use the Up or Down Arrow key to get into Point mode first.
John Predmore

The programmer who included this escape from Edit mode must have forgotten to explain it to whoever wrote the manual. In fact, the Release 2 manual states clearly that when you are in Edit mode, the Up or Down Arrow keys enter the formula you were editing and move the cursor to the next cell, just as they do when you enter a brand new formula. Most of the time, that's true. Up or Down Arrow keys switch you from Edit to Point mode only when you edit a formula so that it ends with an arithmetic character, as Mr Predmore explains. And you don't need to delete anything. If you hit F2 and just add a + sign to the end of a formula, the Up and Down Arrow keys do their magic. Add a + sign to the middle of the formula, and the Arrow keys won't *put you in Point; they'll behave just as the manual says they will — JT.*

into two equal-sized groups of N/2 items and performing the task separately on each half. That is, given N items:

Total Task
N items

try to decompose the total task into two separate parts:

Task A	Task B
N/2 items	N/2 items

and derive the final result by combining the result from tasks A and B.

At this point it is difficult to see how it is that the amount of work involved can possibly be reduced by this division. Surely task A and task B together must take at least as long as the total task, and what about the extra time required to perform the division and recombination? Surprising though it seems, such a division does, in practice, usually provide an increase in speed because the dominating factor is how many items the task is working on. If you assume that a speed gain is indeed produced by such a division — what would you do to increase it even further? The answer is that you would repeat the process by dividing each of the two sub-tasks into two sub-sub-tasks, and so on, until each section of the problem consisted of a single element and the task performed on it would usually turn out to be trivial.

If there are N items, how many times can this division process be repeated? The answer to this is (perhaps not surprisingly for those mathematically minded) $\log_2 N$. Of course, it is only possible to carry out this division exactly if N is a power of two, and this accounts for the restriction of many of the fast methods to values of N that are a power of two. For example, if N=8 then the division process can be applied exactly three times. It is sometimes useful to draw a diagram of the division process as a binary tree. In the case of N=8, see the example shown on the following page.

In general the decomposition tree for N items has $\log_2 N$ levels, and this visualisation gives us a way of exploring the way that an increase in speed might be gained. If each division process takes time T, then the total time taken is $T\log_2 N$ and this is often smaller than the time taken for the straightforward approach.

If you still think that all this is unlikely, then to a certain extent I have to agree with you. But it is surprising how often in practice a division process can be found which not only gives the same result as the original process, but is an order faster! The trouble with all this is

Binary search

Binary search is perhaps the best known of all the fast methods. Indeed, it is so well-known and loved that it is often not counted as an improved method but as *the* method. If you are searching a list of items for a particular target item, then the simplest algorithm is the linear search — that is, start at the top of the list and compare each item to the target. It is not difficult to see that linear search takes, on average, N/2 operations to find an item and N operations to discover that an item is not in the list. Thus, linear search is an algorithm that takes time proportional to O(N).

If the list of items is sorted into order, a better method of searching — binary search — can be employed. This works by repeatedly dividing in two the range that the target is thought to be in. If the items are stored in the array A with the smallest in A(1) and the largest in A(N), then at the beginning of the search we assume that the target will be in the range of L to U with L=1 and U=N — that is, the entire array.

The first state in the division process is to decide if the target, stored in T, is in the lower or upper half of the array. If M is in the middle of the range, we can decide which sub-range the target must be in by the following tests:

IF A(M) <T THEN target is in upper range — that is, M+1 to U

IF A(M) >T THEN target is in lower range — that is, M-1 to L

Of course there is also the possibility that A(M)=T, and in this case we have found the target and the process terminates. That is: IF A(M)=T THEN target found

This division process continues until either the target is found or L>U, in which case the interval has been shrunk to nothing and the target is not in the array. This sketch of the method is sufficient to write a Basic subroutine to perform a binary search:

```
1000    L=1:U=N
1010    REM DIVISION LOOP
1020    IF L>U THEN I=0:RETURN:        REM EMPTY RANGE EXIT LOOP
1030     M=CINT((L+U)/2):              REM COMPUTE MIDDLE OF RANGE
1040     IF A(M)<T THEN L=M+1:         REM TARGET IN UPPER HALF
1050     IF A(M)>T THEN U=M-1:         REM TARGET IN LOWER HALF
1060    IF A(M)=T THEN I-M:RETURN:     REM TARGET FOUND EXIT LOOP
1070    GOTC 1010
```

When this subroutine terminates, I contains either the position of the target in the array or O to indicate that it hasn't been found.

