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

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

```
: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.

```
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

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:

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

```
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 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.

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

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 subrange 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    GOTO 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 subtasks: 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.

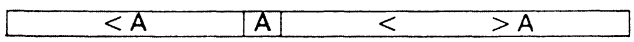
From the point of view of our general repeated division method, binary search is an example of an improvement on an O(N) method — linear search — that can be implemented as $\log_2 N$ divisions in which each takes a time that doesn't depend upon N. Binary search is a very powerful and general algorithm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

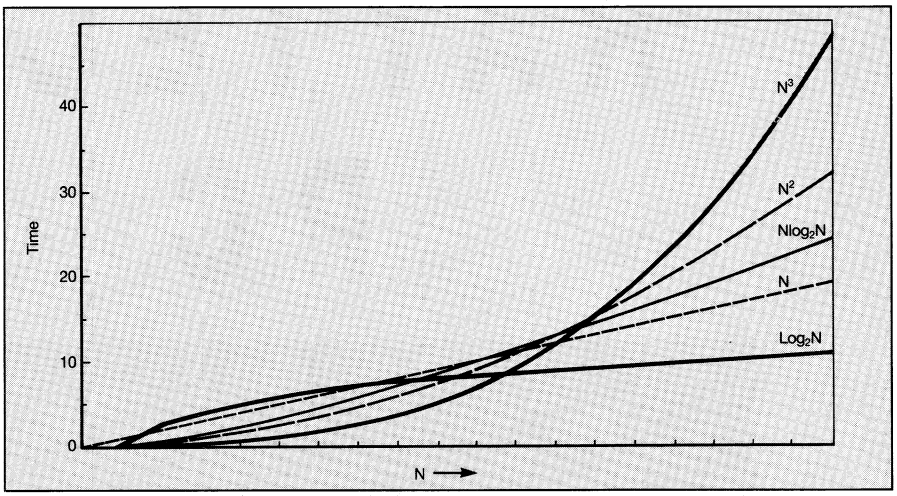
For example, an exponential time algorithm $O(e^N)$ performs worse than any polynomial time algorithm. In other words, $O(e^N)$ is another order of badness! Most of the magic algorithms described in this article take time proportional to either $O(N\log_2 N)$ or $O(\log_2 N)$. An $O(N\log_2 N)$ algorithm 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:

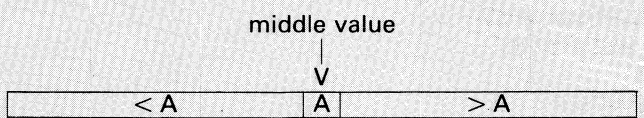
2 3 4 7 10 18 60

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.

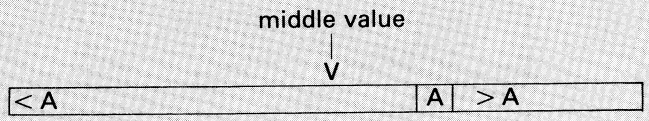
The most common way of finding the median is to proceed as above and sort the numbers into order before finding the middle value. If the best sorting method is used, this takes $O(N\log_2 N)$, but there is a much faster method that will find the median in time proportional to O(N) based on the partitioning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

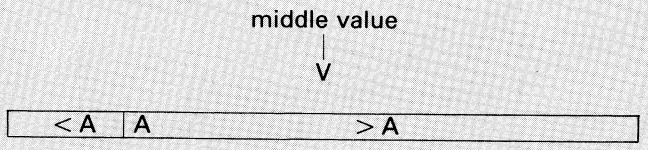
If this division is into two equal parts, then A is the median of the set of values:



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

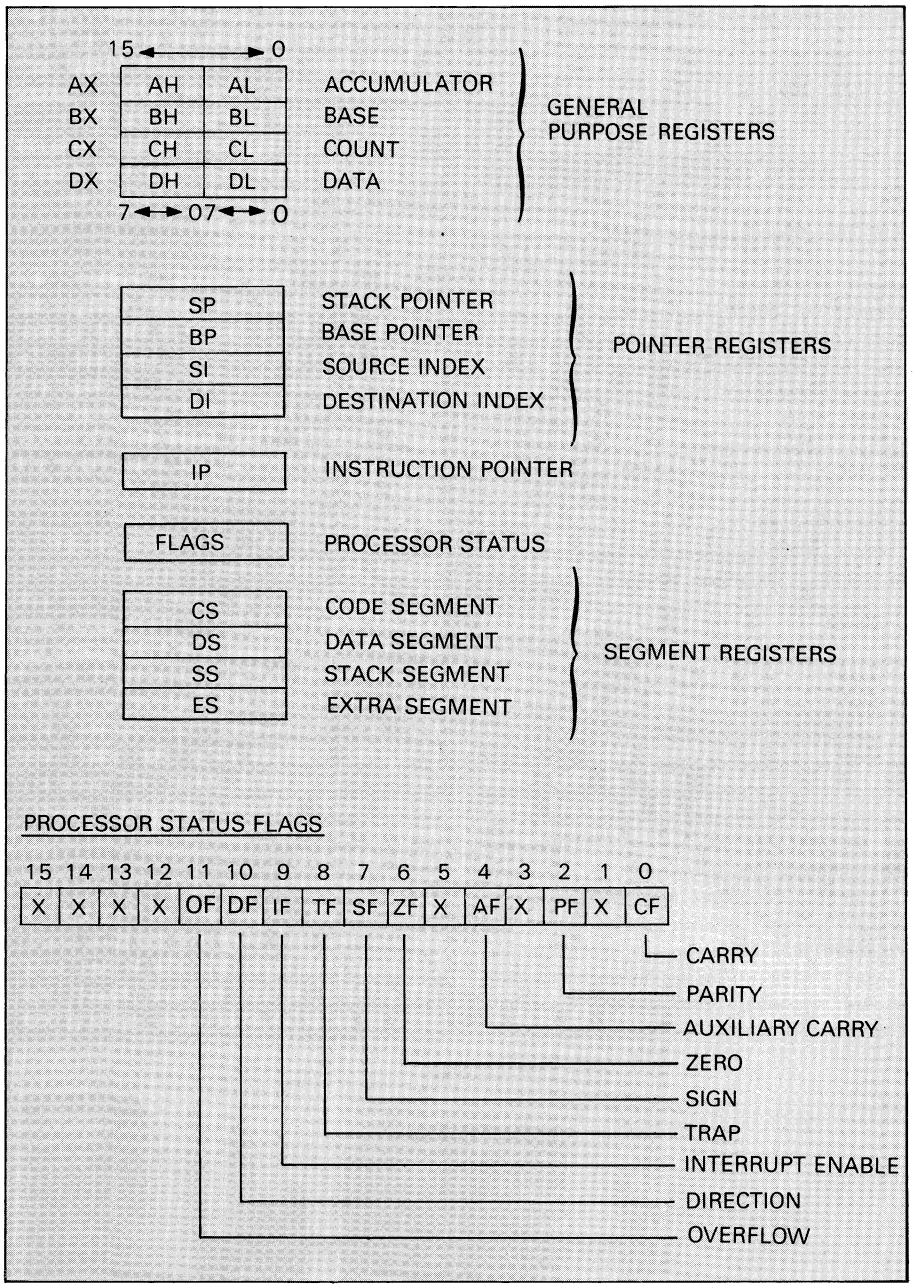


Figure one: The 8086 register structure

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-

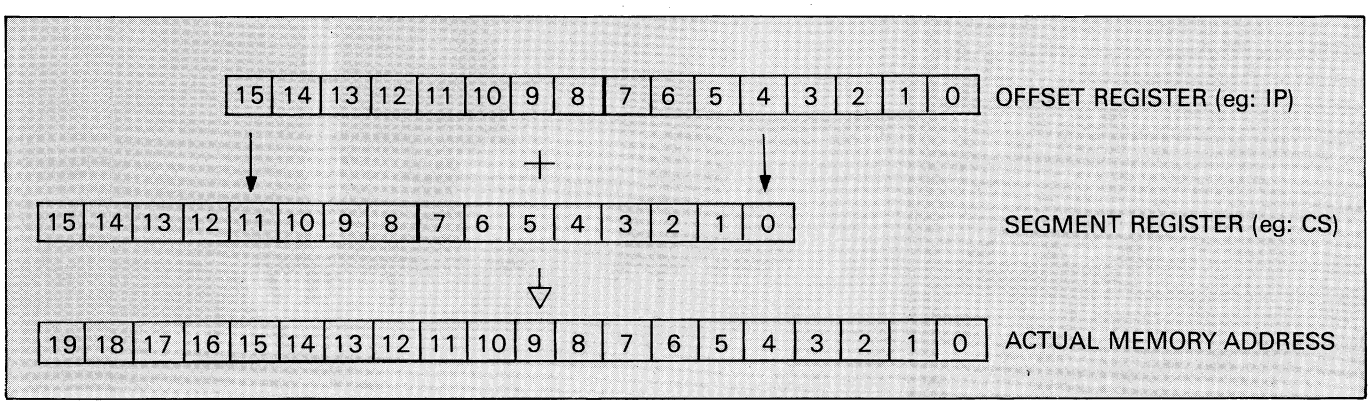


Fig 2: How the segment registers affect memory references

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 1Mbyte. 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. 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.

Port	Function
40h	Programmable Interval Timer (PIT) command word 7 6 5 5 4 2 1 0 SC1 SC2 RL1 RLO M2 M1 M0 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	SC0	
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	M0	
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 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

from the .COM file. For example, this program uses a 2000 byte screen buffer. If the buffer were defined to the assembler as "DB 2000 dup (?)", then the .COM file would include 2000 bytes of uncommitted junk. By defining these work areas relative to the current IP in the assembler, that is, relative to the '$' symbol, the work areas do not have to be written to the object file. The resultant work area will overlay either program code or other areas in memory, and it is the programmers responsibility to ensure that nothing comes to harm.

The PrtDisk utility not only redefines its initialisation code to be work area, but also extends the work area past the physical end of the program. The astute reader will have noticed that the program is actually making use of a data area which overlays the initialisation code before the initialisation code has completed. This is risky stuff, and therefore the very essence of any decent assembly language program. Nevertheless, it works, simply because *that* area of the initialisation code which is being overwritten *should* have been finished with by the time it gets overwritten. Very efficient stuff, but I wouldn't run an airline that way.

The dynamic work area is used to hold the file name specified on the command line. This file name must be fully qualified. That is, it must include the drive, directory and name of the file to receive the output. This is because the user may change directories, or even default drives between screen dumps, and we need the output to be written in the one place regardless of default setting. That is, the program should be started with a command such as:

PRTDISKC:\DUMPS\SCREENS.DAT

and not just:

PRTDISKSCREENS.DAT

Once the file name has been copied along with a terminating zero to make it a valid DOS ASCIIZ string (ASCII delimited with a zero), function call 3ch is invoked to create the file. Any existing file of the same name is emptied of its contents. If the create fails (carry flag set), then the program prints a message and terminates. Otherwise function 25h is used to set vector 25h to the new print screen handler, the existing vector 28h is saved, and replaced with a pointer to the new INT 28h handler. Finally a completion message is printed, and the program loads register DX with the offset of its last important byte, plus 80h paragraphs to reserve dynamic work space, and terminates while staying resident.

Bios INT Number	Input conditions	Output conditions
11h	None	AX set with hardware configuration: Bits 14,15 — Number of printers Bit 12 — Game port Bits 11-09 — Number of RS232 ports Bits 07,06— Number of diskettes Bits 05,04 — Initial video mode 01 40x25 graphics 10 80x25 graphics 11 Monochrome Bits 03,02 — RAM on motherboard Bit 00 — Diskettes present
10h	AH=00	Graphics mode set to AL
	AH=01	Cursor size set to CL
	AH=02	Cursor position set to BH,DX
	AH=03	DX,CX returns cursor position
	AH=04	CX,DX,BX returns light pen position
	AH=05	Active page set to AL
	AH=06	Scroll up by AL,CX,DX,BX
	AH=07	Scroll down by AL,CX,DX,BX
	AH=08	AX holds byte and attribute at cursor
	AH=09	AL,BL is written at cursor CX times
	AH=0a	AL is written at cursor CX times
	AH=0b	Palette set to BX
	AH=0c	Pixel written at CX,DX
	AH=0d	Pixel read from CX,DX
	AH=0e	Print character in AL
	AH=0f	Current video state in AX

Table 2: Bios functions used in this example

The existing value of vector 28h is stored since the new handler should chain to whatever used to be the INT 28h handler, thus providing more chance that other resident software will coexist with PrtDisk. The existing vector 05h pointer is not stored, as we do not want to chain to it. To do so would result in the screen being both written to disk and dumped to the printer.

From this point, we can look to the resident portion of the program, particularly the INT 05h (print screen) handler. This portion commences by saving all the registers (using the PUSHES macro) and enabling interrupts thereby ensuring that serial communications and timer interrupts will be serviced during the potentially lengthy routine. It also copies the CS code segment register to the DS data segment register to establish addressability, as seen last month.

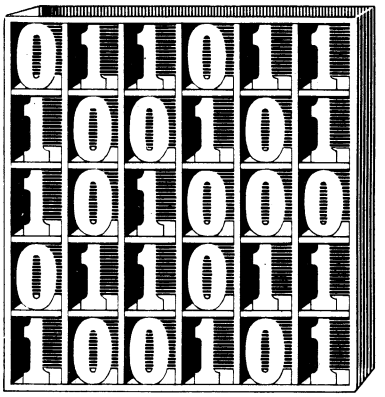
Its next step is to see if it has a full buffer already. The buffer can hold only one screen full, and hopefully should be emptied immediately. However, it is possible that DOS will not have been in a suitable state when the INT 05h first occurred, and it is also possible that the buffer will not have been emptied before the *next* INT 05h occurs. For this reason, if the routine is called when the buffer is still full, it simply beeps at the user and returns, popping the registers from the stack and executing an IRET (return from interrupt) instruction.

There are many ways in which a beep may be produced, the easiest of which is using DOS function 09h (print) to display the ASCII code 07h. This is a definite 'no no', as we can be sure that the system is in an unstable state, and so should avoid issuing a DOS function call, even a trivial one.

The hardest way to generate a beep is manipulating the timer and speaker ports directly. Perhaps the easiest and most robust method is to call the Bios print routine. This is performed by executing an INT 10h instruction, accessing the Bios video I/O handler, loading register AX with 0eh (the command code to print a character), and AL with 07h (the character to print). The Bios video I/O handler is quite powerful, providing a graphics interface, display attribute control, and all sorts of other nifty features missing from DOS except through the unwieldy ANSI.SYS.

If the buffer is not already full, the program loads registers DS:SI with the address of the first byte of the 'regen'

SUBSET



David Barrow presents more documented machine code routines and useful information for the assembly language programmer. If you have a good routine, an improvement or conversion of one already printed, or just a helpful programming hint, then send it in and share it with other programmers. Subroutines for any of the popular processors and computers are welcome but please include full documentation. All published code will be paid for. Send your contributions to Subset, APC, 2nd Floor, 215 Clarence Street, Sydney 2000.

Z80 32-BIT ARITHMETIC

The first six datasheets this month, from T Sullivan, form the basis of an excellent but unusual 32-bit integer arithmetic suite. The seventh datasheet, POWX, calculates positive integer powers. While not matching the speed and efficiency of the 68000's single instructions, the suite demonstrates that higher-precision arithmetic on the Z80 can be reasonably short and fast when all options are carefully thought out.

The suite's internal operations act on arguments contained in registers. This is much faster than operations on values held in memory. However, as far as the calling program is concerned, each argument or variable resides in four bytes of memory and is addressed by one of the register pairs: HL, BC or DE.

The storage order in memory reverses that normally found in Z80 code, and has the high-order bytes first in low memory. This method is unusual, but does have much to recommend it. I can think of two major advantages: numbers are easier to read in a dump; and the variable's sign bit resides at the addressed byte, enabling quick sign reference. Although Z80 purists might object, the only disadvantages are quite niggling: the four bytes have to be picked up individually rather than by two 16-bit loads, which cause reverse ordering in the registers; and any initial values have to be assembled by four DEFBs rather than by two DEFWs.

More unusually, the values are not stored or acted upon as two's complement numbers, but are kept in memory as sign bit and 31-bit magnitude (absolute value). All routines in the suite extract the sign bits from the arguments and work separately on the positive values and signs.

Simple addition and subtraction is slowed considerably by using the signed magnitude format, but there are obvious benefits for multiplication, division and exponentiation, which operate on absolute values. Sign bit extraction and manipulation is far quicker than testing input arguments and output results for possible 4-byte negation.

The most unusual aspect is the use throughout the suite of the Z80's alternate register set — AF, BC, DE and HL. Without this extra register bank the Z80 has only A, F and five 16-bit registers, and this number is not sufficient for most 32-bit arithmetic operations. With the extra set, a Z80 programmer has eight 16-bit registers, two 8-bit accumulators and two sets of flags at his or her disposal, though not all are available simultaneously.

Use of the alternate registers is frowned upon in SubSet, since some operating systems reserve their use for fast interrupts. It takes eight program bytes, eight stack bytes and 84 clock cycles (time states) to PUSH and POP AF, BC, DE and HL, but switching between register sets can save and restore the interrupted state in only four program bytes, zero stack bytes and 16 clock cycles. The suite has a non-interruptable classification, since its routines rely on the alternate registers and could find their data suddenly and inexplicably corrupted by a previously transparent interrupt. Nevertheless, it should present no problems in situations where interrupts can be guaranteed not to corrupt any registers.

Structure: The suite has eight main entry points — ADDX, SUBX, CMPX, MULX, DIVX, MODX, SQX and POWX, acting on signed variables, and 14 internal subroutines dealing with I/O and magnitude operations. Most of the main routines are reduced to a short sequence of subroutine calls by the highly-structured approach. However, I think the structure is *too* rigid, giving rise to some time-consuming anomalies. For example, SQX requests XENTRY to read and process a variable addressed by BC when only that addressed by DE is required, and the signed comparison, CMPX, might be achieved more efficiently by using the accumulator A to compare bytes still resident in memory.

XENTRY does five separate jobs. At the cost of a few extra bytes, a set of six routines to: (1) save registers; (2 to 5) load a variable to any of BCBC, DEDE, HLHL or IXIY while extracting the sign bit to Carry Flag; and (6) shift Carry into A, storing the parity of 7-bit A in bit 7, A, would have greater flexibility, and so be far more useful to any additions to the suite.

DATASHEET 1

