White Crane

WHITE CRANE SYSTEMS

WCS DISK UTILITIES

for the

VICTOR 9000

Personal Computer

REQUIRES MS-DOS 2.11

$50.00

The White Crane Systems Disk Utilities package contains the following programs:

FPATH.COM	**- File Path facility**
MV.COM	**- Move files**
RM.COM	**- Remove files**
CP.Com	**- Copy files**
SORTDISK.COM	**- Sort Disk directory**
X.COM	**- Execute multiple commands**
PPORT.SYS	**- Parallel Port driver**
PBUFF16.SYS	**- 16K Print Buffer**
PBUFF32.SYS	**- 32K Print Buffer**
PBUFF64.SYS	**- 64K Print Buffer**

In addition, the following two public domain programs are provided free of charge:

SCRNSAVE.COM	**- Automatically Dim Screen**
SDIR.COM	**- Display Sorted Directory**

Fifteen pages of documentation are provided, designed to fit into the MS-DOS Users Guide.

NEW PROGRAMS ADDED TO THE WCS DISK UTILITIES
8 April 1986

Reformatted for PS Technical WordProcessor
by R.N. Folsom, 1 December 1989.

This file covers new programs have been added to the WCS Disk Utilities with version 2.00. (RNF: BK, CP, MV, and RM have been renamed BKF, CPF, MVF, and RMF.)

WHEREIS.COM is a utility for finding files on a hard disk that has many sub-directories.

BACKUP.COM (BKF.COM) is used to copy only new files from one drive or directory to another. A full page of documentation has been added to the manual covering BACKUP.

CHKLABEL.COM is a new utility for which printed documentation is not yet available. You can use CHKLABEL in a batch file to make certain that the correct diskette has been inserted. The command "CHKLABEL d:label" will set the error level to 1 if the diskette in drive d: does not have the Volume Label "label". Here is an example of its use with BACKUP:

```
:NEXT                   ;label to ask for next disk
REM Please insert diskette BACKUP3 in drive B:
PAUSE
CHKLABEL B:BACKUP3
IF ERRORLEVEL 1 GOTO NEXT
BKF C:\DIR3 B:
```

PARK.COM will park the read/write heads of all drives C: and above. Simply run the command PARK before turning off your machine. You may change your mind and continue operations by pressing any key to return to DOS.
If you have a single hard disk volume as A:, then use the program PARKA.COM instead of PARK. PARKA parks drive A:, skips B:, and parks all drives C: and above.

DOCUMENTATION FOR SCREEN.COM

White Crane Systems utility

Originally for Dbase III on the Victor 9000

The program SCREEN handles most of the IBM style screen and keyboard functions. It replaces another White Crane Systems utility called SCRNSAVE which performs only the one function of dimming the screen. If you are currently using SCRNSAVE, you must discard it and use SCREEN instead. Be sure to remove SCRNSAVE from your AUTOEXEC.BAT file and replace it with SCREEN, as SCREEN will not install properly if SCRNSAVE has already been run.

Screen performs three different user functions in addition to working with the BIOS enhancements:

The command "SCREEN 5" will automatically dim your CRT 5 minutes after you stop typing. Pressing any key, such as shift or ALT, will restore the screen to its previous brightness. You may specify a timeout value of from 3 to 30 minutes. The default timeout is 3 minutes.

There will be times when you do not want the screen turning off after a few minutes of inactivity. You can disable the screen save function with the command "SCREEN-D". Also, some programs bypass the Victor keyboard handler. If you run Lotus 123 with screen save active, your screen will go blank 5 minutes after you enter 123 even though you are typing. In this case, use the ALT-UP cursor key to raise the brightness of the screen. Before running programs such as 123 or Crosstalk, it is a good idea to disable screen save first, then turn it back on with "SCREEN 5" when you exit the program.

SCREEN provides you with a text Print Screen facility. If you press the CTRL (LOCK), ALT, and RIGHT SHIFT keys at the same time, SCREEN will dump a copy of your screen to the parallel printer. If you have the WCS Print Buffer (PBUFF) installed, the screen dump will go into the buffer rather than straight to the printer.