Binary search is a clear application of the repeated division principle. Each division produces a pair of sub-tasks: that is, find the target in half the number of items, but one of the sub-tasks is trivial because we can decide that the target is not in its half of the array. In this case, each division takes the same amount of time and doesn't depend on the number of items in the list. As each division takes a constant time and the number of divisions is $\log_2 N$, the entire process takes $O(\log_2 N)$. This represents a considerable saving in time for large lists.

For example, a linear search of 1000 items takes 500 comparisons to find the target and 1000 to report that it isn't present. A binary search of the same set of items takes roughly 10 divisions either to find, or not find, the target.

Of course, for a binary search the items have to be in order and the additional time it takes to sort them has been taken into account, but often the list has to be sorted for other reasons. Even if this isn't the case, it doesn't need many look-ups to make binary search plus sort more efficient than linear search.

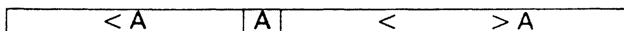
From the point of view of our general repeated division method, binary search is an example of an improvement on an O(N) method — linear search — that can be implemented as $\log_2 N$ divisions in which each takes a time that doesn't depend upon N. Binary search is a very powerful and general algorithm that can crop up in many unexpected places.

For example, if you are looking for a serious bug in a program that completely crashes the system, the quickest way to track it down with minimum effort is to place print statements that divide the program into ranges in a manner of a binary search. That is, for the first run place the print statement in the middle of the program. If you see its output, the bug must be in the second half; if not, the bug is in the first half. Simply repeat the operation until the bug is pinned down into a small enough range for you to spot it. At most this will take $\log_2 N$ runs and print statements, where N is the number of lines in the program.

Quicksort

Quicksort, invented in 1962 by CAR Hoare, is still the fastest sorting method that we know. And as it takes time proportional to $O(N\log_2 N)$, we know that it must be within a constant of being optimum. Simple sorting methods such as selection sort, insertion sort, and so on, take time proportional to $O(N^2)$ or worse, so quicksort is a great improvement when N is large. For small numbers of items, simple sorting methods may actually be faster than the more complicated quicksort, but as N increases it doesn't take long for quicksort to live up to its name.

The fundamental operation of quicksort is a division of the array into a right-hand part that contains items greater than a given value A, and a left-hand part that contains items less than this value. (The value of A is arbitrary, but for an efficient method it is desirable that it divides the array into roughly two equal-sized portions.) That is, after the first partition the array is:



This partitioning operation can be performed using two pointers — I and J, say. Firstly, a scan to the right is performed using I to find an element bigger than A, then a scan to the left is performed using J to find an element smaller than A. These two elements are clearly in the wrong portions of the array, so they have to be swapped. That is:

```
      REM SCAN RIGHT
      I=1
x     IF X(I)<A THEN I=I+1:GOTO x
      REM SCAN LEFT
      J=N
y     IF X (J)>A THEN J=J-1:GOTO y
      REM SWAP X (I) AND X(J)
```

After the first swap, the left and right scans continue from where they left off and elements are swapped until the two pointers meet somewhere in the middle of the array. At this stage the partition is complete, and all the elements to the left of the meeting place are less than A and all the elements to the right of the meeting place are greater than A.

A partitioning of this type doesn't result in a sorted array, but the array is more ordered in that during subsequent sorting, no elements will have to be moved between the two halves. This, of course, means that the two halves can be sorted independently of one another, and we have succeeded in splitting the task of sorting N items into two tasks of sorting N/2 items.

The next stage should be obvious in that further applications of the partitioning method would reduce the task even more. Repeatedly partitioning the array finally results in partitions of single elements which need no additional work to sort: that is, the array can be completely sorted by use of nothing but the partitioning method.

You should recognise in this all the features of the general partitioning method described earlier. As each partition takes roughly $O(N)$ operations and on average $\log_2 N$ partitions will be needed, the entire quicksort procedure will take $O(N\log_2 N)$.