```
;> XENTRY - Entry routine for 32-bit signed arithmetic routines.
;> XEXIT  - Exit routine for 32-bit signed arithmetic routines.
;> RESTOR - Register restore routine.

;JOB         To provide entry and exit facilities for 32-bit
;           arithmetic suite; saving registers and loading
;           arguments; storing result and restoring registers.
;           XENTRY is CALLED by arithmetic routines.
;           XEXIT & RESTOR are JUMPED TO by arithmetic routines.
;ACTION      See individual routine comments.

;CPU         Z80
;HARDWARE    Variables in RAM.
;SOFTWARE    CLRH, NEGH.

;INPUT       XENTRY:  DE & BC address the two arguments.
;                     HL addresses the result variable.
;            XEXIT:   Cy=0: HLHL' contains valid result.
;                          Bit 7,A contains result sign.
;                     Cy=1: No valid result.
;OUTPUT      XENTRY:  All registers & flags stacked.
;                     Result address on top of stack.
;                     DEDE' = magnitude of input (DE...DE+3)
;                     BCBC' = magnitude of input (BC...BC+3)
;                     HLHL' = 0
;                     Bit 7,A = sign DEDE' XOR sign BCBC'
;                     Bit 6,A = sign DEDE'
;                     Bit 5,A = sign BCBC'
;                     Bits 4-0,A = 0.       Cy = 0
;                     IX & IY changed.
```

```
;            XEXIT:    Cy=1: Jump to RESTOR.
;                      Cy=0: Sign + magnitude of result
;                            stored to address from stack top.
;                            Z = result zero status.
;                            Fallthrough to RESTOR.
;            RESTOR:   All registers and alternate F restored.
;                      Normal F preserves output from routines.
;ERRORS      None.
;REG USE     F BC DE HL (suite use of all registers).
;STACK USE   XENTRY: 24 (leaves 22 bytes on stack at return).
;            XEXIT: 2 (then RESTOR).
;            RESTOR: -22.
;RAM USE     None.
;LENGTH      XENTRY: 71.  XEXIT: 28.  RESTOR: 19.
;CYCLES      Not given.
;-------------------------------------------------------------
;CLASS 2     *discreet      -interruptable      *promable
;*-**-*      *reentrant     -relocatable        *robust
;=============================================================
;
XENTRY EX   (SP),IY   ;Save IY getting return address.  FD E5
       PUSH IX        ;Save IX.                         DD E5

       PUSH HL        ;Save normal regs, HL, DE & BC,   E5
       PUSH DE        ;as addresses of variables,       D5
       PUSH BC        ;(HL) = (DE) op'n (BC).           C5
       PUSH AF        ;Save normal accumulator & flags. F5
       EXX            ;Access alternate registers.      D9
       PUSH HL        ;Save alternate regs,             E5
       PUSH DE        ;HL', DE' & BC'.                  D5
       PUSH BC        ;                                 C5
       EXX            ;Access normal regs.              D9
       EX   AF,AF'    ;Access alternate acc. & flags.   08
       PUSH AF        ;Save AF'.                        F5
       EX   AF,AF'    ;Access normal AF.                08
       PUSH HL        ;Save result address to t-o-s.    E5
       PUSH IY        ;Push return address for return.  FD E5
;
;...get two arguments (DE) & (BC) to regs & clear HLHL'.
;
       CALL CLRH      ;Clear 32-bit accumulator HLHL'.  CD lo hi
       PUSH DE        ;Move arg1 address from DE        D5
       POP  IX        ;to IX.                           DD E1
       PUSH BC        ;Move arg2 address from BC        C5
       POP  IY        ;to IY.                           FD E1
       LD   D,(IX+0)  ;Load arg1 hi-word to DE.         DD 56 00
       LD   E,(IX+1)  ;                                 DD 5E 01
       LD   B,(IY+0)  ;Load arg2 hi-word to BC.         FD 46 00
       LD   C,(IY+1)  ;                                 FD 4E 01
       EXX            ;Access alternate regs.           D9
       LD   D,(IX+2)  ;Load arg1 lo-word to DE'.        DD 56 02
       LD   E,(IX+3)  ;                                 DD 5E 03
       LD   B,(IY+2)  ;Load arg2 lo-word to BC'.        FD 46 02
       LD   C,(IY+3)  ;                                 FD 4E 03
       EXX            ;Access normal regs.              D9
;
;...get signs: arg1 XOR arg2, arg1, arg2. Clear arg sign bits.
;
       RLC  B         ;prepare for clearing signs       CB 00
       RLC  D         ;while copying to A.              CB 02
       LD   A,D       ;Get sign DE XOR sign BC          7A
       XOR  B         ;in 0,A. Shift it to 1,A and      A8
       RLA            ;eventually into 7,A.             17
       SRL  B         ;Clear sign BC while copying      CB 38
       RRA            ;it eventually to bit 5,A.        1F
       SRL  D         ;Clear sign DE while copying      CB 3A
       RRA            ;it eventually to bit 6,A.        1F
         RRA            ;Get XOR sign to bit 7,A.         1F
         AND  0E0H      ;Mask out non-sign bits and       E6 E0
         RET            ;exit to arithmetic routine.      C9
; :
;-------------------------------------------------------------
;
XEXIT  JR   C,RESTOR  ;Invalid result if Cy set.        38 1A
;
       EX   (SP),IX   ;Get result addr. (SP unchanged). DD E3
       CALL NEGH      ;Test if zero, result in DE,      CD lo hi
       AND  H         ;Clear sign if HLHL' was 0.       A4
       AND  80H       ;Isolate sign and merge result    E6 80
       OR   D         ;hi-byte with sign.               B2
       LD   (IX+0),A  ;Store result hi-byte + sign      DD 77 00
       LD   (IX+1),E  ;and 2nd byte.                    DD 73 01
       EXX            ;Access alternate regs.           D9
       LD   (IX+2),D  ;Store result lo-word.            DD 72 02
       LD   (IX+3),E  ;                                 DD 73 03
       EXX            ;Access normal regs.              D9
       SRL  H         ;Bit 7,H set if result NOT zero,  CB 3C
       OR   H         ;so clear Cy & set S & Z status.  B4

;
;-------------------------------------------------------------
;
RESTOR POP  BC        ;Tidy result address from stack.  C1
       EX   AF,AF'    ;Access alternate accumulator     08
       POP  AF        ;and flags, and restore AF'.      F1
       EX   AF,AF'    ;Access normal AF.                08
       EXX            ;Access alternate regs.           D9
       POP  BC        ;Restore alternate registers,     C1
       POP  DE        ;BC', DE & HL'.                   D1
       POP  HL        ;                                 E1
       EXX            ;Access normal regs.              D9
       POP  BC        ;Pop stacked input AF and         C1
       LD   A,B       ;restore A without changing F.    78
       POP  BC        ;Restore normal registers,        C1
       POP  DE        ;BC, DE & HL, to addresses of     D1
       POP  HL        ;variables.                       E1
       POP  IX        ;Restore index registers.         DD E1
       POP  IY        ;                                 FD E1
       RET            ;Exit arithmetic suite.           C9
;=============================================================
```

DATASHEET 2

```
;= ADDX   - 32-bit signed integer addition.
;= SUBX   - 32-bit signed integer subtraction.
;= CMPX   - 32-bit signed integer comparison.
;=============================================================
;JOB        32-bit (sign bit + 31-bit magnitude) addition,
;           subtraction and comparison.
;ACTION     See individual routine comments.
;-------------------------------------------------------------
;CPU        Z80
;HARDWARE   Variables in RAM.
;SOFTWARE   XENTRY, XEXIT, RESTOR, EXDH, NEGH, ADDHB, SUBHB.
;-------------------------------------------------------------
;INPUT      ADDX/SUBX: Three variables at (HL), (DE) & (BC).
;           CMPX:      Two variables at (DE) & (BC).
;OUTPUT     ADDX/SUBX: Cy=1: magnitude overflow.
;                      Cy=0: (HL) = (DE) + or - (BC).
;                            S, Z return sign & zero status.
;           CMPX:      Z=1: Cy=0:  (DE) = (BC)
;                      Z=0: Cy=0:  (DE) > (BC)
;                      Z=0: Cy=1:  (DE) < (BC)
;ERRORS     None.
;REG USE    F BC DE HL
;STACK USE  26
;RAM USE    None.
;LENGTH     ADDX/SUBX: 38.  CMPX: 32.
;CYCLES     Not given.
;-------------------------------------------------------------
;CLASS 2     *discreet      -interruptable      *promable
;*-**-*      *reentrant     -relocatable        *robust
;=============================================================
;
ADDX   CALL XENTRY    ;Save regs. & get arguments.      CD lo hi
       JR   ASX       ;Go to common ADD/SUB code.       18 05
;
SUBX   CALL XENTRY    ;Save regs. & get arguments.      CD lo hi
       XOR  80H       ;Complement XOR sign for sub.     EE 80
;
ASX    CALL EXDH      ;Move 1st argument to HLHL'.      CD lo hi
       RLA            ;Get XOR sign, arg1 sign to 7,A.  17
       JR   C,ASSUB   ;Subtract if signs different.     38 09
;
       CALL ADDHB     ;Add magnitudes arg2 to arg1.     CD lo hi
       RLC  H         ;Bit 7,H set if overflow, so      CB 04
       SRL  H         ;convert to Cy & clear 7,H.       CB 3C
       JR   SAXEND    ;Go exit, store if no carry.      18 0A

ASSUB  CALL SUBHB     ;Sub magnitudes arg2 from arg1.   CD lo hi
       JR   NC,SAXEND ;Exit okay if sub went.           30 05
       CALL NEGH      ;Else negate and swap result      CD lo hi
       XOR  80H       ;sign, clearing Cy.               EE 80
;
SAXEND JMP  XEXIT     ;Exit, add or sub done.           C3 lo hi
;
;-------------------------------------------------------------
;
CMPX   CALL XENTRY    ;Save regs. & get arguments.      CD lo hi
       RLA            ;Get arguments XOR sign           17
       JR   NC,CXSE   ;and skip if signs are equal.     30 03
;
       RL   A         ;Signs different, sign arg1 gives CB 17
       JR   CXEND     ;correct Cy result, Z reset.      18 13
;
CXSE   CALL EXDH      ;Move 1st argument to HLHL'.      CD lo hi
       CALL SUBHB     ;Sub magnitudes arg2 from arg1.   CD lo hi
       RLA            ;Arg1 sign to Cy, sub result sign 17
       ADC  A,0       ;sign to 1,A. Change result sign  CE 00
       OR   0FEH      ;if arg1 -ve & set before mask.   F6 FE
       CALL NEGH      ;Set Cy=0 only if HLHL' is zero   CD lo hi
       SBC  HL,HL     ;Propogate Z status through H     ED 62
       AND  H         ;reset all bits if HLHL' was 0    A4
       RR   A         ;and set correct Cy and Z flags.  CB 1F
;
CXEND  JMP  RESTOR    ;Exit returning flags only.       C3 lo hi
;=============================================================
```

DATASHEET 3

```
;= MULX   - 32-bit signed multiplication.
;> MULSUB - 31-bit magnitude multiplication.
;=============================================================
;JOB        32-bit (sign bit + 31-bit magnitude) multiplication.
;ACTION     Binary long multiplication (see comments).
;-------------------------------------------------------------
;CPU        Z80
;HARDWARE   Variables in RAM.
;SOFTWARE   XENTRY, XEXIT, LWRLD, LWRLH, ADDHB.
;-------------------------------------------------------------
;INPUT      MULX:    Three variables at (HL), (DE) & (BC).
;           MULSUB:  Two 31-bit magnitudes in DEDE' & BCBC'
;                    HLHL' = 0.
;OUTPUT     MULX:    Cy=1: magnitude overflow.
;                    Cy=0: (HL) = (DE) * (BC).
;                          S, Z return sign & zero status.
;           MULSUB:  S=1:  overflow.
;                    S=0:  HLHL' = DEDE' * BCBC'.  DEDE' = 0.
;ERRORS     None.
;REG USE    MULX:    F BC DE HL
;           MULSUB:  F BCDEHL AF' BCDEHL'
;STACK USE  MULX: 26.  MULSUB: 2.
;RAM USE    None.
;LENGTH     MULX: 13.  MULSUB: 27.
;CYCLES     Not given.
```