With SCREEN installed, you can reboot your Victor from the keyboard by pressing the CTRL-ALT-DEL keys at the same time.

Instead of the Victor Supplied parallel port driver PPORT.EXE, you can install a Print Buffer by putting the line "DEVICE = PBUFF64.SYS" in the file CONFIG.SYS in place of the line naming PPORT.EXE. The files PBUFFnn.SYS (where nn = 2, 4, 8, 16, 32, or 64) set aside 2 to 64K of system memory for use as a Parallel Port Print Buffer. If you have a large system (512K), you will probably want to use a 32K or 64K buffer. Smaller systems may dictate the use of a smaller buffer.

With a Print Buffer installed, characters sent to the Listing Device (LST: or PRN:) are not immediately sent to the printer. Instead, PBUFF stores them in its buffer. Printing is done in background whenever there is free system time. This way you can execute other programs while the printer runs. Free system time is found whenever keyboard input is called for. PBUFF then prints one character if the printer is ready. Your printer should print at full speed whenever the program running on the computer is expecting you to type something on the keyboard. The printer will slow down or even stop temporarily when you are running a disk intensive program such as FORMAT or DISKCOPY.

PRINT.COM (page 7-75) may be used to spool files to the printer, but this will be noticeable slower than using the command "COPY FILENAME PRN" to copy your file to the printer. Typically it may take 2-3 minutes to copy a 40K file into the Print Buffer, following which the printer will run for 15 minutes while you are able to do other work. Be aware that PRINT.COM expands tabs while COPY does not. If the file you are printing contains tabs, you may need to preset the tab settings on your printer.

You can clear out the print buffer by holding down the ALT key and pressing the CAN key. You can send a Form Feed to the printer with the ALT-EOL key combination. Not that these two features depend on the standard Victor definition of these keys as hex codes DE and DF. These keys may not put out the right codes if you use MODCON to install a special keyboard.

If you really have to conserve memory on your system, you can use the file PPORT.SYS. This is a direct replacement for the Victor parallel port driver PPORT.EXE. It requires 1K less system memory. Also, you can remove the serial port drivers PORTA.EXE and PORTB.EXE if you are not driving a serial printer. This will save an additional 7K of memory.

COPY FILES (CP) CPF

CP [/D] [/0] [/V] source [source] . . . [destination]

where source, destination = [path] [filename]

CP.COM is an external command that copies files. It is based upon the UNIX CP command and compliments the MV and RM commands. Its advantages over the COPY command are the use of multiple source files and the Verify option. It may also be used for additional safety as it will not automatically copy over an existing file.

CP copies the source files specified on the command line to the destination path\file. Wild cards may be used. For example:

```
CP  B:*.BAS
```

Copies all BASIC files from drive B: to the current directory, provided they do not already exist there. You may also specify a different destination:

```
CP  B:*.ASM A:
```

Copies all assembly files from B: to A: Unlike the COPY command, multiple files may be copied with one command. For example:

```
CP  *.COM  *.EXE  *.BAT  C:\BIN
```

Copies all binary and batch files to the \BIN directory on drive C:

Note that if there is more than one path or file name on the command line, the last one is taken as the destination.

CP has several option flags:

/D Destructive copy. Unlike the COPY command, CP will not automatically copy over a file which already exists. If the /D option is specified, CP will overwrite an existing file.

/O Override. This option causes CP to override the read-only attribute of any file and copy over it anyway.

/V Verify copy. If this option is specified, CP will display the name of the file that is about to be copied. You are given the choice to copy it or not. Valid responses are:

Y - Yes	**Copy this file.**
N - No	**Skip this file.**
C - Continue	**Copy this and all remaining files without verify.**
Q - Quit	**Skip this file and Quit.**

Your answers do not have to be upper case, and the action is taken as soon as one valid letter is typed. For example, the command "CP /V A:*.COM D:", followed by "ynnyync", gives the following output:

```
Copy A:CP.COM [yncq]? Yes
Copy A:FPATH.COM [yncq]? No
Copy A:MV.COM [yncq]? No
Copy A:RM.COM [yncq]? Yes
Copy A:SCRNSAVE.COM [yncq]? Yes
Copy A:SDIR.COM [yncq]? No
Copy A:SORTDISK.COM [yncq]? Continue
A:SORTDISK.COM
A:S.COM
          5 File(s) copied
```

The error level is set if an error is encountered, and may be tested in a batch file to branch to other operations if there was an error. (see MV.)