The subroutine given below performs a quicksort on the array X. It is essentially based on the methods described above but with some practical modifications to make the process more efficient. In particular, to minimise storage overheads, the smallest of the two portions of the array produced by a partition is selected for further partitioning. If you are not convinced that such an elaborate subroutine could be faster than a simple selection or insertion sort, it is worth examining the following table:

	256 items	512 items
Insertion sort	366	1444
Selection sort	509	1956
Bubble sort	1026	4054
Quicksort	60	146

(The times are in milliseconds for Pascal versions — taken from N Wirth, *Algorithms+DataStructures=Programs.*)

```
1000 REM QUICKSORT
1010 M=12:                           REM DEPTH OF STACK
1020 S=1:                            REM STACK POINTER
1030 DIM STACK(M,2):                 REM ***MOVE TO MAIN PROGRAM***
1040 STACK(1,1)=1:STACK(1,2)=N:      REM INITIALISE STACK
1050 REM LOOP POP STACK
1060  L=STACK(S,1):R=STACK(S,2):S=S-1
1070  REM DO DIVISION OF L TO R
1080  I=L:J=R:A=X(INT((L+R)/2))
1090  REM SWAP X(I), X(J) LOOP
1100   REM SCAN RIGHT LOOP
1110    IF X(I)<A THEN I=I+1:GOTO 1110
1120   REM SCAN LEFT LOOP
1130    IF X(J)>A THEN J=J-1:GOTO 1130
1140   IF I>J THEN GOTO 1190:        REM EXIT SWAP LOOP
1150   W=X(I):X(I)=X(J):X(J)=W:      REM SWAP VALUES AT I AND J
1160   I=I+1:J=J-1:                  REM SET POINTERS FOR NEXT SCAN
1170   IF I>J THEN GOTO 1190:        REM EXIT SWAP LOOP
1180  GOTO 1090
1190  REM STACK SMALLEST PARTITION
1200  IF J-L<R-I AND I<R THEN S=S+1:STACK(S,1)=I:STACK(S,2)=R
1210  IF J-L>=R-I AND L<J THEN S=S+1:STACK(S,1)=L:STACK(S,2)=J
1220  REM SORT REMAINING PARTITION
1230  IF J-L<R-I THEN R=J ELSE L=I
1240  IF L>=R THEN GOTO 1260:        REM EXIT DIVISION LOOP
1250 GOTO 1070
1260 IF S=0 THEN GOTO 1280:          REM FXIT LOOP STACK EMPTY
1270 GOTO 1050
1280 RETURN
```

ask how much the time taken increases if N is doubled:

N doubles:	$O(N)$	$O(N^2)$	$O(N^3)$
Increases time by:	2	4	8

Using this table, it isn't difficult to see that an $O(N)$ algorithm remains practical long after you have grown old waiting for an $O(N^3)$ algorithm to finish. You might think that this is an exaggeration: an $O(N^3)$ algorithm that takes one second to process 100 items seems inefficient, but not ridiculously so when compared with an $O(N)$ algorithm that takes .001 seconds. However, for 1,000,000 items the $O(N)$ algorithm would take something like 10 seconds, but the $O(N^3)$ algorithm would take 32,000 years!

Algorithms which take times proportional to $O(N^c)$, where c is a constant, are called 'polynomial time algorithms'. But there are algorithms that perform worse than polynomial time algorithms.

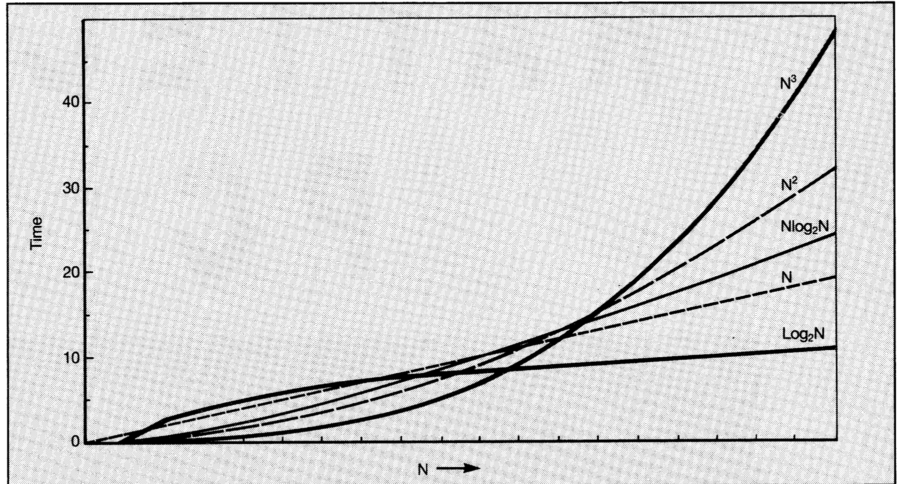
For example, an exponential time algorithm $O(e^N)$ performs worse than any polynomial time algorithm. In other words, $O(e^N)$ is another order of badness! Most of the magic algorithms described in this article take time proportional to either $O(N\log_2 N)$ or $O(\log_2 N)$. An $O(N\log_2 N)$ algorithm is worse than $O(N)$ but better than $O(N^2)$, and an $O(\log_2 N)$ is particularly prized because it is even better than $O(N)$. If N is doubled, the time taken by an $O(\log_2 N)$ algorithm only increases by one unit of time whereas an $O(N)$ algorithm doubles the time it takes.