```
;-------------------------------------------------------------------------
;CLASS 2      *discreet       -interruptable      *promable
;*-**-*       *reentrant      -relocatable        *robust
;=========================================================================
;
MULX   CALL XENTRY     ;Save regs. & get arguments.      CD lo hi
       CALL MULSUB     ;Do magnitude multiplication.     CD lo hi
       RLC  H          ;Bit 7,H set if overflow, so      CB 04
       SRL  H          ;convert to Cy & clear 7,H.       CB 3C
       JMP  XEXIT      ;Exit, storing product if Cy=0.   C3 lo hi
;
;-------------------------------------------------------------------------
;
MULSUB OR   A          ;Clear Cy and shift multiplier    B7
       CALL LWRLD      ;magnitude up to highest bits.    CD lo hi
       EX   AF,AF'     ;Access alternate accumulator     08
       LD   A,31       ;set 31-bit multiplier count.     3E 1F
;
MSLOOP EX   AF,AF'     ;Access normal flags.             08
       CALL LWRLH      ;Product to next bit place.       CD lo hi
       RET  M          ;Exit S=1 if 31-bit overflow.     F8
       CALL LWRLD      ;Get next multiplier bit,         CD lo hi
       JR   NC,MSLT    ;skip if not set else add         30 03
       CALL ADDHB      ;multiplicand at this bit place   CD lo hi
       RET  M          ;but exit S=1 if overflow.        F8
MSLT   EX   AF,AF'     ;Access count in A'.              08
       DEC  A          ;Count off this bit place done    3D
       JR   NZ,MSLOOP  ;and repeat until product.        20 EE
;
       EX   AF,AF'     ;Access normal AF and exit with   08
       RET             ;HLHL'product, DEDE'=0, S=0.      C9
;=========================================================================
```

DATASHEET 4

```
;=========================================================================
;= DIVX   - 32-bit signed integer division.
;= MODX   - 32-bit signed modulus (division remainder).
;> DIVSUB - 31-bit magnitude division (integer & remainder).
;=========================================================================
;JOB          32-bit (sign bit + 31-bit magnitude) division
;             quotient or division remainder.
;ACTION       Binary long division (see comments).
;-------------------------------------------------------------------------
;CPU          Z80
;HARDWARE     Variables in RAM.
;SOFTWARE     XENTRY, XEXIT, LWRLD, LWRLH, ADDHB, SUBHB, CLRH,
;             EXDH.
;-------------------------------------------------------------------------
;INPUT        DIVX/MODX:   Three variables at (HL), (DE) & (BC).
;             DIVSUB:      Two 31-bit magnitudes in DEDE' & BCBC'
;                          HLHL' = 0.
;OUTPUT       DIVX:   Cy=1: division by zero.
;                     Cy=0: (HL) = integer quotient (DE) / (BC).
;                           S, Z return sign & zero status.
;             MODX:   Cy=1: division by zero.
;                     Cy=0: (HL) = integer remainder (DE) / (BC).
;                           S, Z return sign & zero status.
;             DIVSUB: Cy=1:  division by zero
;                     Cy=0:  DEDE' remainder HLHL'= DEDE' / BCBC'.
;ERRORS       None.
;REG USE      DIVX/MODX: F BC DE HL
;             DIVSUB:    F BCDEHL AF' BCDEHL'
;STACK USE    DIVX: 26.  MODX: 26.  DIVSUB: 2.
;RAM USE      None.
;LENGTH       DIVX: 12.  MODX: 10.  DIVSUB: 36.
;CYCLES       Not given.
;-------------------------------------------------------------------------
;CLASS 2      *discreet       -interruptable      *promable
;*-**-*       *reentrant      -relocatable        *robust
;=========================================================================
;
DIVX   CALL XENTRY     ;Save regs. & get arguments.      CD lo hi
       CALL DIVSUB     ;Do magnitude division.           CD lo hi
       CALL EXDH       ;Move quotient to HLHL'           CD lo hi
       JMP  XEXIT      ;Exit, storing quotient if Cy=0.  C3 lo hi
;
;-------------------------------------------------------------------------
;
MODX   CALL XENTRY     ;Save regs. & get arguments.      CD lo hi
       RLA             ;Get remainder sign to 7,A.       17
       CALL DIVSUB     ;Do magnitude division.           CD lo hi
       JMP  XEXIT      ;Exit, storing remainder if Cy=0. C3 lo hi
;
;-------------------------------------------------------------------------
;
DIVSUB CALL SUBHB      ;Sub will reset Cy if divisor     CD lo hi
       CCF             ;is zero, so swap Cy state and    3F
       RET  C          ;exit early if division by 0.     D8
;
       CALL CLRH       ;Re-zeroise remainder, HLHL'.     CD lo hi
       EX   AF,AF'     ;Access alternate accumulator     08
       LD   A,31       ;and set quotient count in A'.    3E 1F
       OR   A          ;Clear Cy and shift d'dend up     B7
       CALL LWRLD      ;to highest bit of DEDE' then     CD lo hi
       CALL LWRLD      ;shift out first d'dend to Cy.    CD lo hi
;
DSLOOP CALL LWRLH      ;Shift next bit to remainder      CD lo hi
       CALL SUBHB      ;and try to subtract divisor      CD lo hi
       CALL C,ADDHB    ;add back if won't go and set     DC lo hi
       CCF             ;Cy for result. Shift in result   3F
       CALL LWRLD      ;bit and get next dividend bit.   CD lo hi
       DEC  A          ;Repeat for 31-bit quotient       3D
       JR   NZ,DSLOOP  ;leaving 31-bit remainder         20 F0
;
       EX   AF,AF'     ;Access normal AF.                08
DSEND  RET             ;Exit, DEDE' quot., HLHL' rem.    C9
;=========================================================================
```

DATASHEET 5

```
:=============================================================
:= SQX     - 32-bit signed square.
:> SQSUB   - 31-bit magnitude square.
:=============================================================
:JOB         32-bit (sign bit + 31-bit magnitude) square.
:ACTION      Binary long multiplication (see comments).
:-------------------------------------------------------------
:CPU         Z80
:HARDWARE    Variables in RAM.
:SOFTWARE    XENTRY, XEXIT.
:-------------------------------------------------------------
:INPUT       SQX:   Two variables at (HL) & (DE).
:            SQSUB: 31-bit magnitude in DEDE'.
:                   HLHL' = 0.
:OUTPUT      SQX:   Cy=1: magnitude overflow.
:                   Cy=0: (HL) = (DE)^2.
:                         S, Z return sign & zero status.
:            SQSUB: S=1:  magnitude overflow.
:                   S=0:  HLHL'= DEDE'^2.  DEDE' unchanged.
:ERRORS      None.
:REG USE     SQX:   F DE HL
:            SQSUB: F DEHL DEHL'
:STACK USE   SQX: 26.  SQSUB: 2.
:RAM USE     None.
:LENGTH      SQX: 10.  SQSUB: 23.
:CYCLES      Not given.
:-------------------------------------------------------------
:CLASS 2      *discreet      -interruptable      *promable
:*-**-*       *reentrant     -relocatable        *robust
:=============================================================
:
SQX     CALL XENTRY     :Save regs. & get root in DEDE'.  CD lo hi
        XOR  A          :Clear sign, squares positive.    AF
        CALL SQSUB      :Square magnitude and exit        CD lo hi
        JMP  XEXIT      :storing square if Cy=0.          C3 lo hi
:
:-------------------------------------------------------------
:
SQSUB   OR   A          :Clear Cy, negate root hi-word    B7
        SBC  HL,DE      :setting S only if word>0. Exit   ED 52
        RET  M          :S=1 if square will overflow.     F8
:
        EXX             :Access alternate regs.           D9
        PUSH DE         :and get 16-bit root, DE'.        D5
        EXX             :Access normal regs. and get      D9
        POP  HL         :root in HL as multiplier.        E1
        LD   D,16       :Set 16-bit multiplier count.     16 10
:
SQLP    ADC  HL,HL      :Next m'plier out, product in.    ED 6A
        JR   NC,SQLT    :Skip if no add this bit place.   30 03
        EXX             :Else access alternate regs. and  D9
        ADD  HL,DE      :add m'cand to product lo-word.   19
        EXX             :Access normal regs.              D9
SQLT    DEC  D          :Count off one multiplier bit     15
        JR   NZ,SQLP    :and repeat for 16-bits.          20 F6
:
        ADC  HL,HL      :Final Cy to product hi-word.     ED 6A
        RET             :Exit, HLHL square, DEDE' root.   C9
```

DATASHEET 6

```
:=============================================================
:> CLRH   - Clear 32-bit accumulator.
:> NEGH   - Negate (2s complement) 32-bit accumulator.
:> EXDH   - Swap 32-bit register DEDE & 32-bit accumulator.
:> ADDHB  - Add register BCBC to accumulator.
:> SUBHB  - Subtract register BCBC from accumulator.
:> LWRLD  - Rotate left register DEDE.
:> LWRLH  - Rotate left accumulator.
:> LWSLAX - Arithmetic shift left register IXIY.
:=============================================================
:JOB         Manipulation of 32-bit registers.
:ACTION      See individual routine comments.
:-------------------------------------------------------------
:CPU         Z80
:HARDWARE    None.
:SOFTWARE    None.
:-------------------------------------------------------------
:INPUT       32-bit values in named registers (see routines).
:            Cy flag status in rotate operations.
:OUTPUT      32-bit values in named registers (see routines).
:            Cy from arithmetic, shift and rotate operations.
:ERRORS      None.
:REG USE     F BCDEHL BCDEHL' IX IY (see individual routines).
:STACK USE   None.
:RAM USE     None.
:LENGTH      (73)  CLRH: 9.  NEGH: 16.  EXDH: 5.  ADDHB:  6.
:                  SUBHB: 8.  LWRLD: 11.  LWRLH: 7.  LWSLAX: 11.
:CYCLES      Not given.
:-------------------------------------------------------------
:CLASS 2      *discreet      *interruptable      *promable
:*****-       *reentrant     *relocatable        -robust
:=============================================================
:
CLRH    LD   HL,0       :Clear hi-word.                   21 00 00
        EXX             :Access alternate regs.           D9
        LD   HL,0       :Clear lo-word.                   21 00 00
        EXX             :Access normal regs.              D9
        RET             :Exit, HLHL' = 0.                 C9
:
:-------------------------------------------------------------
:
```

```
NEGH    OR    A           :Clear Cy for sub no carry.       B7
        EXX               :Access alternate regs.           D9
        EX    DE,HL       :Move HL' to DE',                 EB
        LD    HL,0        :clear HL' and subtract to        21 00 00
        SBC   HL,DE       :negate lo-word.                  ED 52
        EXX               :Access normal regs.              D9
        EX    DE,HL       :Move HL to DE,                   EB
        LD    HL,0        :clear HL AND subtract with       21 00 00
        SBC   HL,DE       :Cy to negate hi-word.            ED 52
        RET               :DEDE' = HLHL'. HLHL' = -HLHL'.   C9
:
:-------------------------------------------------------------
:
EXDH    EXX               :Access alternate regs.           D9
        EX    DE,HL       :Swap DE' with HL'.               EB
        EXX               :Access normal regs.              D9
        EX    DE,HL       :Swap DE with HL                  EB
        RET               :Exit, DEDE' swapped HLHL'.       C9
:
:-------------------------------------------------------------
:
ADDHB   EXX               :Access alternate regs.           D9
        ADD   HL,BC       :Add (no carry) lo-word.          09
        EXX               :Access normal regs.              D9
        ADC   HL,BC       :Add hi-word.                     ED 4A
        RET               :Exit, sum in HLHL'.              C9
:
:-------------------------------------------------------------
:
SUBHB   OR    A           :Clear Cy for sub no carry.       B7
        EXX               :Access alternate regs.           D9
        SBC   HL,BC       :Subtract lo-word.                ED 42
        EXX               :Access normal regs.              D9
        SBC   HL,BC       :Subtract hi-word.                ED 42
        RET               :Exit, difference in HLHL'.       C9
:
:-------------------------------------------------------------
:
LWRLD   EXX               :Access alternate regs.           D9
        RL    E           :Rotate left lo-word, byte        CB 13
        RL    D           :at a time.                       CB 12
        EXX               :Access normal regs.              D9
        RL    E           :Rotate left hi-word, byte        CB 13
        RL    D           :at a time.                       CB 12
        RET               :Exit, DEDE' left rotated.        C9
:
:-------------------------------------------------------------
:
LWRLH   EXX               :Access alternate regs.           D9
        ADC   HL,HL       :Rotate left lo-word.             ED 6A
        EXX               :Access normal regs.              D9
        ADC   HL,HL       :Rotate left lo-word.             ED 6A
        RET               :Exit, HLHL' left rotated.        C9
:
:-------------------------------------------------------------
:
LWSLAX  ADD   IY,IY       :Shift left arithmetic lo-word.   FD 29
        EX    (SP),IX     :Move hi-word, IX, to HL          DD E3
        EX    (SP),HL     :(preserving HL & no stack use)   E3
        ADC   HL,HL       :and rotate left hi-word, getting ED 6A
        EX    (SP),HL     :Cy from lo-word. Restore HL and  E3
        EX    (SP),IX     :hi-word to IX.                   DD E3
        RET               :Exit, IXIY left shifted to Cy.   C9
:=============================================================
```

Z80 32-BIT POWERS

POWX is the final routine from T Sullivan. It makes extensive use of other routines in the suite for transferring the arguments and results between working registers and memory-held variables, and also for the necessary squaring, multiplication, shifting, clearing and exchanging.

DATASHEET 7