FILE PATH (FPATH)

```
FPATH [path] [;path] ...
```

FPATH.COM is an external command that allows the user to define a ''file path'' similar to the DOS PATH. When any application program attempts to open a file in the current directory, such as an overlay or data file, FPATH first tries to open the file in the current directory. If that fails, FPATH attempts to open the file in each directory specified in the File Path.

You may specify a list of path names and optionally drive letters, separated by semicolons. For example, to set the File Path to first search a RAM disk (D:) and then the BIN directory on drive C: you would use:

```
FPATH  D:\;C: \BIN
```

Entering FPATH with no parameters displays the current File Path, while FPATH; resets the File Path to no search.

FPATH is especially useful with programs that use overlays or ''help'' files. If these files are placed in the \BIN directory along with the executable program, and both the path and fpath contain the path \BIN, then you will be able to run the program from any drive or directory, and both overlays and help files will be available to the program from any drive or directory. This specifically does not work with dBase II overlays because dBase looks first to see it the file is present before trying to open it. Note that if an application attempts to open a file in a specific directory, only that directory is tried, not the whole File Path.

Programmers can use FPATH to let their compilers find headers, include files, object files, or libraries on a RAM disk or in a library directory.

MOVE FILES (MV) MVP

MV [/D] [/0] [/V] source [source] ... [destination]

where source, destination = [path] [filename]

MV.COM is an external command that moves files from one drive or directory to another. If the source and destination drives are the same, only the directory entry is moved to the new directory. This is much faster than COPY followed by DEL as no physical copy of data is necessary. If the destination directory is on another drive then the file is copied to the destination and the source file is deleted. Thus the MV command is more efficient than COPY followed by DEL, and safer as MV will not automatically write over any file.

If only one file path\name is specified, it is moved to the current directory. Wild cards are allowed in the file name. If a drive letter is not specified the default drive will be used. If a path is not specified the current directory will be used. For example:

```
MV   A:*.COM
```

Moves all .COM files from A: to the current directory. You may also specify a different destination:

```
MV   \SOURCE \*.ASM   \SOURCE\ASM
```

Moves all assembly code files from the \SOURCE directory to the \SOURCE \ASM directory. Since these are both on the default drive, only the file *names* will have to be moved.

If multiple file paths \names are given, the last is taken as the destination directory. Multiple source files can be named. For example:

```
    MV  *.COM  *.EXE  C: \BIN
```

Will move all .COM and .EXE files in the current directory to the \BIN directory of drive C:. The file names will not be changed. The command:

```
    MV  A:PROG.EXE  D:NEWPROG.EXE
```

Copies the file A:PROG.EXE to drive D: under the new name NEWPROG.EXE and deletes the original file.

MV has several option flags:

/D Destructive Move. MV will not automatically destroy a destination file that already exists. If the option /D is added to the command line, MV will Delete any destination files of the same names before it moves the source files.

/O Override. This option causes MV to override the read-only attribute of any file and delete it if necessary before a move.

/V Verify Move. This option causes MV to display the name of the file that is about to be moved, and have you verify the name. Valid responses are:

Y - Yes	**Move this file.**
N - No	**Skip this file.**
C - Continue	**Move this and all remaining files without verify.**
Q - Quit	**Skip this file and Quit.**

Your answers do not have to be upper case, and the action is taken as soon as one valid letter is typed. For example, the command "MV /DV *.COM \BIN", followed by "ynnc", gives the following output:

```
Move D:CP.COM [yncq]? Yes
Move D:EDT.COM [yncq]? No
Move D:RM.COM [yncq]? No
Move D:SCRNSAVE.COM [yncq]? Continue
D:SCRNSAVE.COM
D:SORTDISK.COM
D:X.COM
           4 File(s) moved
```

The error level is set as shown below if an error is encountered. This may be tested in a batch file to verify that the move occurred, and branch to other operations if there was an error.

```
0  No Error.
1  Invalid parameter.
2  Invalid number of parameters.
3  Parameter out of order.
4  Invalid directory.
5  No free file handles.
6  File creation error.
7  Insufficient disk space.
8  No files found.
```

REMOVE FILES (RM) RMP

RM [/O] [/V] [/Z] [path]file [path]file . . .