If you are not familiar with the $\log_2$

(that is, log to the base 2) function, it is worth saying that $\log_2 N$ is simply the power of 2 that equals N. In other words, if $a=\log_2 N$ then N=2a. For example, $\log_2 8=3$ because $2^3=8$, and $\log_2 2.828=1.5$ because $2^{1.5}=2.828$ (not so easy to verify by simple arithmetic due to the fractional power).

The reason why $\log_2$ is involved in the performance of all of the fast methods is no accident and, indeed, it is a clear indication of the division principle that they all embody. This is because asking how many times N can be divided by 2 before reaching 1 is equivalent to asking how many 2s have to be multiplied together to make N — that is, what power of 2 equals N, and this is simply $\log_2 N$.

END



Graph to show time in relation to number of items

The fast median finder

The fast median finder was invented in 1970, and it's a strange blend of binary search and quicksort. The median of a set of numbers is the value that 'lies in the middle': that is, half of the values are smaller or equal to it and the other half are larger or equal to it.

Another definition of the median is that it is the middle value after sorting the set into order. For example, the median of:

4 2 10 3 7 18 60

can be found by first sorting the set into order:

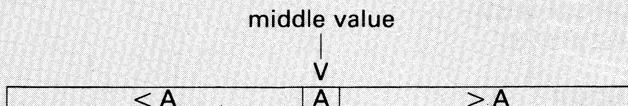
2 3 4 <u>7</u> 10 18 60

and picking the middle value — that is, the median is 7. In statistics, the median is often used in place of the mean to indicate the value that a set of numbers is centred on.

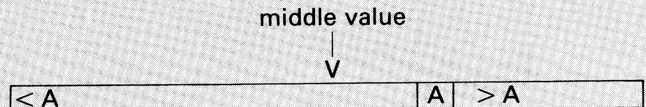
The most common way of finding the median is to proceed as above and sort the numbers into order before finding the middle value. If the best sorting method is used, this takes $O(N\log_2 N)$, but there is a much faster method that will find the median in time proportional to O(N) based on the partitioning operation introduced as part of quicksort.

If you perform the partitioning operation used in quicksort on an array using a value A, then the result splits the array into two portions: one smaller than or equal to A; and one bigger than or equal to A.

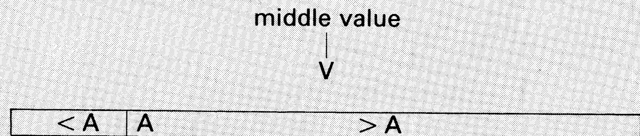
If this division is into two equal parts, then A is the median of the set of values:



However, as A was chosen at random, this equal split is unlikely to be obtained. If the left-hand portion of the split is larger, the value A is too big to be the median which must lie in the left-hand portion:



If, on the other hand, the right-hand portion is larger, the value of A is too small to be the median which must lie in the right-hand portion:



At this stage you should be able to see that the partitioning operation either finds the median, or pins it down to lying in one of the two portions of the array. This is remarkably similar to the division of the range that a target is assumed to lie in during a binary search.

The next stage is to repeatedly apply the partitioning operation to the portion of the array that the median is located in until it is found. This on average takes 2N operations, so the entire process is O(N) which is a considerable improvement over $O(N\log_2 N)$.