```
:=============================================================
:= POWX   - 32-bit signed Nth power.
:=============================================================
:JOB        32-bit (sign bit + 31-bit magnitude) power.
:ACTION     ON result overflow
:           [ Set overflow flag and exit. ]
:           IF power index negative
:           THEN [ set negative index flag and exit. ]
:           ELSE [ Result sign = root sign AND power index lsb.
:                  Set result = 1.
:                  IF power index > 0 THEN
:                  [ FOR power index bits (msb to bit 0)
:                    [ Square result.
:                      IF current power index bit = 1 THEN
:                      [ Result = result * root. ] ] ] ]
:-------------------------------------------------------------
:CPU        Z80
:HARDWARE   Variables in RAM.
:SOFTWARE   XENTRY, XEXIT, LWSLAX, EXDH, SQSUB, CLRH, MULSUB.
:-------------------------------------------------------------
:INPUT      Three variables at (HL), (DE) & (BC).
:OUTPUT     Cy=1: Error, (HL), (DE) & (BC) unchanged.
:                 S=0: magnitude overflow.
:                 S=1: negative power index input.
:           Cy=0: (HL) = (DE)^(BC).
:           S, Z return sign & zero status.
:ERRORS     None.
:REG USE    F BC DE HL
:STACK USE  28
:RAM USE    None.
:LENGTH     75
:CYCLES     Not given.
:-------------------------------------------------------------
:CLASS 2    *discreet     -interruptable    *promable
:*-**-*     *reentrant    -relocatable      *robust
:=============================================================
:
POWX    CALL XENTRY     :Save regs. & get arguments.      CD lo hi
        RLCA            :Move power index sign to sign    07
        RLC  A          :bit of A, root sign to 0,A,      CB 07
        SCF             :set Cy for if error, and exit    37
        JP   M,XEXIT    :S=1 if power is negative.        FA lo hi
:
        EXX             :Access alternate regs.           D9
        AND  C          :Get result sign and return       A1
        RRCA            :it to bit 7,A.                   0F
        INC  L          :Initialise part result to 1.     2C
        PUSH BC         :Move power index lo-word         C5
        POP  IY         :from BC' to IY.                  FD E1
        PUSH DE         :Move root lo-word from DE' to    D5
        POP  BC         :BC' as multiplicand.             C1
        EXX             :Access normal regs.              D9
        PUSH BC         :Move power index hi-word         C5
        POP  IX         :from BC to IX. Now in IXIY.      DD E1
        PUSH DE         :Move root hi-word from DE to     D5
        POP  BC         :BC. BCBC' now multiplicand.      C1
:
        PUSH AF         :Save result sign in normal A.    F5
        LD   A,32       :Set power index bit count.       3E 20
:
PXSIGL  CALL LWSLAX     :Shift index higher until msb     CD lo hi
        JR   C,PXSIGF   :found then jump into power loop. 38 13
        DEC  A          :Repeat until a significant bit   3D
        JR   NZ,PXSIGL  :is found or index = 0. If 0      20 F8
        JR   PXPLE      :then result = 1, so exit.        18 1D
:
:...Power loop. Square part result and if index bit set then
:...multiply by root (actually, result := root * result).
:
PXPLP   CALL EXDH       :Move part result to DEDE'        CD lo hi
        CALL SQSUB      :and square (square in HLHL').    CD lo hi
        JP   M,PXPLE    :Exit if result 31-bit overflow.  FA lo hi
        CALL LWSLAX     :Get next power index bit and     CD lo hi
        JR   NC,PXPLT   :skip if not set.                 30 0C
:
:...Entry point to power loop, 1st squaring unnecessary.
:
PXSIGF  CALL EXDH       :Move result to DEDE' as m'plier. CD lo hi
        CALL CLRH       :Clear product accumulator and    CD lo hi
        CALL MULSUB     :mul by root. (DEDE' is cleared). CD lo hi
        JP   M,PXPLE    :Exit if result 31-bit overflow.  FA lo hi
:
PXPLT   DEC  A          :Repeat for remaining power       3D
        JR   NZ,PXPLP   :index bits.                      20 E3
:
:...Exit ensuring bit 7,H cleared and Cy set only if overflow.
:
PXPLE   POP  AF         :Restore result sign (7,A).       F1
        RLC  H          :Bit 7,H set if overflow, so      CB 04
        SRL  H          :convert to Cy & clear 7,H.       CB 3C
        JMP  XEXIT      :Exit, store result if Cy=0.      C3 lo hi
:=============================================================
```

MC/LCASE FILE SIZE

Russell Greene has sent improvements to his CP/M MAC.PRN comments case conversion routine (*APC*, June).

Although the original code is suitable for small conversion files (less than 16k on Russell's micro but dependent on system disk parameters), it crashes out by underestimating the size of medium-to-large files.

There are three short alterations which add 11 bytes to the program; these are shown in Fig 1. The first change involves a BDOS call to store one data word equal to the file sector size into FCB (file control block) locations 34 and 35, after the file read but before conversion; the second change picks up the two-byte file size for use as a write counter; and the third change is needed for the two-byte decrement and check for zero.

```
;=============================================================
;...MC/LCASE improvements (larger file handling).
;
;...1st change.
;...INSERT at location 01DB (March routine).
;
CONVERT MVI  C,35       ;Compute file sector size and     0E 23
        LXI  D,FCB1     ;store in FCB1+33,34              11 5C 00
        CALL BDOS       ;(FCB locations 34,35)            CD 05 00
```

C P / M

On the move with PIP

This CP/M copying utility can do more than you expect

by Steve McMahon

If any one word of the arcane jargon of CP/M ever enters into popular parlance, it just might be "PIP," as in "Harry, could you pip this package over to Cleveland?" and "Could you please pip me the salt?"

When that day comes, everyone will have forgotten that PIP is an acronym for the CP/M Peripheral Interchange Program; it will be up to some obscure lexicographer to discover that PIP was once a computer program that CP/M users cursed and cajoled in the process of moving data from one place to another.

PIP stands a chance of elevation to household word status because of its tremendous importance to CP/M users. Mastery of the features of PIP is the hurdle that separates proficient CP/M users from novices. Those who have pretty well figured out PIP can set up disks as they please and get data through their printers and modems as necessary. Those who haven't are stuck with using their disks as set up by software distributors or dealers and are often forced to use awkward methods to get data out of their computers.

Although there are some good commercial and public domain programs that imitate PIP's features—some with excellent menus to ease use—they don't really replace PIP. PIP is compact, versatile, extremely reliable if used properly, and it costs you nothing extra since it's included with CP/M.

This article is meant to be an introduction to using PIP's various facilities, to get novices over that first hurdle (which isn't really a very high one at all). It is hoped that more experienced CP/M users will also find some useful information here.

PIP as pipeline

If you don't want to think of PIP as the "Peripheral Interchange Program" (and who does?), you might try thinking of it as a pipeline; with it, you can "pipe" data such as text or programs for one place to another. Those places can be files stored on disks, printed copy on your printer, or even the telephone line connected to your modem. (Unlike any real, physical pipeline, though, PIP does its moving without taking anything away from the source of the flow—it just copies an image of electric pulses from one place and recreates it in another.)

When you invoke PIP, you need to tell the program fairly precisely how to build the pipeline by telling it the names of the places you want to connect. PIP will understand you if your instructions take the form:

destination place = place or origin

The PIP pipeline flows from right to left, with the equal sign representing the pipe connecting destination and origin. Most of the rest of this article will be concerned with explaining how to describe places of destination and origin in a way that PIP will understand.

You can give PIP instructions in two different ways. Instructions may be put on the same CP/M command line in which you invoke PIP, for example:

A>pip destination = origin<RETURN>

Or, you may just run PIP like any other simple program, typing its name at the CP/M prompt and following it with a return:

A>pip<RETURN>

If you use the latter method, PIP will issue its own * prompt to which you reply with your specifications for the pipeline:

*destination = origin<RETURN>

If you have more than one command to issue to PIP, this latter form is generally the better. When PIP finishes up one set of orders under this form, it re-issues its * prompt—you don't have to wait through a warm boot. When finished issuing orders to PIP, just type a return in response to its prompt and PIP will return you to the CP/M prompt with a warm boot. Note that PIP generally translates lower-case to upper-case letters; you don't have to type PIP commands all in capitals.

Copying Files With PIP

By far the most frequent use of PIP is to copy files either for backup purposes or to create a convenient working collection of files on a disk. These files might be text files created with your text editor or spreadsheet program, command files with the suffix **.COM**, or files of data that are not command files, but also aren't text files (more on the latter later—they ought to be handled in a special way). PIP can copy these files between disk drives or on the same disk. When copying files, the "place" names you need to specify to PIP are simply the names of the files involved.

Say you wish to copy a file **LETTER1.MSS** to another place on the same diskette. This might be done in preparation for editing **LETTER1.MSS** without endangering your original file. Since this copy is going to go on the same disk, you will need to think of a new name for the duplicate (since you can't have two files with the same name on the same disk). As with the old, the new name will have to be a valid CP/M file-name consisting of up to eight characters followed by an optional period and three letter suffix. Also, the new name should be one that makes sense to you, and probably one that suggests the nature of its relation to the original. Your choice might be **LETTER2.MSS**, in which case you would issue the command:

A>pip letter2.mss = letter1.mss
<RETURN>

When PIP finishes the copy, the computer will warm boot and, if you ask for a directory, the new file should show up.

Actually, you should never issue PIP a copy command that looks like the one above. Instead, you should add a **[v]** to the end of the line so that PIP will know you wish to be sure the transfer was made correctly. When PIP finds a **v** parameter inside square brackets when copying files, it will read the copy it makes over to *verify* no copying error has been made. So, if you want to make sure your data gets copied accurately, your command would appear:

A>pip letter2.mss = letter1.mss[v]
<RETURN>

Verified copies take longer, but, if your files are of any value to you, all that time and effort will be repaid the first time PIP detects a copying error. The confidence this verification can give you is one of the reasons PIP is superior to many of its commerical and public domain imitators, some of which are faster but do not verify.

PIP can move files between disk drives just as easily. All you have to do is specify the drive name (**A:** or **B:**) whenever the destination or origin is on the drive other than the one you are currently logged onto (It never hurts to specify a drive name even if the destination or origin is on the same disk.). If you are logged onto drive A:, which is where **LETTER1.MSS** is, and wish to create a copy **LETTER2.MSS** on the disk in your B: drive, your command to PIP would appear:

A>pip b:letter2.mss = letter1.mss[v]
<RETURN>

The command to move files between disks can often be even simpler. Since the copy will be on another disk, you can give it the same name as the original without violating the rule against having two similarly-named files on the same disk. If you wish to create a copy of a file and put that copy on the disk in the other drive, you may instruct PIP to give it the same name as the original file. PIP understands some shorthand for this operation; if the destination given is a drive with no filename, PIP assumes you want the new file to have the same name as the original. If, for example, you were logged onto A: and wanted to put a copy of a file on A: named **LETTER1.MSS** onto B:, you would enter:

A>pip b: = letter1.mss[v]<RETURN>

Given a command like this, PIP will look for **LETTER1.MSS** on the disk in the A: drive (since this is the current drive as indicated in the A> prompt) and create a new copy of it on the disk in the B: drive. The same thing will occur as if the command had been written:

A>pip b:letter1.mss = a:letter1.mss[v]
<RETURN>

Using this form, one of the first commands you might wish to use is:

A>pip b: = a:pip.com[v]<RETURN>

to copy PIP from your system disk in the A: drive to a new working disk in B: (PIP is one command you'll probably want to have on nearly all your working diskettes.).

When making a transfer from one disk to another like this, PIP will allow you to refer to a whole set of files with one **ambiguous** name. An ambiguous name is one that refers to several real file names. An example is the ambiguous name *.*, which refers to every file on a disk, or ***.COM**, which stands for every file on a disk with a **.COM** suffix. You would back up every file with a **.MSS** suffix from the disk in A: to the disk in the B: drive by entering the command:

A>pip b: = a:*.mss[v]<RETURN>

PIP will list all the files it acutally transfers in response to this ambiguous command. Another way to write an ambiguous file name is to use a **?** symbol to stand for every character in the filename for which PIP is to accept any value. Thus, if given the ambiguous name **LETTER?.MSS**, PIP would copy **LETTER1.MSS, LETTER2.MSS**, and so on (but not **LETTER1A.MSS**) to a new disk.

What does PIP do if there is already a file on the disk you are copying to by the name of the one you wish to copy? Unless you have caused the file on the

(Continued)

destination disk to be write-protected, PIP will erase it to avoid having two files with the same name on the same disk (If the file is write-protected, PIP will first ask you if you wish to delete it.). This is often exactly what you want done when backing up a disk. If it's not, you'll need to do something (like re-naming) to protect the similarly-named file on the destination disk. (*Note: this refers to write-protection using* software, *not adhesive write-protect tabs that cover the notches on your disks. PIP may be powerful, but* no *program can write to a disk with a covered notch.*)

File-copying errors

The most common error you will see in using PIP to copy files will be a disk write error caused by there being insufficient space on the disk on which you wish to put the copy. When an error like this occurs, PIP will report a **WRITE ERROR** and you will find in the directory of your destination disk a funny file with a **.$$$** suffix. Unfortunately, **.$$$** is not a Digital Research jackpot and it doesn't mean the next copy of CP/M is on the house. Instead, it marks the place on the disk where CP/M unsuccessfully tried to create the new copy. You should go ahead and erase the **.$$$** file and look around for some way to make more room on the disk by erasing other unnecessary files. **STAT *.*** or a public domain extended directory program can report file sizes to help you find the space you need.

If this occurs, it may be that the file you will need to delete will have the same name as the file you are trying to copy. PIP does not immediately delete similarly-named files on the destination disk. Instead, it creates a working file with the **.$$$** suffix. When the transfer is complete, the similarly-named file is deleted and the **.$$$** file is renamed. This means you need to have free space on a disk equal to the size of the file you wish to copy—even if you intend to overwrite a similarly-named file already on the destination disk. If that space isn't available, you will need to delete the similarly-named file on the destination disk *before* attempting the copy.

Another type of error you may encounter is a **BDOS ERROR** in which the error message indicates the destination disk is read-only. This might mean that you have a write-protect tab on the destination disk, but it probably means you have changed disks without resetting the disk system. CP/M needs to know when you have changed disks or even opened a drive door. Tell it you have done so by typing a CONTROL-C immediately after the CP/M prompt.

A special case: object files

To copy a file, PIP has to have some way of knowing when to quit; it must be able to detect the end of a file. This is no problem for text files created by word processing or spreadsheet programs. These programs mark the end of a file with a special character, a CONTROL-Z, and are careful to avoid putting that character in any place other than the end of the file. While this end-of-file convention is convenient for text files, it doesn't work at all for what are called **object** or **binary** files.

Examples of object files include CP/M command files, indexes kept by data base programs, numeric data files created by some computer languages, and program overlays. Data in this type of file are kept in a form that is meant to be immediately useful to the computer as instructions or operating data, and the equivalent of a CONTROL-Z may appear at many places other than the end of the file.

Fortunately, PIP has a way of finding the end of such files, though you may have to tell PIP when to use this alternate method. CP/M keeps a record of how many sectors (a chunk of disk memory 128 bytes long) a file takes up. If PIP knows it can't look for a CONTROL-Z as an end of file marker, it just transfers all the sectors having anything to do with the file.

PIP automatically uses this technique with any file with a **.COM** suffix. The problem is that many object files do not have a **.COM** suffix. When copying these files, you must inform PIP that the file is an object file by putting a **[o]** after the origin file's name. So, if you were copying an object file named **BASIC.LIB**, your command to PIP might appear:

```
A>pip b:=a:basic.lib[o]<RETURN>
```

You would probably want to verify that a good copy was made by adding a **v** for verify to the parameter list inside the square brackets. In this case, the command would be:

```
A>pip b:=a:basic.lib[ov]<RETURN>
```

Object files are commonly marked by suffixes like **.INT, .OVR, .SWP, .REL, .CRL, .DTA, .INX**, or **.LIB**, but suffixes won't really help you be sure which files are object files and which files are text files. The best rule to follow is: If you didn't create it with your word processing program, treat it as an object file.

What happens if you copy an object file without including the **o** parameter? Possibly nothing bad. But, it is also possible that PIP will quit copying early and your copy will be useless.

Using PIP to add files together

Although PIP can create a pipeline to only one destination, it can go to several points of origin to get the data for that destination. This means you can use PIP to append one file to another, or even several files together. The basic form of the command is:

```
destination=origin1,origin2,origin3 . . .
```

You should use commas to separate the places of origin you describe to PIP in such a command. Don't put any spaces between the names.

If you had a pair of files, **DEMAND.MSS** and **THREAT.MSS**, which you wished to combine into a single file, **EXTORT.MSS**, you would issue the command (to the computer):