RM.COM is an external command that deletes files. It is based upon the UNIX RM command to remove (delete) files.

RM deletes the files specified on the command line. Wild cards may be used. Unlike the DOS DEL and ERASE commands, multiple files may be removed with one command. For example:

```
RM  *.OBJ  *.EXE  PROG.COM
```

Deletes all .OBJ and all .EXE files from the current directory, as well as the file PROG.COM.

RM has several option flags:

/O Override. If this option is specified, RM will override the read-only attribute of a file and delete it anyway.

/V Verify. This option causes RM to display the name of each file that is about to be deleted, and ask you to verify the deletion. Valid responses are:

Y - Yes	**Delete this file.**
N - No	**Skip this file.**
C - Continue	**Delete this and all remaining files without verify.**
Q - Quit	**Skip this file and Quit.**

For example, the command "RM /V \BIN" followed by "ynyynq" gives the following output:

```
Remove   \BIN\ CP.COM [yncq]? Yes
Remove   \BIN\ SCRNSAVE.COM [yncq]? No
Remove   \BIN\ FPATH.COM [yncq]? Yes
Remove   \BIN\ X.COM [yncq]? Yes
Remove   \BIN\ MV.COM [yncq]? No
Remove   \BIN\ PBUFF16.SYS [yncq]? Quit
```

/Z This option causes all specified files to be Zeroed first before deletion. This process of writing zeroes over the entire file ensures data security as the file cannot be recovered after deletion.

The error level is set as shown below if an error is encountered. This may be tested in a batch file to verify that the deletion occurred, and branch to other operations if there was an error.

0	**No Error.**
1	**Invalid parameter.**
2	**Invalid number of parameters.**
3	**Parameter of out order.**
4	**Invalid directory.**
5	**No files found.**

AUTOMATICALLY DIM SCREEN (SCRNSAVE)

```
SCRNSAVE n
```

SCRNSAVE.COM is an external command that dims your CRT screen from 3 to 30 minutes after the last activity from the keyboard or the screen. For example, the command:

```
SCRNSAVE 5
```

In your AUTOEXEC.BAT file will cause your screen to dim 5 minutes after you last press a key, or your program last displays a character. Pressing any key (including the shift keys) will restore the screen to its original brightness. You may run SCRNSAVE to change the time out value as often as you like. The default time out value is 3 minutes. The command:

```
SCRNSAVE -D
```

Will disable SCRNSAVE. That is, the screen will never time out. This is necessary for some programs, such as Lotus 123, which bypass the BIOS screen and keyboard routines. If you did not disable SCRNSAVE before running 123, the screen would go blank 3 to 30 minutes after you started. If this happens to you, simply use the ALT cursor keys to increase the brightness. Programs that write directly to the screen but still read the keyboard normally, such as Crosstalk, will allow the screen to dim even though they are actively writing to it. In this case you can turn the screen back on by pressing a shift key. To avoid frustration, you may want to use a batch file to disable SCRNSAVE, run 123, then enable SCRNSAVE automatically.

SORTED DIRECTORY (SDIR)

SDIR [path] [filename[.ext]] [options]

[options]	**/P - Print image - no esc sequences**
	/E - No screen Erase
	/C - Single Column display
	/H - List Hidden files
	/B - List file size in Blocks (clusters)
	*** /X - Sort by EXtension**
	*** /S - Sort by Size**
	*** /D - Sort by Date/time**
	*** /N - No sort, original order**

SDIR.COM is an external command that lists the files in a directory sorted in a chosen order. By default, SDIR lists all files in the current directory, sorted by filename.ext with screen erase and pause. You may specify any valid DOS path and obtain a listing of all or selected files in any directory.

The option switches allow you to change the manner in which the files are displayed. Multiple options are allowed, but only one of the four sort options (labeled with ★ above) may be used at one time. For example:

```
SDIR   \BIN   /H/X/C/P
```

Lists all files (including hidden files) in the \BIN directory, sorted by extension and displayed in a single column and in print image.