The following program finds the median of the values stored in the array X using the method described above. The median returned is A(K).

```
1000 L=1:R=N:REM SET INITIAL RANGE
1010 K=CINT(N/2) :REM K=POSITION OF MEDIAN I.E. MIDDLE OF ARRAY
1020 REM PARTITION UNTIL L>R
1030 IF L>R THEN GOTO 1200
1040 A=X(K):                          REM SET VALUE OF A
1050 I=L:J=R:                         REM SET SCAN POINTERS
1060  REM DO DIVISION OF L TO R
1070  REM SWAP X(I), X(J) LOOP
1080   REM SCAN RIGHT LOOP
1090    IF X(I)<A THEN I=I+1:GOTO 1090
1100   REM SCAN LEFT LOOP
1110    IF X(J)>A THEN J=J-1:GOTO 1110
1120   IF I>J THEN GOTO 1170:         REM EXIT SWAP LOOP
1130   W=X(I):X(I)=X(J):X(J)=W:       REM SWAP VALUES AT I AND J
1140   I=I+1:J=J-1:                   REM SET POINTERS FOR NEXT SCAN
1150   IF I>J THEN GOTO 1170:         REM EXIT SWAP LOOP
1160  GOTO 1070
1170  IF J<K THEN L=I:                REM SET POINTERS TO LARGEST
1180  IF K<I THEN R=J:                REM PARTITION
1190 GOTO 1020
1200 RETURN
```

security systems make very heavy use of the ports, and so a quite transportable product may lock itself tightly to IBM compatible hardware simply through the use of security disks. Each of the controllers shown in the I/O port map corresponds to an actual chip, usually one manufactured by Intel. For a machine to be IBM compatible, it must not only use the same I/O ports for the same functions, but must also utilise exactly the same controller chips. In this series, we will also be looking at the most useful ports, what they do, and how to use them.

Figure 4 shows the interrupt map of the PC. The first kilobyte of memory is dedicated to the interrupt vectors. This is not an attribute of DOS or the Bios, rather it is something hardwired into 808X processors. Interrupts are numbered from zero to 255 and, when a particular interrupt occurs, the 808X automatically looks up the corresponding four byte address stored in the interrupt table, saves a few registers, and branches to that location. There are actually three different types of interrupts. The first are those generated internally by the 808X; for example, divide by zero and instruction single step. The second are those generated by external devices; for example, when the status of the communications adaptor changes or a disk completes an I/O. Finally, interrupts can be generated by software. This provides the mechanism by which DOS and the Bios can be called from user programs, as it makes the user program independent of the actual address of DOS or the Bios routines. For example, interrupt number 21 (hex) is the main DOS function dispatcher.

In an ideal ('well behaved') application, the program would not interact with any of these devices. Instead, the application would call DOS, DOS would call the Bios, and the Bios would deal with the devices knowing, as it does, where they are kept, what sort they are, and how to talk to them. However, since DOS (and even the Bios) does not provide the degree of control and flexibility required for many applications, it is often necessary to control the devices directly. While this reduces the degree of portability of your software, it does deliver a higher quality result and since most machines are IBM compatible, does not affect portability too seriously.

Port Address	Function
000 - 00F	DMA Controller
020 - 021	Programmable Interrupt Controller
040 - 043	Timer/Counter
060 - 063	Programmable Peripheral Interface
064	Keyboard Controller
200 - 20F	Game control adapter
2F8 - 2FF	COM2: Serial comms adapter
378 - 37F	Printer adapter
3B0 - 3BF	Monochrome display controller
3D0 - 3DF	Colour/Graphics display controller
3F0 - 37F	Floppy disk controller
3F8 - 3FF	COM1: Serial comms adapter

Fig 3: System I/O port map

Interrupt Number	Function
0	Divide by Zero intercept
1	Single Step intercept
2	Non-Maskable interrupt
3	Breakpoint instruction vector
4	Overflow
5	Print Screen
8	Ticker (18.2 ticks per second)
9	Keyboard interrupt
E	Diskette interrupt
10	Video I/O
11	Configuration determination
12	Memory size determination
13	Diskette I/O
14	Communications I/O
16	Keyboard I/O
17	Printer I/O
19	Bootstrap loader
1A	Time of day
1B	Keyboard interrupt extension
1C	Ticker extension
20	DOS Program terminate
21	DOS Function dispatcher
22	DOS Terminate address
23	DOS Control/break address
24	DOS Critical error handler
25	DOS Absolute disk read
26	DOS Absolute disk write
27	DOS Terminate & stay resident

Fig 4: Interrupt vectors

Software Architecture

On the surface, three levels of software exist. The highest level is the application program, whether it be dBase, Lotus or your latest Turbo Pascal program. The application software is concerned purely with the application — being an accounting system, word processor, or whatever.