```
A>pip extort.mss=demand.mss,
   threat.mss[v]<RETURN>
```

Other places PIP knows

PIP's view of the world is not limited to files and disks. PIP can also get data from places of origin like your keyboard and serial port and move data to places like your video screen, serial port and parallel port.

(Continued)

PIP knows your printer as **LST:** (list device) and your keyboard and video screen as **CON:** (console device). You may put these names in your PIP commands in a variety of ways that PIP will understand. The most commonly used is to send a file to the printer by commanding:

A>pip lst:=threat.mss<RETURN>

if **THREAT.MSS** is the file you wish to print. Notice that this command is really little different from the ones given above: the PIP pipeline flows from the right side of the equal sign (in this case the file **THREAT.MSS**) to the left (the printer). Also notice that there is no **[v]** in this command: PIP can't re-read your printer to make sure everything got there okay. You'll have to do that yourself.

If you had wanted to print more than one file, one after another, the command might have been:

A>pip lst:=greeting.mss,demand.mss,
threat.mss,closing.mss<RETURN>

LST: could be either your parallel or serial port, depending on how your CP/M is installed. If you want, though, you can specify exactly which one of these you want PIP to send the file to. You might want to do this if you have two printers hooked to your computer or if you have a modem connected to your serial port and want to send a file to it. PIP knows the serial port by the name **TTY:** (for teletype device) and your parallel port by the name **LPT:** (for line-printer).

By combining place names in unusual ways, PIP can be made to do some amazing things. For example, you can make PIP into a crude word-processor by issuing the command:

A>pip file.doc=con:<RETURN>

After PIP digests this command, it will accept whatever you type as the contents of the file **FILE.DOC**. If you wish to type more than one line, use both the return and line feed keys to separate lines. Type a CONTROL-Z to end the file and get back to the CP/M prompt. A more useful PIP command might be:

A>pip lst:=con:<RETURN>

This command turns your computer and printer into an expensive (and not very good) electric typewriter by building a pipeline between your keyboard and printer. You can type more than one line and end the typing in the same way as noted immediately above. (If your printer has a buffer in it, it won't type immediately when you do. Instead, it may type only after you type returns, when the buffer is full, or when you quit typing by entering a CONTROL-Z.)

Another useful "place" PIP knows about is a fictional one. When PIP is given the device name **PRN:** (for print) as a destination, it sends the output to the printer with line numbers before every line. PIP also will issue your printer a form feed every sixty lines and will print eight spaces where ever your file contains a tab character. This feature is awfully useful for program printouts and any text draft that requires line number references. You would print the source code of a BASIC program **BRIBE.BAS** this way with the command:

A>pip prn:=bribe.bas<RETURN>

PIP's other parameters

The **PRN:** device mentioned above gives you an effect that instead could have been achieved using PIP parameters. Parameter **p** will cause form feeds to be added every sixty lines (if you want a form feed every forty lines, use the parameter **p40**), **t8** will cause eight spaces to be inserted in place of tab characters (**t5** would call for five spaces instead), and **n** would cause line numbering. So, if you wished to print out **BRIBE.BAS** with numbered lines, five space tabs, and a form feed every forty lines, you would type:

A>pip lst:=bribe.bas[nt5p40]
<RETURN>

The tab parameter **t** can be useful not just for changing the appearance of tabs, but to successfully print a file containing tabs on a printer that doesn't accept the tab character properly.

The **d** parameter provides another way to get around printer limitations. You may use the **d** parameter followed by a number to cut off lines sent to your printer at a certain number of columns. For example, if your printer goes haywire if you try to print lines longer than eighty characters and you want to print a program file that has a few unimportant comments that would run longer, you would enter:

A>pip lst:=bribe.bas[d80]
<RETURN>

Two other PIP parameters deserve mention here: the **s** and **q** (for start and quit) parameters can be used to pick out some particular part of a file to copy or print. Other parts of the file are then left behind. You may choose the sections to be printed by looking for unique strings of characters that mark off the section you wish to copy to another file or to the printer.

When you issue the command to start or stop printing at a particular string, enter the string after the parameter letter and mark the end of it by typing a CONTROL-Z, which will show up in the command.

If you had **THREAT.MSS** which contained the text, **Do this or we will be forced to erase all your floppy disks**, and you wished to print just the words **we will be forced**, you would enter the command:

A>pip<RETURN>
*lst:=threat.mss[swe^Z qforced^Z]
<RETURN>

A command like this should be entered at an asterisk prompt. If you enter the command at the CP/M command line, CP/M will convert your start and quit strings into capital letters.

Where to go for more

This information should be sufficient for most of what you're likely to want to do with PIP. PIP, though, knows of several more devices and accepts more parameters than have been mentioned here. Most of these are useful for more specialized purposes, for example transferring hex-format files. If you wish to know more, you should consult Digital Research's *An Introduction to CP/M Features and Facilities*, which was the primary technical source used preparing this article. □

B?9 specifies one sector to be read

Lines 210 to 320 print each file and its directory, and allow the user to alter the directory via the subroutine at 370. If the directory number is greater than &7E, the file is locked. To obtain the directory, subtract ?BO.

M Morgan

```
   10 REM ** DIRectory changer **
   20 REM ** by Meirion Morgan **
   30 MODE 3
   40 PRINT 'STRING$(79,"*")
   50 PRINT '"   DIRECTORY CHANGER FOR BBC MICRO (
 USED ON 8271 ) - BY MEIRION MORGAN "
   60 PRINT 'STRING$(79,"*")
   70 INPUT TAB(0,8)"Which drive ",drive
   80 IF drive<0 OR drive>3 OR INT(drive)<>drive G
OTO 70
   90
  100 DIM B 12:DIM H 255
  110 A%=&7F:X%=B MOD 256:Y%=B DIV 256
  120 B?0=drive
  130 B!1=H
  140 B?5=3
  150 B?6=&53
  160 B?7=0
  170 B?8=0
  180 B?9=&21
  190 CALL&FFF1
  200
  210 CLS:yes=0:B?6=&4B:name$=""
  220 FOR files=14 TO 254 STEP 8:IF (H?files)>&7E
OR (H?files)<&20 GOTO 330
  230 FOR peek=files-6 TO files
  240 name$=name$+CHR$(H?peek)
  250 NEXT peek:dir=H?(files+1)
  260 IF dir>&7E dir=dir-&80:yes=1 ELSE yes=0
  270 dir$=CHR$(dir)
  280 PRINT dir$;" . ";name$;
  290 IF yes=1 PRINT ;" Locked . Change ? "; ELSE
PRINT ;" Change ? ";
  300 c$=GET$:IF INSTR("YyNn",c$)=0 GOTO 300 ELSE
PRINT ;c$;
  310 IF INSTR("Yy",c$) GOSUB 370 ELSE PRINT
  320 name$="":NEXT files
  330 PRINT ''"*********** END OF PROGRAM ********
**"
  340 CALL&FFF1
  350 REPEAT UNTIL FALSE
  360
  370 PRINT ;" To directory ? ";:d$=GET$
  380 PRINT ;d$;:byte=ASC(d$)
  390 PRINT ;" Lock ? ";:lock$=GET$
  400 IF INSTR("YyNn",lock$)=0 GOTO 390 ELSE PRINT
 ;lock$
  410 IF INSTR("Yy",lock$) byte=byte+&80
  420 H?(files+1)=byte
  430 RETURN
 >
```

TI SOUNDS

The noises −4 and −8 vary the tone of the third tone specified (pg II−85 Users Reference Guide) in a CALL SOUND statement.

I have noticed that by using −4 and −8 any noise can be created. Where −4 can create noises −1, −2 & −3 and where −8 can create noises −5, −6 & −7.

The following program demonstrates this by using 129 different noises created by −4 to form the sound of an aeroplane taking off.

```
100 FOR T = 110 TO
    4000 STEP 30 :: CALL
    SOUND (−100,110,30,
    110,30,T,30,−4,0) ::
    NEXT T
110 CALL SOUND
    (−100,110,30,110,30,T,
    30,−4,0) :: GOTO 110
```

Hence 89246 noises (not tones) are available on the TI, and you can hear all of them, none are out of the range of hearing. 44623 of those noises are generated by −4 and another 44623 noises are generated by −8.

The following program will let you hear every noise the TI is capable of: (if you have the willpower to listen to them all).

```
100 FOR T = 110 TO
    44733 :: CALL SOUND
    (−100, 110,30,110,30,
    T,30,−4,0) ::
    NEXT T
110 FOR T = 110 TO
    44733 :: CALL SOUND
    (−100,110,30,110,30,T,
    30,−8,0) :: NEXT T
```

The first couple of sounds you won't be able to hear because −100 duration doesn't give enough time for it to be activated.

P Bruce

. . . /RAM . . .

If you have a copy of Prodos and an Apple IIc or Extended IIe, then you have a RAM disk.

The RAM disk uses up the auxiliary memory. This gives you a 64k (128 blocks), fast, silent, and convenient RAM disk.

If you know all about prefixes then you should have no trouble using it. You boot up a disk that is Prodos formatted and when you want to access the RAM disk, you do a PREFIX /RAM. Now if you do a CAT or CATALOG, you will be given the contents of your RAM disk. /RAM operates just like a normal disk and performs all the normal Prodos functions. To get out of /RAM you do a PREFIX /.

You might be able to use /RAM with a 64k machine, but I am not sure of this.

S De Silva

VZ EDITOR/ ASSEMBLER TIPS

To enter hi-res mode (mode (1)) in assembler set bit 3 of address 6800 H(26624) to 1. For example:

```
LD A,(6800H) ; Load A with
  content of 6800H
OR 8 ; Set Bit 3 of A to 1
LD (6800H),A ; Load new
  information back
LD (783BH),A ; into 6800H
  and 783BH
```

If you want to change the background colour to buff (normally it's green), instead of [OR 8], as above, change that to OR 24 (setting bit 4 to 1).

(783BH) is the copy of (6800H). It is important to load A into (783BH) if you want to use the sound driver routine in ROM, because the SDR does a Read (783BH) to see what mode you are in, and loads that into (6800H).

To Call the sound driver routine

```
LD HL, Frequency
LD BC, Duration
Call 345CH
```

Before returning back to the Editor/Assembler use the program below to clear bit 3 of (783BH). If you don't, the screen will change to mode (1) (hi-res) when you use [Tape Save] in the Editor/Assembler.

```
LD A,(783BH)
AND 247
LD (783BH),A
```

T Lam

CLEAN HEADS

Normally when cleaning the 1541 drive the user is forced to keep the drive read/write heads in constant movement by either loading/saving while the cleaning disk is inside the drive, consequently the user must keep typing load/save to move the heads — in order to clean them thoroughly.

This short program eliminates this repeated load/saving and makes drive cleaning a much easier task. It spins the disk while cleaning the drive and will

the commands with RANDOMIZE USR 65000. To use the commands in your own programs, make the first line of the program RANDOMIZE USR 65000. The command allows up to 100 nested loops, and if it comes across an ★UNTIL without a corresponding ★REPEAT, it will give the NEXT WITHOUT FOR error message.

D Hales

```
  1 REM
  2 REM   REPEAT / UNTIL
  3 REM   by David Hales
  4 REM
 10 CLEAR 64999
 20 PRINT AT 10,10; FLASH 1;"PLEASE WAIT"
100 FOR A=65000 TO 65174
110 READ B
120 POKE A,B
130 NEXT A
140 CLS
150 PRINT "   THE CODE IS NOW IN MEMORY"
160 PRINT
170 PRINT "BEFORE RUNNING THE DEMONSTRATIONPROGRAM SAVE THE CODE FIRST."
180 PRINT
190 PRINT "SAVE TO TAPE WITH:"
195 PRINT
200 PRINT "SAVE ""R/U""CODE 65000,174"
210 PRINT
220 PRINT "OR MICRODRIVE 1 WITH:"
230 PRINT
240 PRINT "SAVE *""M"";1;""R/U""CODE 65000,174"
250 STOP
900 REM
910 REM MC DATA
920 REM
1000 DATA 33,183,92,54,252,35,54,253,33,154
1010 DATA 254,34,151,254,62,0,50,153,254,201
1020 DATA 215,24,0,254,42,40,3,195,240,1
1030 DATA 215,32,0,254,82,40,7,254,85,40
1040 DATA 54,195,240,1,6,6,215,32,0,16
1050 DATA 251,205,183,5,58,153,254,254,100,32
1060 DATA 5,253,54,0,3,239,60,50,153,254
1070 DATA 42,151,254,58,69,92,119,35,58,70
1080 DATA 92,119,35,58,71,92,60,119,35,34
1090 DATA 151,254,195,193,5,6,5,215,32,0
1100 DATA 16,251,215,130,28,205,183,5,58,153
1110 DATA 254,254,0,32,5,253,54,0,0,239
1120 DATA 215,148,30,254,1,40,26,42,151,254
1130 DATA 229,43,126,50,68,92,43,126,50,67
1140 DATA 92,43,126,50,66,92,225,34,151,254
1150 DATA 195,193,5,42,151,254,43,43,43,34
1160 DATA 151,254,58,153,254,61,50,153,254,195
1170 DATA 193,5,0,0,0
```

```
  1 REM
  2 REM   REPEAT UNTIL DEMO
  3 REM PROGRAM TO FIND FIRST
  4 REM  OCCURENCE OF A BYTE
  5 REM  IN THE SPECTRUM ROM
  6 REM (NOT ONE GO TO HERE!)
  7 REM
  8 RANDOMIZE USR 65000
 80 LET INFINITY=0
 90 *REPEAT
 95 *REPEAT:*UNTIL INKEY$=""
100 PRINT AT 10,3;"REPEAT UNTIL DEMONSTRATION";AT 12,5;"PRESS ANY KEY TO START"
110 *REPEAT:*UNTIL INKEY$<>""
120 CLS
140 *REPEAT
150 INPUT "TYPE A NUMBER BETWEEN 0 & 255   ";N
160 *UNTIL N>=0 AND N<=255
165 PRINT AT 5,5;"SEARCHING ROM FOR ";N;AT 7,7;"PRESS ""X"" TO EXIT"
170 LET ADDRESS=-1
180 *REPEAT
190 LET ADDRESS=ADDRESS+1
195 PRINT AT 10,5;"SEARCHING ADDRESS ";ADDRESS
200 *UNTIL PEEK ADDRESS=N OR INKEY$="X"
205 CLS
210 IF PEEK ADDRESS=N THEN  PRINT AT 5,6;"FOUND AT ADDRESS ";ADDRESS
220 *UNTIL INFINITY
```

BBC EASY DISK LOADING

This utility speeds up the loading of Basic programs from disk, and should be used when !BOOT is unsuitable. The utility places a piece of code at &900 which inserts a VDU 12 control code, followed by a CH. "HHHHHHH" and a return control code, into the keyboard buffer. The program can then be loaded as a ★ command, as the code strips off the filename after the ★ and automatically chains it.

To use the program, first check that the required Basic program is on disk, then chain the Easy Load program (which you should have previously typed in and saved). You will be asked to name the Basic program Easy Loader; this name should be entered exactly as it is given in the disk catalogue. You will then be asked for the name of the machine code loader (the name to be placed after the asterisk) which should be as short as possible. You must ensure that this name is not the same as any of the OS commands, or any other Easy Load programs.

H Rees

```
 10 REM   +++++++ EASY-LOAD +++++++
 20 REM   A Disc Utility for easier
 30 REM   loading of BASIC programs.
 40
 50 REM      Written By Huw Rees.
 60 REM         13th Aug 1985
 70 MODE 7
 80
 90
100 P%=&900
110 REM   Start of assembler.
120 [
130 OPT 2
140 PHA
150 TYA:PHA
160 TXA:PHA
170 LDA #&8A
180 LDX #&0
190 LDY #&C
200 JSR &FFF4
210 LDY #&43
220 JSR &FFF4
230 LDY #&48
240 JSR &FFF4
250 LDY #&2E
260 JSR &FFF4
270 LDY #&22
280 JSR &FFF4
```

MS-DOS COUNTRIES

MS-DOS versions 2.0 or higher use a file called CONFIG.SYS during booting to set up various operating system parameters, one of which is COUNTRY=44. This refers to the operating system's ability to express the date in different formats, depending on the country of use. (We use the same as the UK).

These are coded into CONFIG.SYS using the international trunk dialling codes, as shown in Fig 1.

These codes only configure the operating system and do not affect programs such as dBasell, so you won't be able to configure these to the local date format.

RJ Akers

CONFIG.SYS entry	date format	country
COUNTRY = 01	month:day:year	USA
COUNTRY = 44	day:month:year	UK
COUNTRY = 81	year:month:day	Japan

Fig 1

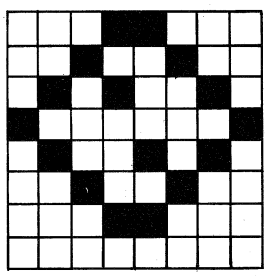
VZ USER GRAPHICS