Note: Attribute column is blank if file is archived (archive bit = 1)
Contrary to Norton FA, + White Crane DiskTool, which report "archive" if archive bit = 1.

The bottom line displays the total number of files listed, the number of bytes they contain, and the number of bytes free on the disk. The /B option shows file sizes in Blocks instead of bytes. Blocks (or clusters) are the actual units of space allocated to a file. Thus a 40 byte file uses up one Block on the disk. Block sizes are different for Single Sided, Double Sided and Hard disks. You may specify a different block size to be displayed with /B=nnn or /B=nnnK.

Files will be displayed in a single column unless there are more than 8 files. More than 8 files will be displayed in two columns unless you use the /C option to force single Column display. The number of lines at which SDIR switches to two columns is the fourth byte in the program and can be changed to suit your preference using DEBUG (e.g. debug sdir.com, e103 14 (hex), w, q).

Unless the /E option is used, the screen is erased before the files are displayed.

The directory will pause when the screen is full until a key is struck. Headers and footer are displayed in bold with the file headers underlined. Since these features use escape sequences which interfere with a printer listing or use in a batch file, the /P option (Print image) turns off the pause and all escape sequences.

SORT DISK DIRECTORY (SORTDISK)

SORTDISK d: [/X | /S | /D]

where d:	**is the disk to sort**
/X	**sorts by file extension**
/S	**sorts by file size**
/D	**sorts by date and time**
default:	**sorts by filename.ext**

SORTDISK.COM is an external command that sorts the root directory of any disk. Since the sorted directory is written back to the disk, the files will remain in sorted order. Deleted entries are sorted to the end of the directory.

The disk may be sorted according to one of the following options:

/X Sorts the files alphabetically by file extension.

/S Sorts the files by increasing file size.

/D Sorts the files by file creation Date and time.

The drive to be sorted must be specified. If no option is specified the files are sorted alphabetically by file name and extension.

Upon successful completion the Error Level is set to 0. Failure sets the Error Level to 1. In batch files this may be tested with the IF command.

Note Attribute column is blank if archive bit = 1. (ie if file is archived)

EXECUTE MULTIPLE COMMANDS (X)

```
X[;][command][;command] ...
```

X.COM is an external command that provides a simple way to execute multiple commands and to create small batch files. A semi-colon is used to separate multiple commands on a single line and to continue to the next line. X creates a batch file, Z.BAT, containing all of the commands, then executes Z. If a semi-colon was included before the first command, Z.BAT is then deleted. If the leading semi-colon was not included then you can re-run the commands simply by typing Z.

The list of commands can be continued on additional lines by ending the current line with a semi-colon. If a semi-colon is required inside one of the commands, it may be inserted by using ;; to represent one semi-colon. For example:

```
X MASM  PROG,PROG;;  ;LINK  PROG,PROG;;  ;
EXE2BIN  PROG PROG.COM  ;RM  PROG.OBJ  PROG.EXE
```

Creates and executes a file Z.BAT containing:

```
MASM PROG,PROG;
LINK PROG,PROG;
EXE2BIN PROG PROG.COM
RM PROG.OBJ PROG.EXE
```

If you wish to keep the batch file you should rename Z.BAT to some other, more descriptive name. Otherwise Z.BAT will be written over the next time X is run.

RamDrive

INSTRUCTIONS FOR THE

WHITE CRANE SYSTEMS

RAM DRIVE

for the

VICTOR 9000

Personal Computer

REQUIRES MS-DOS 2.11

The White Crane Systems RAM DRIVE acts as an additional disk drive which is up to 50 times faster than a floppy disk and 10 times faster than a hard disk system. The size of the Ram Disk is chosen by the user, limited to the amount of available memory. Unlike all other Ram Drives, the White Crane Systems Ram Drive may be changed in size at any time without having to re-boot the operating system. Ram Drive 2.11 works with both Graphics and fixed disks.