Theoretically, when the application needs to do something involving the physical machine — open a file, read the keyboard, allocate memory — it asks DOS to do the work. DOS is accessed through a series of eight interrupts, with one of them, Int 21, providing over 50 different functions. DOS is concerned with files and other system resources in a very device independent fashion. That is, DOS doesn't tell you (or need to be told by you) whether a disk is a floppy or hard disk, how many sectors per track, how many surfaces, what sort of controller chip is being used, and where that chip is located.

DOS consists of two portions: resident and transient. The resident portion loads into low memory and provides the ablity to deal with the DOS requests an application program may issue. The transient portion loads into high memory and provides the command line parser and the ability to load other programs. The transient portion may be overwritten by memory hungry applications, in which case the resident portion can detect the fact and reload COMMAND.COM when it is next needed. One of the cardinal rules of an MS-DOS machine is that you are never allowed to know where in memory MS-DOS is located. In fact, it is possible to find out, but MS-DOS will not necessarily load into the same position each time the system is booted. Since you don't know where MS-DOS lives, access to it is via the software interrupts.

OEMs, manufacturers of compatible PCs, license MS-DOS from Microsoft. Some OEMs make extensive modifications to the standard MS-DOS before distributing it with their machine. For this reason, Microsoft says that there is no such thing as a 'standard' MS-DOS

	15 ← → 0			
AX	AH	AL	ACCUMULATOR	GENERAL PURPOSE REGISTERS
BX	BH	BL	BASE	
CX	CH	CL	COUNT	
DX	DH	DL	DATA	
	7 ← → 0	7 ← → 0		

Register	Name	Group
SP	STACK POINTER	POINTER REGISTERS
BP	BASE POINTER	
SI	SOURCE INDEX	
DI	DESTINATION INDEX	
IP	INSTRUCTION POINTER	
FLAGS	PROCESSOR STATUS	
CS	CODE SEGMENT	SEGMENT REGISTERS
DS	DATA SEGMENT	
SS	STACK SEGMENT	
ES	EXTRA SEGMENT	

PROCESSOR STATUS FLAGS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF

CF — CARRY
PF — PARITY
AF — AUXILIARY CARRY
ZF — ZERO
SF — SIGN
TF — TRAP
IF — INTERRUPT ENABLE
DF — DIRECTION
OF — OVERFLOW

Figure one: The 8086 register structure

It's a mystery why IBM chose the 8088 instead of the 8086. True, the 8086 requires 16 data lines running throughout the machine, but this is trivial.

Since the 8086 can store 16 bits of data at a time, it runs twice as fast as the 8088 where 16-bit operations are involved. For instructions which transfer only 8 bits, the two run at much the same speed. Thus, how the software is written can greatly influence how much faster the 8086 is. Despite this however, the 8088 and 8086 have identical instruction sets and can be considered to be the same chip from the programming point of view. This is why many compatible vendors have chosen the 8086 and have achieved enhanced performance.

The 80186 was not a significant device, mainly being an 8086 with more of the support circuitry 'on-chip', rather than residing in additional packages. This lowered manufacturing costs and meant smaller circuit boards.

The 80286 was a real improvement. It can address up to 16Mbytes of memory and provides hardware support for multi-tasking and multi-user environments, and is the chip chosen for the IBM AT. Moreover, much of the support gear has been moved back off the chip, and additional 'smarts' have been placed in the chip to perform instruction pipe-lining, a feature normally only found in mainframes and minis. Really, the 80286 is two chips in one. It can either be considered as a fast 8086 single user device (as it is in the AT), or can form the basis of a true multi-user system, as could be done in the AT, but no-one is using it that way.

The 80386 is where Intel finally catches up to the old Z8000. This chip out-runs most minicomputers. It can address up to 64 gigabytes of real memory, has on-chip support of virtual memory up to 64 terabytes, and goes faster than many $200,000 mini-

OFFSET REGISTER (eg: IP): 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

+

SEGMENT REGISTER (eg: CS): 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

↓

ACTUAL MEMORY ADDRESS: 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Fig 2: How the segment registers affect memory references

computers. Internally, the chip reeks of being designed for a true mini environment. Intel plans to use the 80386 to get into the computer systems business, and might just do it — the 80386 is a very impressive device.