```
1000 A=44800:B=65536
1010 READ C:IF C=1THEN
     1070ELSEPOKEA-B,C
     :A=A+1:GOTO1010
1020 DATA245,197,213,
     229,33,0,0,17,0,0
1030 DATA14,8,26,119,
     35,19,26,119,6,31
1040 DATA35,5,120,254,
     0,194,20,175,19,13
1050 DATA121,254,0,
     194,12,175,241,193
1060 DATA209,225,
     201,-1
1070 POKE30862,0:POKE
     30863,175:RETURN
```

This routine will provide any VZ programmers with the ability of creating their own definable high resolution characters in 8x8 pixels.

For example

00000011	11000000
00001100	00110000
00110011	00001100
11000000	00000011
00110000	11001100
00001100	00110000
00000011	11000000
00000000	00000000



Refer to the technical manual for more details on high resolution graphics.

To activate this routine, you

simply poke the starting address of the code for your user definable graphic into the memory location 44808/9 and the screen position of your user definable graphic into the memory location 44805/6.

Sample Program

```
 5 GOSUB 1000
10 FOR T=45000 TO
   45015
20 READ S:POKE
   T-65536,S
30 NEXT T
40 POKE 44808-65536,
   200:POKE 44809-
   65536,175
50 POKE 44805-65536,
   0:POKE 44806-
   65536,112
60 MODE(1):X=USR(0)
70 GOTO 70
1080 DATA 3,192,12,48,
   51,12,192,3,48
1090 DATA 204,12,48,3,
   192,0,0
```

APPLE SWITCH

The Apple IIe has a colour switch which allows a sharper image on monochrome monitors. Unfortunately the switch is inconveniently located within the computer's case. This is a hassle if both a monochrome monitor and a colour TV (with a video modulator) is attached, as you have to frequently switch to colour for games and back to monochrome for word processing using graphic-text, for instance.

To improve accessibility, I connected a switch in parallel with the existing one and mounted the new switch in a more convenient location at the back of the computer where there are already various holes punched in the panel for cables etc. To do this the following parts are needed:

1xSPDT switch (push on/push off alternate action type is ideal);
2xE-Z clips or similar;
2x0.5 metres of insulated hookup wire.

Solder the switch to one end of each wire and the clips to the other ends. Switch off the power, open the top lid on the computer's case and attach one clip to the right end of resistor R72 on the motherboard. (Left and right with respect to the keyboard being nearest to you and keyboard facing upwards). This resistor is found near coordinate B-14. (The motherboard is marked with letters along one edge, and numbers along the other). Attach the other clip to the left end of resistor R76 located at coordinate B-13. Mount the switch in a convenient location like the back panel. Set the existing colour switch located at B-15 to 'color' (ie, away from the keyboard). Finally close the lid and switch on the power.

K Lau

EXTRA VIC-20 VOICE

To gain an extra sound channel to the VIC-20 remove the main circuit board and connect a piece of wire between the CB2 line on the user port to the audio out line in the audio/video port and type the following:

POKE 37147,16
POKE 37144,N (where N is between 0 and 255 representing the note to be played)
POKE 37146,W (where W is 15, 51 or 85 for square waves)
WARNING: DO NOT USE THE DATASETTE IN THIS MODE.
POKE 37147,0 will restore the VIC to normal operations.

This extra voice may be used in conjunction with the other voices but there is no volume control for the extra voice.

P Austen

byte machine code routine is needed. A good place to locate this is within the first user-defined graphic position, which always starts one byte above HIMEM so can be easily located.

The technique used is to write in the routine using the SYMBOL command, ensuring that the number used is the same for the SYMBOL AFTER command which sets up the table of characters. The first byte of the SYMBOL character is used to store the character which is read in from the screen. Some juggling is needed to get the address of this byte into the code; lines 30 and 40 of the program do this.

The SYMBOL 200 used here is quite arbitrary. If you are not defining any characters for your program, you could make it SYMBOL AFTER 255 to economise on memory.

To use the routine, LOCATE to the position you want to test, CALL HIMEM+2, then PRINT PEEK (HIMEM+1) which yields the character number found. PRINT CHR$(PEEK (HIMEM+1)) will print out the actual character.

J Durst

```
10 SYMBOL AFTER 200
20 SYMBOL 200,0,&CD,&60,&BB,&32,0,0,&C9
30 ×1=INT ((HIMEM+1)/256):×2=HIMEM+1−256*×1
40 POKE HIMEM+6,×2:POKE HIMEM+7,×1
```

MACHINE LANGUAGE CALLS

This simple VZ200/300 routine can save programmers from using lots of POKE commands in a Basic program when calling a lot of machine code subroutines.

Conventional method:
To call the address 13392 & 13404

```
10 POKE 30862,80:POKE
   30863,52
20 x=USR(0)
30 POKE 30862,92:POKE
   30863,52
40 X=USR(0)
```

New method:

```
10 X=USR(13392):
   X=USR(13404)
```

Main program:

```
0 POKE 52992-65536,58:
  POKE 52993-65536,33
1 POKE 52994-65536,
  121:POKE 52995-
  65536,50
2 POKE 52996-65536,13:
  POKE 52997-65536,207
3 POKE 52998-65536,58:
  POKE 52999-65536,34
4 POKE 53000-65536,121
  :POKE 53001-65536,50
5 POKE 53002-65536,14:
  POKE 53003-65536,207
6 POKE 53004-65536,195
  :POKE 30862,0
7 POKE 30863,207
```

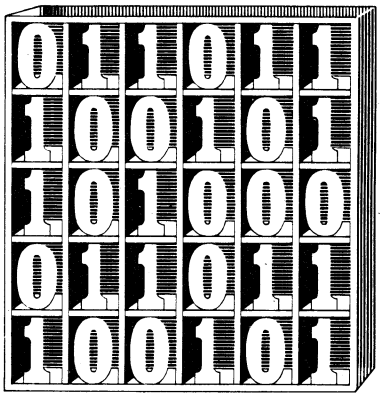
APPLE HI-RES SCROLLER

This routine scrolls the hi-res screen in any one of four directions. To use it, type in the listing (assuming you know how to use the system monitor, if not then refer to chapter 3 of the Apple II Reference Manual) and save it by typing BSAVE SCROLL, A$300,L$8B. There are four different calls for scrolling:

CALL 768 — scroll up
CALL 777 — scroll down
CALL 838 — scroll left
CALL 873 — scroll right

To scroll page 1 do a POKE 230,32 and to scroll page 2, POKE 230,64. Note that when scrolling left and right, the routine scrolls

SUBSET



David Barrow presents more documented machine code routines and useful information for the assembly language programmer. If you have a good routine, an improvement or conversion of one already printed, or just a helpful programming hint, then send it in and share it with other programmers. Subroutines for any of the popular processors and computers are welcome but please include full documentation. All published code will be paid for. Send your contributions to Subset, APC, 2nd Floor, 215 Clarence Street, Sydney 2000.

FASTER DIVISION?

It isn't often that I receive letters beginning: 'I've done it! I expect vestal virgins to fan me with ostrich feathers, a gross or two of PhDs, lecture tours, the adoration of the masses . . .', and ending: 'I await your adulation.' I am happy to say that most contributors to SubSet are rather more circumspect that Charles Wehner about the historical significance of their work.

But then, most contributors cannot lay claim to having improved the standard method of binary long division.

The standard method subtracts divisor from remainder at each bit place — but only if that will leave the remainder positive. Normally, this selectivity is achieved either by a comparison before subtraction or by subsequent addition if the subtraction result is negative.

The former method involves a comparison for every bit and a subtraction for each set (1) bit of the quotient. The latter method involves a subtraction for every bit and an addition for each reset (0) bit. Either way, the number of arithmetic operations is, on average, 50 per cent greater than the quotient bit length.

Charles's improvement eliminates both pre-comparison and post-addition by implementing two cross-connected complementary division loops. One loop subtracts the divisor from a positive remainder and sets the result bit; the other adds the divisor to a negative remainder and resets the result. A sign change in the remainder after subtraction/addition forces control to be passed to the complementary loop, ensuring that the correct result bit is shifted into the quotient.

DIV8 (Fig 1) is the 8-bit division routine that Charles has added to his Jupiter Ace's Forth. RST 24 passes a parameter to DE from the Forth stack, and RST 18 passes DE back to Forth. Timing is given in clock cycles, and the loop pass timing lists possible single-bit times but totals only the average.

```
:
:                                                  cycles   code
:
DIV8    RST   24        :Get 8-bit divisor in      11+      DF
        PUSH  DE        :DE, stack it, then get    11       D5
        RST   24        :8-bit dividend in DE.     11+      DF
        XOR   A         :Clear quotient and         4       AF
        PUSH  AF        :clear stack byte.         11       F5
        INC   SP        :Adjust SP for pop of       6       33
        POP   HL        :divisor to H & 0 to L.    10       E1
        INC   SP        :Tidy up stack.             6       33
        EX    DE,HL     :Dividend HL, divisor DE.   4       EB
        LD    B,9       :Count 8-bits + 1.          7       06 09
:
DSUBLP  SBC   HL,DE     :+ve remainder - divisor. 15  -     ED 52
        CCF             :Set correct result bit.   4  -     3F
        JR    NC,DCLRR  :Jump if -ve remainder.    7 12     30 0C
DSETR   ADC   A,A       :Shift in result bit.      4  -     8F
        SRL   D         :Move divisor down to      8  -     CB 3A
        RR    E         :next bit place.           8  -     CB 1B
        DJNZ  DSUBLP    :Repeat until done        (13/8)    10 F4
        JR    EXIT8     :then exit DIV8.          12  -     18 0A
```

```
:
DADDLP  ADD   HL,DE     :-ve remainder + divisor.  - 11    19
        JR    C,DSETR   :Jump if +ve remainder.    12  7    38 F4
DCLRR   ADC   A,A       :Clear result bit.         -  4    8F
        SRL   D         :Move divisor down to      -  8    CB 3A
        RR    E         :next bit place.           -  8    CB 1B
        DJNZ  DADDLP    :Repeat until done.       (13/8)    10 F6
:
EXIT8   LD    E,A       :Move quotient to E,       4        5F
        LD    D,0       :clear D, and pass         7        16 00
        RST   16        :quotient DE back.        11+       D7
        JP    (IY)      :Exit to FORTH.            8        FD E9
:
:T.   start & end:                                111 + Restarts.
:   loop pass   1: ([59]*1)                        59
:   loop pass   2: ([59,64:61.5]*1)                61.5
:   loop pass 3-8: ([59,64,56,51:57.5]*6)         345
:   loop pass   9: ([71,64,68,51:63.5]*1-5)        58.5
:                                                 ----
:                  FIG 1                          635 + Restarts.
:
```

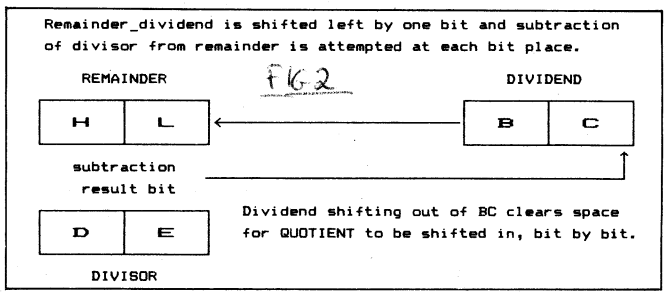
OPTIMISED DIVISION

DIV8 is machine-specific but is easily adapted to general use. However, the basic algorithm it employs is not the most efficient, and 8-bit routines have a somewhat limited usefulness.

I decided to put Charles's idea to the test on a 16-bit long division after attempting what other timing improvements I could. The register assignments and basic Z80 concept of this method are shown in Fig 2. A straightforward coding, DIV16A, utilising post-addition in an IF-THEN-ELSE construct, is given in Fig 3.

A number of improvements immediately suggested themselves. The most obvious is to exchange the assignments of registers A and B to make use of their unique functions as accumulator and loop count register. And, as selection of add-back or increment is based on the state of carry (which is set after adding back the divisor), the flag itself can be used as the result bit to remove one time-costly jump from the loop. These initial improvements are shown in Fig 4.

The algorithm seems to waste a lot of time performing the complete set of operations on a remainder which must necessarily be zero until the most significant bit of the dividend is shifted in. Fig 5 shows the implementation of a quicker pre-division loop which passes control to the main division loop only when the



```
;
;                                                  cycles  code
;
DIV16A PUSH AF          ;Save AF.                  11      F5
       LD   HL,0        ;Clear remainder.          10      21 00 00
       LD   A,16        ;16-bit quotient count.     7      3E 10
;
DIVLPA SLA  C           ;Clearing result bit,       8      CB 21
       RL   B           ;shift next dividend        8      CB 10
       ADC  HL,HL       ;bit up into remainder.    15      ED 6A
       SBC  HL,DE       ;Subtract divisor.         15      ED 52
       JR   C,NOGOA     ;Skip if rem. too small     7 12   38 03
       INC  C           ;else set result bit        4  -   0C
       JR   DLTSTA      ;and go do next place.     12  -   18 01
NOGOA  ADD  HL,DE       ;Add divisor back.          - 11   19
DLTSTA DEC  A           ;Repeat for 16-bit          4      3D
       JR   NZ,DIVLPA   ;quotient.                (12/7)  20 EF
;
       POP  AF          ;Restore AF                10      F1
       RET              ;and exit.                 10      C9
;
;T.    start & end:                                48
;   loop pass 1-16: ([85,85:85]*16-5)             1355
;                                                 ----
;                  FIG 3                          1403
;
```

```
;
;...Exchange functions of A and B to use the quicker "RLA" and
;..."DJNZ" instructions inside the loop.
;...Use "CCF" to convert Carry flag to result bit ready for
;...shifting into C at start of next iteration, substituting
;..."RL   C" for "SLA  C" and deleting "INC  C","JR   DLTST".
;
;                                                  cycles  code
;
DIV16B PUSH AF          ;Save AF.                  11      F5
       LD   HL,0        ;Clear remainder.          10      21 00 00
       LD   A,B         ;A = dividend hi-byte.      4      78
       LD   B,16        ;B = 16-bit result count.   7      06 10
;
DIVLPB RL   C           ;Get result bit and         8      CB 11
       RLA              ;shift next dividend        4      17
       ADC  HL,HL       ;bit to remainder.         15      ED 6A
       SBC  HL,DE       ;Subtract divisor.         15      ED 52
       JR   NC,DLTSTB   ;Skip if +ve remainder     7 12   30 01
       ADD  HL,DE       ;else add divisor back.    11  -   19
DLTSTB CCF              ;Sub Cy become result.      4      3F
       DJNZ DIVLPB      ;Repeat for 16-bits.       (13/8)  10 F3
;
       RL   C           ;Get last result bit        8      CB 11
       RLA              ;into quotient.             4      17
       LD   B,A         ;Quotient into BC.          4      47
       POP  AF          ;Restore AF                10      F1
       RET              ;and exit.                 10      C9
;
;T.    start & end:                                68
;   loop pass 1-16: ([77,71:74]*16-5)             1179
;                                                 ----
;                  FIG 4                          1247
;
```

dividend MSB (most-significant byte) is moved out to carry. Each iteration of the MSB loop takes only 32 time states against the 77 time states that would be used in the division loop.

However, the saving is not