CONTENTS

I. INSTALLING THE RAM DRIVE

The first thing you should do upon opening this package is **put a write protect tab on the disk and make a working copy of the programs.** The White Crane Systems Ram Drive consists of two files: RAMDRIVE.SYS and RAMDRIVE.COM. You should copy both of these files onto your system boot disk (either your hard disk or the floppy disk you use when you first turn on your computer). Do not proceed with any of the operations below until you have made your working copy and safely stored your original disk.

RAMDRIVE.SYS is an installable device driver under MS-DOS 2.00 and above. It is installed when DOS boots if it is named as a device in the file CONFIG.SYS. Edit the file CONFIG.SYS using EDLIN (supplied with the operating system) or any other text editor or word processor. Enter the line "DEVICE = RAMDRIVE.SYS" in the file and exit your text
* editor. A sample CONFIG.SYS file is included with Ram Drive.

You should now re-boot your computer. The Ram Drive will appear as the next drive letter available on your system. If you have two diskette drives (A: and B:) the Ram Drive will automatically become C:. If you have a hard disk as drive C: the Ram Drive will become drive D:, etc.

II. SETTING THE RAM DISK SIZE

The command to set the Ram Disk size is:

```
RAMDRIVE /S=nK /D=m /X
```

Where the parameters set the disk size, number of directory entries, and erase the disk respectively.

When your computer is first turned on, (or rebooted), the Ram Disk will have a size of near zero (256 bytes, the smallest allowed by the operating system). In order to make the disk large enough to be useful, you must run the program RAMDRIVE.COM to set the disk size. For example:

```
* buffers = 24
  files   = 20
  break   = on
  country = 1
  switchar = /
  device = ramdrive.sys
```

```
RAMDRIVE /S=200K
```

The above command enlarges the Ram Disk to 200 Kb. You may now copy files to drive C: (or whichever drive letter is next after your physical drives). This command may be placed in your AUTOEXEC.BAT file where it will be executed automatically on power up or re-boot.

III. CHANGING THE RAM DISK SIZE

A unique feature of the White Crane Systems Ram Drive is the ability to change the disk size at any time, without having to re-boot the computer. To change the disk size, simply enter the command:

```
RAMDRIVE /S=nK
```

Where n is the size in K that you want the Ram Disk to become (0 to 900 kilobytes). For clarity, you may want to add the letter K after the size, but this is not necessary.

If you try to make the Ram Disk too small for the files already present, the message ''ERROR: Change would have destroyed files'' will appear instead, and no change will be made. You will have to delete some files in order to shrink the Ram Disk to that size. Alternatively, you can specify the switch /X on the command line, causing RAMDRIVE to delete all files on the disk before changing size. For example, the command:

```
RAMDRIVE /S=50K /X
```

Will give you an empty 50K Ram Disk.

REMEMBER: The Ram Disk is temporary. If you wish to save your files, you must copy them to a physical disk before turning off the power or re-booting the computer. You should save important files often to guard against the possibility of a power failure.

IV. TECHNICAL INFORMATION

None of the information in this section need be understood in order to use the Ram Drive. It is included for the technically inclined user.

The Ram Disk defaults to 64 possible directory entries. This value may be changed by using /D=m on the command line, where m is the maximum number of files allowed on the disk (up to 2048). Making this number small (e.g. 16) can save up to 2K on a small system. For efficiency you should use multiples of 8. If the /S switch is not specified in the RAMDRIVE command, the disk size will not change, only the directory size. To make the Ram Disk as small as possible, enter:

```
RAMDRIVE /S=O /D=1 /X
```

Ram Drive uses a cluster size of one 256 byte sector. This makes for very efficient use of memory. On a hard disk, a 20 byte BATCH file might actually take up 4 or 8 K, while on the Ram Disk it will take up only 1/4 K.

The disk size specified in the /S parameter is the amount of free space on the disk, and does not include the space used by the FAT and directory. You can use the command CHKDSK to display the disk size and the amount of free memory.

Space for the Ram Disk is allocated at the top of memory. This is important to VICTOR 9000 users because it leaves the lower 128K of memory available for use with GRAPHICS. The transient portion of COMMAND.COM is reloaded below the Ram Disk whenever the disk size is changed.