The 80286 and 80386 both provide instructions not present in the 8088/86; however, they do implement all of the 8088/86 instruction set. This means that (in general), anything written for an 8088/86 will run on an 80286/386. This is why PC software happily chugs away on the AT.

Throughout this series, we will refer to the 8086, but in our context, the comments can also be applied to the 8088, 80186 and 80286.

The 8086

To do anything adventurous with MS-DOS or the Bios generally requires the use of assembler. Except for 'terminate and stay resident' software, the MS-DOS and INTR functions of Turbo Pascal can greatly reduce the need for this, but even so, knowledge of the 8086 structure is a must. Old 8080 and Z80 enthusiasts will find much of this information hauntingly familiar.

The 8086 contains 14 registers, as shown in Fig 1. Although their names reflect specific functions, most of these registers can be used for most activities. For example, arithmetic can be performed on any general purpose or pointer register, and general purpose registers can be used as pointer registers.

The four general purpose registers can be treated either as 16 bits, or as two groups of 8 bits. As the registers are only 16 bits wide, Intel must employ a slight trick to obtain addressability to 1 Mbyte. This is done using the segment registers. Simply, whenever memory is referenced in any way, whether it be by the stack, IP, or a pointer, the 8086 adds one of the segment registers to it. However, the segment register is first multiplied by 16. The effect is shown in Fig 2.

Thus two 16-bit quantities combine to generate a 20-bit effective address, thus providing access to up to 1Mbyte of memory. For example, if the segment register CS contained $1234, and IP contained $4567, then the actual byte of memory accessed would be $168A7. All memory addresses are actually written in segment offset form, for example $1234:4567.

This has two interesting side effects. One is that any unique location in memory can be expressed in 4096 different ways. For example, $1234:4567 and $1334:3567 actually refer to the same memory location. The other is that the scheme tends to break memory up into 16-byte areas, as segment registers can only point to 16-byte boundaries. This quantity of 16-bytes is called a paragraph, and makes relocation a breeze.

Anyone from the Z80 days will remember the problems of relocating a program from one area of memory to another. In the 8086, however, as long as the program does not load absolute values into any of the segment registers, programs can be relocated simply by moving the program to the desired starting paragraph, and then loading the segment registers with new values.

It's important to remember that you simply cannot get at a memory location without first having a segment register pointing to within 64k of that location. This is why four segment registers are provided. Their functions are:

CS — points to the executable code being run
SS — points to the stack
DS — points to the data being operated upon
ES — points to anything else

ES is probably one of the most important registers, as it allows inter-segment operations to be performed.

Each instruction in the 8086 uses a default segment register. For example, CS and IP are generally paired, as are SS and SP. Direct addresses and indexing via most pointer registers default to the DS segment register. Some move and compare instructions using DI and SI default DI to ES and SI to DS. The BP register generally defaults to SS, and this is really neat, as it makes the 8086 very applicable to stack-frame based languages such as Pascal and PL/I.

Importantly, segment register defaults can be overridden. Internally, this is done by a one byte prefix instruction. In assembler, it looks like:

MOV AX, CS:[THING]

The 24 addressing modes are generally broken into seven categories, as shown in Fig 3. Some of these appear to be inconsistent, for example, the machine does not allow a segment register to be loaded with a constant (for good reason). This apparent inconsistency is due to the fact that there are really 24 modes, and they are shown in seven groups.

The instruction set in the 8086 is similar to many other microprocessors. Its distinguishing features include signed and unsigned multiplication and division instructions, and a complete set of conditional jump instructions which take care of different flag combinations and are duplicated for signed and unsigned comparisons. All JMP and CALL instructions exist in different forms (with different lengths), depending on whether the transfer is inter or intra segment.