```
;
;...Utilise a pre-division loop to shift the dividend msb out to
;...Carry - quotient is reset at each bit place of shift.
;
;                                                  cycles  code
;
DIV16C PUSH AF          ;Save AF.                  11      F5
       LD   HL,0        ;Clear remainder.          10      21 00 00
       LD   A,B         ;A = dividend hi-byte.      4      78
       LD   B,16        ;B = 16-bit result count.   7      06 10
;
MSBLPC SLA  C           ;Clear result bit, shift    8      CB 21
       RLA              ;next dividend bit to Cy.   4      17
       JR   C,REMSLC    ;Divide when msb found      7 12   38 07
       DJNZ MSBLPC      ;else loop.                (13/8)  10 F9
       JR   EXITC       ;Exit if zero dividend.    12      18 0E
;
DIVLPC RL   C           ;Get result bit and         8      CB 11
       RLA              ;shift next dividend        4      17
REMSLC ADC  HL,HL       ;bit to remainder.         15      ED 6A
       SBC  HL,DE       ;Subtract divisor.         15      ED 52
       JR   NC,DLTSTC   ;Skip if +ve remainder     7 12   30 01
       ADD  HL,DE       ;else add divisor back.    11  -   19
DLTSTC CCF              ;Sub Cy become result.      4      3F
       DJNZ DIVLPC      ;Repeat for 16-bits.       (13/8)  10 F5
;
       RL   C           ;Get last result bit        8      CB 11
       RLA              ;into quotient.             4      17
       LD   B,A         ;Quotient into BC.          4      47
       POP  AF          ;Restore AF                10      F1
       RET              ;and exit.                 10      C9
;
;T.    start & end:                                68
;   loop pass 1-16: ([77,71:74]*16-5)             1179
;  msb loop saving: ((77-32)*65535/65536)       -  45
;   msb loop costs:                               12
;                                                 ----
;                  FIG 5                          1214
;
```

as great as at first seems likely. Only half the range of dividend values have at least one leading zero bit, only a quarter have two, an eighth have three, and so on. The total gross savings average out to fractionally less than the timing difference of one iteration, and this is considerably reduced by the MSB loop timing overheads.

Another way of speeding up the division is not to complement the result carry bits individually before shifting them into the quotient, but as the completed quotient outside the division loop. The net saving for this simple act is 48 time states, shown in Fig 6.

Post-loop complementing is not totally incompatible with the pre-loop MSB adjustment of Fig 5. The MSB loop would have to shift ones into the quotient instead of zeros, but adding it to the code in Fig 6 will save only a further 25 time states.

Fig 7 shows Wehner's method applied to the 16-bit division. One final addition is necessary on exit from the addition loop to return a valid remainder.

Again, a pre-division adjustment to move the divident MSB out to carry will save a further few time states, but the method rules out savings by producing the complement of the quotient.

```
;
;...Use Carry as complement of result bit, finally complementing
;...the quotient before exit.
;
;                                                  cycles   code
;
DIV16D PUSH AF          ;Save AF.                  11       F5
       LD   HL,0        ;Clear remainder.          10       21 00 00
       LD   A,B         ;A = dividend hi-byte.      4       78
       LD   B,16        ;B = 16-bit result count.   7       06 10
;
DIVLPD RL   C           ;Get NOT result bit and     8       CB 11
       RLA              ;shift next dividend       4       17
       ADC  HL,HL       ;bit to remainder.         15       ED 6A
       SBC  HL,DE       ;Subtract divisor.         15       ED 52
       JR   NC,DLTSTD   ;Skip if +ve remainder      7 12    30 01
       ADD  HL,DE       ;else add divisor back.    11  -    19
DLTSTD DJNZ DIVLPC      ;Repeat for 16-bits.       (13/8)   10 F4
;
       RL   C           ;Get last complemented      8       CB 11
       RLA              ;result bit to quotient.    4       17
       CPL              ;Complement quotient        4       2F
       LD   B,A         ;hi-byte, into B.           4       47
       LD   A,C         ;Get quotient lo-byte,      4       79
       CPL              ;complement it and          4       2F
       LD   C,A         ;return to C.               4       4F
       POP  AF          ;Restore AF                10       F1
       RET              ;and exit.                 10       C9
;
;T.    start & end:                                 84
;   loop pass 1-16: ([73,67:70]*16-5)             1115
;                                                 ----
;                    FIG 6                        1199
;
```

```
;
;...Apply Wehner's method.
;
;                                                  cycles   code
;
DIV16E PUSH AF          ;Save AF.                  11       F5
       LD   HL,0        ;Clear remainder.          10       21 00 00
       LD   A,B         ;A = dividend hi-byte.      4       78
       LD   B,16        ;B = 16-bit result count.   7       06 10
;
DSUBLE SLA  C           ;Clear result bit and       8  -    CB 21
       RLA              ;shift next dividend       4  -    17
       ADC  HL,HL       ;bit to +ve remainder.    15  -    ED 6A
       SBC  HL,DE       ;SUBTRACT divisor.        15  -    ED 52
       JR   C,DALTE     ;Skip if -ve remainder.    7 12    38 0D
DSLTE  INC  C           ;Else set result bit.      4  -    0C
       DJNZ DSUBLE      ;Repeat for next bit,     (13/8)   10 F4
       JR   EXITE       ;then exit.               12  -    18 0B
;
DADDLE SLA  C           ;Clear result bit and       -  8    CB 21
       RLA              ;shift next dividend       -  4    17
       ADC  HL,HL       ;bit to -ve remainder.     - 15    ED 6A
       ADD  HL,DE       ;ADD divisor.              - 11    19
       JR   C,DSLTE     ;Skip if +ve remainder.   12  7    38 F3
DALTE  DJNZ DADDLE      ;Repeat for next bit.     (13/8)   10 F6
       ADD  HL,DE       ;Make remainder +ve.       - 11    19
;
EXITE  LD   B,A         ;Quotient into BC.          4       47
       POP  AF          ;Restore AF                10       F1
       RET              ;and exit.                 10       C9
;
;T.    start & end:                                 56
;   loop pass    1: ([67,66:66.5]*1)                66.5
;   loop pass 2-15: ([58,67,67,66:64.5]*14)        903
;   loop pass   16: ([69,79,78,78:76]*1-5)          71
;                                                  ----
;                    FIG 7                        1096.5
;
```

TJ'S WORKSHOP

TWO SCREENS FOR THE MICROBEE

Here is a subroutine which will let you have two 64*16 separate screens in Microworld Basic without any need for machine code routines. Line 1030 is the workhorse and providing a=0 or a=4, then it can be used in any program.
I Florance

```
00100  REM Two screens for the
Microbee using basic statements
00110 A1$="SCREEN 2":FORI=1TOLEN
(A1$):POKE I+62463,ASC(A1$(;I,I)):
NEXTI:REM something to identify
the 2nd SCREEN
00120 CLS:PRINT"PRESS '1' TO GO TO
1ST SCREEN AND '2' TO GO TO
THE 2ND SCREEN"
00130 GOSUB 1000 :REM
A CALL TO CHANGE SCEEN?
00140 GOTO130
01000  REM SUBROUTINE to Change
screens
01010 A1$=KEY$:IF A1$<>"1"
AND A1$<>"2" THEN 1010
01020 IF A1$="1" THEN LET A=0 ELSE
LET A=4
01030 IN#3:OUT 12,12:OUT 13,
A:IN#0:RETURN
```

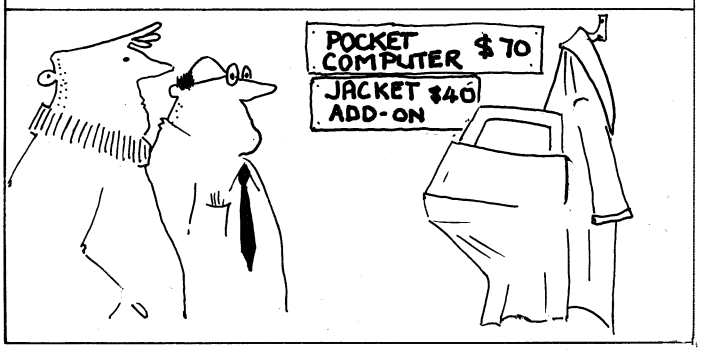
SPECTRUM ROUTINE

The "Spectrum draw to" routine which appeared in the January 1985 issue of *APC* isn't really much faster than the Basic routine. The comparison given in the Listing 2 isn't a fair one: the machine code routine has a step value of 10, the Basic routine a step value of 1, so the latter has about ten times as much work to do!

Compared on an equal basis, the code routine is only slightly faster. I've found the routine useful in machine code programs, but for Basic it is easier to write lines such as:

```
100 DRAW x-PEEK 23677,
      y-PEEK 23678
```

This draws a line from the current plot position, or end of a previous line, to (x,y), the absolute co-ordinates of the endpoint of the new line
M Davis



Basic listing

```
10      FOR I=0 TO 17
20      READ D$ :D=VAL("&"+D)
30      POKE#400+I,D
40      NEXT I
50      DOKE#229,#400
60      DATA 48,8A,48,98,48,A9,00,8D,74,02
70      DATA 68,A8,68,AA,68,4C,03,EC
```

IBM PC # REDEFINITION

Something like "GWBASIC LPT_INIS" included in an autoexec can make spreadsheet automatic pounding considerably more useful.

W Roberts

```
10      'Epson FX 80/100 printer £s and #s redefine. For IBM PC or Compatables
20 ON ERROR GOTO 280
30 CLS
40 LOCATE 10,24:PRINT"Printer inisalization program."
50 LOCATE 12,0
60 PRINT"This redefines the £ sign and the # to be the same as the screen."
70 READ N                       '  If no WHILE/WHEND then change 80 & 110 to:
80 WHILE N<>-1                  '  IF N=-1 THEN GOTO 120
90 LPRINT CHR$(N);
100 READ N
110 WEND                            GOTO 80
120 ON ERROR GOTO 0
130 LOCATE 20,23
140 PRINT"Printer inisalization done with."
150 SYSTEM                      '   This could be system, return, stop, or .
160                             '   Move rom chars. into ram
170 DATA 27,58,0,0,0
180                             '   Use ram chars.
190 DATA 27,37,1,0
200                             '   Define 35 as #
210 DATA 27,38,0,35,35,140,40,40,254,254,40,40,254,254,40,40,0
220                             '   Define 156 as £
230 DATA 27,38,0,156,156,139,18,18,126,146,18,128,2,128,66,0,0
240                             '   Let chars. 128-159 & 255 be printed
250 DATA 27,54
260                             '   End of data flag
270 DATA -1
280 IF (ERR=24)OR(ERR=25)OR(ERR=27) THEN LOCATE 20,24:PRINT"THE PRINTER IS PLAYI
NG DEAD.":BEEP
290 LOCATE 22,26:PRINT"HIT A KEY TO TRY AGAIN."
300 RESTORE '                       -Start again!
310 DEF SEG :POKE 106,0 '           -Empty= the keyboard buffer
320 IF INKEY$ <>"" THEN RESUME 10 ELSE GOTO 320
```

AMSTRAD LVAR

This short utility for the AWA Amstrad will print out all the variables and their current values used in a Basic program. Put a break point where the bug occurs and type GOTO 65500. The program should be MERGEd in from tape or disk rather than loaded in.

This routine uses a few variables of its own which should be avoided in your Basic program. These are LV,LVS, LVL and LVS$.

J Jack

```
65500 '-----------------------------------------------
65501 '- LVAR. - J.W.Jack - 1985.
65502 DEF FN lv(x)=PEEK(x)+256*PEEK(x+1) :lv=0 :lvs=FN lv(44679)-17
65503 FOR lvl=FN lv(44677) TO lvs-16
65504 IF PEEK(lvl)>0 THEN GOSUB 65508
65505 NEXT
65506  lvs=lvs-16  :   POKE  44679,lvs-INT(lvs/256)*256  :   POKE
44680,INT(lvs/256)
65507 END
65508 '- Label.
65509 lvl$=""
65510 IF PEEK(lvl)>128 THEN lvl$=lvl$+CHR$(PEEK(lvl)-128) :  GOTO
65511 ELSE lvl$=lvl$+CHR$(PEEK(lvl)): lvl=lvl+1 : GOTO 65510
65511 PRINT lvl$;
65512 '- Type & Value.
65513 lvl=lvl+1
65514  IF PEEK(lvl)=1 THEN PRINT"% = ";FN lv(lvl+1)  :  lvl=lvl+3:
RETURN
```

```
65515   IF   PEEK(lvl)=2   THEN  PRINT"$   =   ";CHR$(34);   :FOR
lvg=FN lv(lvl+2)    TO    FN lv(lvl+2)+PEEK(lvl+1)-1    :     PRINT
CHR$(PEEK(lvg));:NEXT   :PRINT  CHR$(34)  :   lvl=lvl+PEEK(lvl+1):
RETURN
65516    IF    PEEK(lvl)=4    THEN    FOR    lvg=1    TO    5:POKE
lvs+lvg,PEEK(lvl+lvg) :NEXT :PRINT " = ";lv :lvl=lvl+6 :RETURN
65517 lvl=lvl+2 : PRINT" is a FuNction." : RETURN
65518 '-----------------------------------------------
```

ATARI CLOCK

This listing is for all Ataris, and is a real-time clock which is accurate to about 10 seconds a day.

When run it provides a 12-hour digital clock in the top right-hand corner of the screen.

The program is interrupt-driven, leaving normal operations unaffected, and updates the time by looking at the Atari's 50Hz clock.

It is affected by scrolling or clearing the screen, but will reappear in the same place after one second.

M Maestranzi

```
9000 GRAPHICS 0:? "REAL TIME CLOCK":? :? "BY MAURO MAESTRANZI":? :? "TIME IS IN
HH:MM:SS FORMAT"
9010 TRAP 9030
9020 FOR A=1 TO 300:READ M:POKE A+1535,M:NEXT A
9030 ? :? "ENTER TIME: ";USR(1536)
9060 DATA 162,0,32,199,6,24,42,42,42,42,141,255,6,32,199,6,24,109,255,6,157,240,
6,232,224,3,240,11,169,58,141,251,2,32
9070 DATA 208,6,76,2,6,173,49,2,133,205,173,48,2,24,105,63,133,204,144,2,230,205
,169,0,141,14,212,169,79,141,36,2,169,6
9080 DATA 141,37,2,169,64,141,14,212,76,0,160,162,0,202,208,253,248,14,162,0,173
,243,6,105,1,141,243,6
9090 DATA 201,80,144,96,142,243,6,173,242,6,105,0,141,242,6,201,96,144,35,142,24
2,6,173,241,6,105,0,141,241,6,201,96
9100 DATA 144,20,142,241,6,173,240,6,105,0,141,240,6,201,18,144,5,169,1,141,240,
6,173,221,6,208,41,24,162,2,160,0,189
9110 DATA 240,6,72,41,15,9,16,145,204,104,136,106,106,106,106,41,15,9,16,145,204
,136,169,26,145,204,136,202,16,225,208
9120 DATA 169,0,145,204,76,98,228,142,254,6,32,226,246,174,254,6,142,254,6,32,17
0,246,41,15,234,174,254,6,96,0,0,0,0,0,0
9130 DATA 0
```

VZ-200 instant colour

This short machine code routine will turn the screen the colour you have put in the data — instantly!!

To call the machine code routine type X=USR (0) where needed in your program.

To get different colours you change the underlined number in the data.

The numbers for the different colours are:

0=GREEN 170=BLUE
85=YELLOW 255=RED

A Willows