256 software interrupts are provided. These are very important, as they are the basis of how MS-DOS and the Bios work. Software interrupts are one or two byte instructions which behave much the same as external interrupts. Issuing one

Mode	Operand Format	Default Segment
Register	Reg	None
Immediate	Data	None
Direct	Disp	DS
	Label	DS
Register indirect	[BX]	DS
	[BP]	SS
	[DI]	DS
	[SI]	DS
Base relative	[BX]+Disp	DS
	[BP]+Disp	SS
Direct indexed	[DI]+Disp	DS
	[SI]+Disp	DS
Base indexed	[BX][SI]+Disp	DS
	[BX][DI]+Disp	DS
	[BP][SI]+Disp	SS
	[BP][DI]+Disp	SS

Notes: Reg can be any 8 or 16-bit register, except for IP
Data can be any constant, 8 or 16-bits in length
Disp can any 8 or 16-bit signed displacement value, and is optional for base indexed addressing

Fig 3: The 8086 addressing modes

1.19318MHz. Each counter may be programmed to increase or decrease once every 1 to 65536 clock cycles in either binary or binary coded decimal (BCD). Each counter may be set to one of five modes. In one shot mode, the counter will count down to zero and then trigger its output and go no further. In the rate generator, the usual setting, after triggering its output the chip will reset the count and repeat the operation. In square wave mode, the output remains high for one half of the count and then toggles to low for the other half, resulting an an output which is high and low for equal amounts of time. The other two modes generate strobes, and are rarely used. Each of the timer registers may be read and written while the counting progresses. The device appears at the four I/O ports starting at I/O port address 40h.

The output of the 8253-5 consists of three discrete pins on the chip. Each of these pins is hardwired via tracks on the circuit board to various destinations. Timer 0 output runs to the IRQ 0 input on the interrupt controller (PIC), thus attracting the attention of the CPU when it triggers. Timer 1 feeds to the DMA controller chip, and timer 2 interfaces to the speaker.

Timer 1 is very important and is generally set to trigger quite rapidly. Timer 2 can be used to generate tones on the speaker, commonly set to produce square waves. Note that once you have started timer 2, the speaker will continue to produce the tone until you stop timer 2, and your program may continue with other work. Timer 0 is used for operating system timing purposes, and is initialised by DOS to count as slowly as possible, that is, the timer will trigger 1,193,180/65,536 times per second, or about 18.2Hz.

Port	Function
40h	Programmable Interval Timer (PIT) command word 7 6 5 5 4 2 1 0 SC1 SC2 RL1 RLO M2 M1 MO BCD
41h	PIT Timer 0 read/write
42h	PIT Timer 1 read/write
43h	PIT Timer 2 read/write

Command word bit assignment meanings:

SC1	SCO	
0	0	Select counter 0
0	1	Select counter 1
1	0	Select counter 2

RL1	RLO	
0	0	Latch current count
1	0	Read/write MSB only
0	1	Read/write LSB only
1	1	Read/Write LSB, MSB

M2	M1	MO	
0	0	0	Command
0	0	1	Programmable one-shot
X	1	0	Rate generator
X	1	1	Square wave generator
1	0	0	Software triggered strobe
1	0	1	Hardware triggered strobe

Table 2: I/O ports used in this example

The timer generates a type 8 interrupt, which will be acknowledged by the CPU if all of the enables are set, and if no higher priority interrupt is pending. User programs should not alter the rate at which timer 0 runs, as this will adversely affect timing dependent operations of the computer, such as disk delays.

A signal on the IRQ 0 interrupt input on the PIC causes the CPU to branch to interrupt vector number 8. The handler for this vector takes care of the various house keeping tasks the operating system performs periodically, such as timing out the disk drive and incrementing the time. After the handler completes its work, it executes a software interrupt instruction to branch to vector number 1Ch. This vector normally just returns without doing anything, but user programs may redirect the vector to themselves thereby providing them with a 'heart beat'.

The SPEED utility operates by redirecting interrupt 1Ch to a small resident handler which loops, continuously looking at the timer 0 register, and not returning until the timer has reached a certain point in its count down to the next trigger point.

The program

As we have now examined most of the system resources used by the example program, let us now look towards Listing 1, the assembly language source code of SPEED.COM.

The entire program is defined to lie within a single segment, and the assembler is to assume that the CS and DS registers point to the start of that segment.

The first fragment of code is the interrupt 1Ch handler itself. The code commences with a marker to detect an already resident version of the utility, and a branch around the marker. The handler then outputs a zero to port 43h, which tells the 8253-5 PIT chip that we want to read the current contents of the timer zero register. We then read the 16 bit counter from port 40h, LSB first, then MSB. A compare instruction checks that the counter has reached the required point in its count back to zero, and the program loops if it has not. Note that the compare instruction does not test for exact equality, as it is possible that the program may not access the counter at the precise instant that it holds the required value. When the counter has reached the required point, the handler performs an inter-segment jump to wherever the 1Ch vector used to point before SPEED was loaded. This allows