```
00010 FORI=-28687 TO -28674
00020 READA:POKEI,A
00030 NEXT
00040 DATA33,0,112,17,1,112,1
      ,255,7,54,85,237,176,201
00050 POKE30862,241:POKE30863,143
```

Reversed REM

Labelling subroutines with REM statements that describe the functions of the subroutines is obviously helpful to the programmer who has trouble remembering what parts do what when designing a long program.

One way to make the subroutines stand out in the LISTing is to use inverse REM statements. But the VZ computer will not straight-

The enhanced keyboard

The short history of the PC has seen nearly as many (IBM) keyboards as does a touring concert pianist. Offerings to date have included the original PC keyboard, with its misplaced and undersized keys, two versions for the ill-begotten PC*jr*, the improved AT keyboard, and finally the 'Enhanced' keyboard. Easily recognised by the 12 function keys across the top and by its dedicated cursor pad, the Enhanced keyboard is IBM's current favourite.

Parallel to these improvements, the ROM BIOS that IBM puts into each machine has undergone a slow but continuous evolution. Changes have included support for new disk drives, video displays, and operating modes. Obviously, one of the more recent additions has been BIOS support for the new keyboard. Specifically, three new interrupt 16h functions have been added: AL = 10h (keyboard read), 11h (keystroke status), and 12h (shift status). In IBM's jargon, an 'Enhanced' keyboard requires 'extended' BIOS functions.

These new functions are used in the same way as were the old functions 0, 1, and 2. The values returned, however, differ slightly. The old functions do not recognise function keys F11 or F12, and they return the same values for duplicate keys. For example, loading the AH register with 0h and executing interrupt 16h causes the BIOS to wait until a key is struck. Pressing the Home key on the numeric pad (with NumLock off) returns AL = 0 (indicating an extended code) and AH = 47h — the same value as does the Home key on the dedicated pad. But using function AH = 10h returns AL = E0h when the dedicated Home key is used. This allows programs to distinguish, at the BIOS level, between different keys that have the same nominal meaning.

To take advantage of the Enhanced keyboard, programs must be able to determine whether the keyboard is supported by the BIOS installed in a

```
CSEG    SEGMENT PARA PUBLIC 'CODE'              ;Start CODE segment
        ASSUME  CS:CSEG,DS:CSEG,ES:CSEG,SS:CSEG ;Set by DOS Loader
                ORG     100H                    ;COM file format

ENTPT:          JMP     MAIN

NO_MSG          DB      "No "
SUPPORT_MSG     DB      "BIOS support for extended keys$"

MAIN    PROC    NEAR

                XOR     AX,AX                   ;Address low memory
                MOV     ES,AX                   ; with ES
        ASSUME  ES:NOTHING                      ;Tell assembler

                MOV     DX,OFFSET NO_MSG        ;Assume no support
                MOV     AH,12H                  ;Extended shift status
                INT     16H                     ;BIOS keyboard int
                CMP     AL,BYTE PTR ES:[417H]   ;Does data returned match?
                JNE     NO_SUPPORT              ; no

                XOR     BYTE PTR ES:[417H],80H  ;Toggle insert mode
                MOV     AH,12H                  ;Ask again
                INT     16H
                CMP     AL,BYTE PTR ES:[417H]   ;First match might be
                JNE     NO_SUPPORT              ; an accident

                MOV     DX,OFFSET SUPPORT_MSG   ;Success
NO_SUPPORT:
                XOR     BYTE PTR ES:[417H],80H  ;Restore insert state
                MOV     AH,9                    ;Display message
                INT     21H                     ; Thru DOS

                MOV     AX,4C00H                ;Terminate program
                INT     21H                     ;Thru DOS
MAIN    ENDP
CSEG    ENDS
        END     ENTPT
```

KBTEST.ASM: the source code for KBTEST.COM, a program that checks your BIOS to see whether it supports the Enhanced keyboard

A variable called STACKPTR is maintained that points to the next location in the command stack into which a string will be copied. As a new one is inserted, STACKPTR is incremented. And when the last increment causes STACKPTR to exceed the boundaries of the stack, it wraps back around so that the next command will overwrite and replace the first one. Thus, the command stack forms a circular queue in which up to 15 DOS commands are stored in their order of entry.

Each time a new command is entered, a second variable, LOCPTR, is set equal to STACKPTR. LOCPTR keeps track of the current position within the stack. When you press Up Arrow or Down Arrow, LOCPTR is decremented or incremented by one, a new address in the stack is calculated from it, and the text of the command at that address is output. Since DOSKEY sets aside 128 bytes for each entry, formulating the offset of the current command in the stack is as simple as multiplying LOCPTR by 128.

A third variable, called ZCOUNT, is functionally equivalent to LOCPTR but is different in value. ZCOUNT maintains a record of where we are in the command stack relative to a zero base. Each time LOCPTR is set equal to STACKPTR, ZCOUNT is zeroed, and each press of Up or Down Arrow in turn increments or decrements it. DOSKEY uses ZCOUNT to determine whether a request to go backward or forward by one more command will violate the boundaries of the command stack.

It usually doesn't take long to stock the stack with the maximum of 15 commands. But until capacity is reached, DOSKEY must prevent attempts to go back beyond the first one. Thus, the first time function 0Ah is called, DOSKEY initialises the command stack by writing a zero to the first byte in each entry. Then, before it replays a command, it checks the first byte. If that byte is zero, the command is invalid and DOSKEY ignores the keystroke. Once the buffer is full, all entries will begin with a legitimate count byte. To make sure this scheme works (and to be efficient), DOSKEY doesn't buffer the null entry you make when you press Enter without typing a command beforehand.

In order to output a command from the stack, DOSKEY calls the subroutine WRITE_COMMAND. WRITE_COMMAND takes as input the offset address of the string to be sent, and it assumes that the first byte of the string

Exchanging data dynamically

Microsoft's DDE protocol enables the flexible exchange of data between DDE-supporting applications running under Windows. Does it represent the end of integrated software as we know it? Robert Schifreen examines its attractions for software developers and users alike.

When the first version of Windows started shipping to customers, users were generally impressed. This new system allowed them to use graphics-oriented programs that could be controlled by a mouse and which shared a common user interface. What was especially exciting was that users were now able to run *more* than one program at a time, which was something that plain MS-DOS could not do on its own. As well as being able to multi-task, programs could also talk to each other: word processors could send data to spreadsheets, databases could send records to word processors, and graphics packages could send graphs to word processors.

Software developers, seeing the world of integration opening up, were clearly *not* happy. They had looked closely at all the software that would talk to each other, and noticed that each package had something in common. It came from Microsoft.

So the developers got out their cheque books and ordered the Windows Toolkit, which is what you need if you want to write graphics-based applications that run under Windows in a multi-tasking way. Lo and behold, the toolkit manual describes a system called a clipboard. which allows you to send images between applications. Conspicuous by its total absence was any information, however, on the way that Microsoft's own applications talked to each other, and how Windows managed all this communication.

With the recent launch of Windows version 2.0 (and Windows 386), Microsoft announced Dynamic Data Exchange, or DDE for short. The important word here is 'announced', as the system had been available to a limited extent within Windows 1.x from the start, but only Microsoft knew how to access it.

Now, though, the system is open to everyone. It is, in Microsoft's own words, a public, open system, available to all software developers to enable them to write applications under Windows (and, when it cometh, Presentation Manager) that talk to each other.

Until the Windows 2.0 toolkits become available, information on the subject of DDE is scarce, which is a shame, as DDE could become vitally important to users and developers alike in the office of the future.

This article explains what DDE is and what it can do. The information was gleaned from technical people within Microsoft, and from various internal Microsoft documents, normally used to brief technical staff and distributors. You don't need to be able to program under Windows to read it, however, as I've steered well clear of using the jargon that has grown up around the system. After reading it, though, you may well want to learn about Windows.

Messages

DDE is *not* a program. Neither is it a piece of hardware. It is simply a protocol — a set of rules for the flexible exchange of data which Microsoft hopes will be adopted by all software developers who write Windows applications.

Windows is a message-passing operating system (as explained in *APC*'s September preview of OS/2) and DDE is a subset of the Windows messages, using message numbers marked officially in version 1 as reserved for future expansion.

Briefly, a message-passing operating system works like this. Every time something happens in the system — for example, a key being pressed, a mouse being moved, an option being selected from a menu, and so on — a message is generated. This message is a piece of data containing fields describing what happened, why it happened, which application it happened in, and so on. The message is then placed on the message stack.

The heart of any properly-written Windows application (bearded, sandal-clad programmers call them 'apps') is a main loop which retrieves the top message from the stack and acts on it ap-

propriately. If the program were a word processor, for example, the sort of messages it would receive would be:
a) a key has been pressed, so echo it on the screen and add it to the file being edited;
b) the mouse has been moved upwards, so scroll the screen;
c) the window in which the program is running has been corrupted by another program, and needs redrawing;
d) the user has terminated the program, so finish;
e) the user has iconised the program, so put it at the bottom of the screen as an icon. Continue running in the background, processing any messages directed at this particular program but ignore, for example, keyboard messages as it is no longer the foreground task.

Messages are sent round the system both by applications, and by Windows itself. An application can, and does, call its own subroutines by sending messages to itself.

Who sends messages?

DDE is normally used by one Windows application to talk to another. Each of the applications in the conversation must be active; that is, it must be running. However, it does not have to be the current foreground task. It's quite possible for an application to be designed purely as a DDE server; that is, it runs as an icon on the screen, and does not interact with the user at all. It can, though, perform certain tasks when another application sends it a DDE message to tell it what to do. The possibilities here are enormous, though I don't know of any useful DDE servers that have yet been written. There is a demo one supplied with Excel, which feeds ever changing numbers into a spreadsheet to simulate real-time data acquisition in a financial application.

The two applications exchanging data do not have to be different. Because Windows allows multi-tasking, it is quite possible for two copies of, say, Windows Write to be running, and to converse with each other via DDE.

When a windows application is running, and analysing messages, it simply rejects any that it cannot deal with by setting various flags and passing the message back to windows or to another application. For an application to support DDE, it means including subroutines in the application to deal with incoming DDE messages properly. There are nine DDE-related messages in total.

The names I've used below are the full message names as used by Windows. The WM indicates a Windows Message (as opposed to, say, MM for Mouse Message).

WM_DDE_INITIATE

The 'Initiate' message is sent from one application to another, to start a conversation. It is similar to opening a file in a high-level language.

Each DDE conversation is between two parties — no more and no less — though *more* than one conversation can be happening at a time.

Included in the message are three pieces of information: firstly, the name of the application from which the message is sent; secondly, the name of the data file upon which operations are to be performed (for example, the name of the word-processing document to be accessed — the 'topic'); and, thirdly, the field within the specified topic, known as the item. If no particular topic or item is required, these can be left blank. An example of an Initiate message might be to ask for a word-processing document to be opened and accessed.

'DDE is normally used by one windows application to talk to another. Each of the applications in the conversation must be active; that is, it must be running.'

All applications receiving the Initiate message then inform the sender whether they are capable of accessing the required document. This is discovered by examining the document's extension to decide whether it is one that can be handled.

The Initiate message is sent in a way that makes it visible to all applications currently running. It is not so much a question of 'Can application x send me data y?', but 'Which application out there is capable of supplying me with data y?'

WM_DDE_ACK

When an executing application sees an Initiate message, it has to check if that message is relevant to itself. If it is, it sends an 'Ack' (acknowledgment) message back to the sender.

The Ack message also includes a number of status bytes, which the application uses to tell the sender whether it is capable of initiating a conversation, whether it is busy, and so on.

WM_DDE_TERMINATE

The 'Terminate' message is sent by one participant in a DDE conversation to say that it is terminating the data exchange. If you're used to programming in high-level languages such as Basic, then the Terminate message is similar to closing a file. Either party in the conversation can issue the Terminate message — not just the application that initiated the conversation in the first place.

WM_DDE_REQUEST

Having opened communications channels and buffers via the Initiate command, applications can now send data back and forth. The 'Request' message is sent by one application to request some data from another.

In addition to the name of the sender, the message contains two other pieces of information. Firstly, the format in which it wants the data, and, secondly, the actual piece of data it wants (which could be a paragraph of text, a graph, a range of cells from a spreadsheet, and so on). The format is a CF number, which identifies a number of what are called 'Clipboard Formats'. These are standard data structures used within windows, which I'll describe in more detail below.

WM_DDE_DATA

In response to a 'Data' message, an application can do one of three things. If it is not set up to support DDE, it will send back a message saying so (or will just ignore the message). If it can support DDE, but is busy, it will again send an appropriate message. Both of these messages will be of the Ack type, as described above, with bits set to indicate why the request will not be complied with. If the application wants to send back the requested data, it uses the Data message.

In the simplest case (avoiding Windows jargon) the message will contain two things: firstly, a pointer to say where the data is (the actual data is not sent within the message), and, secondly, a number to indicate which CF format the data is in. When an application requests data from Microsoft Excel, for example, that data (the location of which is returned in the Data message) will be found to be in plain ASCII format.

Extra status bits can be set in the Data message to give additional information to the application that will receive the data. You can, for example, request that an Ack is sent after the data has been processed, or that the data must not be corrupted by the application which receives it.

WM_DDE_ADVISE

'Advise' is a special form of the Request message and is sent by one application to another, to request that a piece of data be sent whenever its value changes. For example a dedicated DDE server could be written that calls up Viatel every half hour to download the latest share prices. A spreadsheet would then issue an 'Advise' message to the server, so that it could be kept up to date if any of the values on which it is operating change.

Even more useful, a DDE server could act as an interface between a company mainframe and a user's PC. If a manager is working out budgets on his desktop PC, a 'hot link' could be added to a spreadsheet that would, for example, automatically look up last month's sales figures from the mainframe and put it into the spreadsheet. Not only would this ensure that the spreadsheet was always up to date, but it would confine confidential data to the mainframe.

As with Request, the data requested (or rather a pointer to it) by an Advise message is also sent back through a Data message. A bit in the Data message is set or cleared, to indicate whether the data is the subject of a Request or Advise message.

WM_DDE_UNADVISE

'Unadvise' is the opposite of Advise. It is sent by the sender of an Advise message, to indicate t hat it no longer needs to be advised of changes in a particular data item. The application that had been previously sending the data would then acknowledge this with an Ack, as a guarantee that no further data will be sent.

WM_DDE_POKE

An application will not normally receive any DDE data unless it specifically requests it with the Request or Advise message. There may be times, though, when you want to send data to an application that hasn't requested it. The 'Poke' message is used to do this. Even though this message sends unsolicited messages, it cannot be used unless the Initiate message has first been used to open a channel of communications.

Equally, it will not be read unless the application to which it is addressed examines the message stack.

In all other respects, a Poke message is similar to a standard Data message. It contains a pointer to the data, plus an indicator to show which format it is in.

WM_DDE_EXECUTE

'Execute' lets you send a command to another application for execution, and is