Programmers may call RAMDRIVE.COM using the DOS EXEC function, thus changing the size of the disk from within an application. Ram Drive makes use of interrupt 84h and thus is incompatible with other software using the same interrupt (none known).

Disk Tools

cluster = group of sectors
cursor keys: page 13

WHITE
CRANE
SYSTEMS

Copyright 1984 by
Dr. Guy C. Gordon

DISK TOOLS DOCUMENTATION

WHAT DISK TOOLS WILL DO

Disk Tools is designed to be as simple to use as possible, while performing the complicated task of recovering deleted disk files. Disk Tools will also help you recover files and even parts of files from damaged disks.

When you erase a file under MS-DOS, using the DEL or ERASE commands, the contents of the file are not actually lost. All that is erased is the first letter of the file name, and the allocation table that tells DOS where the file resides on disk. All of the information in the file is there, waiting to be recovered, until you copy something else over it. If you stop immediately after your mistake you can almost certainly recover the file using Disk Tools. If you copy another file onto the disk, chances are good that you will wipe out at least part of the deleted file, but you may still be able to recover the rest of it. If you Format the disk, there is absolutely no way to recover what was once on it.

Disk Tools will let you find the deleted file entry in the disk directory, put back the first letter of the file name, and help you find where the file resides on disk. In fact, in most cases, Disk Tools will do all the work for you automatically, asking only simple questions such as "what was the first letter of the file name."

Disk Tools also provides an easy way to change file dates, times, and attributes.

In more technical terms, Disk Tools allows you to view and edit the file control areas of an MS-DOS disk known as the Directory and the File Allocation Table (FAT). You may view any sector on the disk as a hex/ASCII dump and as readable text. The FAT and Directory are presented in a tabular manner, allowing you to point to entries with the cursor keys and change any part of an entry. Using this feature you may change any part of a Directory entry such as file name, date, time, size or attributes. You may also edit the FAT to link any sector on the disk into your file. Using these tools you can recover sectors from a damaged disk or a deleted file "by hand" as it were.

The automatic file recovery feature will rebuild the FAT by linking together contiguous clusters. If the deleted file was not contiguous, Disk Tools will ask you to intervene and provide the needed information of where the file continues on disk. You will be able to determine this using the Sector and FAT display facilities. This procedure recovers files in place. You do not need a second disk to receive the recovered files.

If the above two paragraphs are not clear to you, please do not be discouraged. Disk Tools is really very simple in operation, and the technical parts will be explained more fully below. If you make a mistake and recover a file incorrectly, you can always simply delete the file and start over. And remember--understanding how best to use Disk Tools will prove invaluable to you when Fred, down the hall, erases your accounts receivable file next week.

FIRST THINGS FIRST

Upon receiving your Disk Tools package, please take the time to read the licensing agreement printed on the cover. When you have opened the package, FIRST put a write protect tab on the diskette, and second, copy all the files to your MS-DOS boot disk. Put your original Disk Tools diskette away in a safe place.

FOR THE IMPATIENT

Here are the steps to follow in recovering a deleted file: Run Disk Tools by typing DT followed by the drive letter of the disk containing the file. Under MS-DOS 2.11 you may specify the drive and path where the file was located. Use Function Key [F2] to display the directory. Point to your deleted file with the cursor keys and press RETURN. When Disk Tools asks, supply the first letter of the file name. When Disk Tools asks if you wish to save the recovered file to disk answer Y for yes. Exit by using the [F7] Function Key.

You can even combine the first three steps by specifying the drive, path, and file name on the command line. Disk Tools will automatically search the directory for your file.

However, I advise you to read the following instructions. They will help you understand not only what to do, but why you are doing it.

GETTING STARTED

To use Disk Tools to recover a deleted file (in this example on drive A:) first enter the command DT A:. Under MS-DOS 2.11 you can specify both a drive and a path name. If none is given, Disk Tools will use the default drive and directory.

Disk Tools is designed for use with any of the standard Victor keyboards. These keyboards define the function keys as hex codes F1 - FA or as B1 - BA, and the cursor keys as ^A, ^B, ^C, ^D. If you are using a program such as dBase II, Wordstar, or Spellbinder, that install a different keyboard you will need to use MODCON to install a Disk Tools compatible keyboard. The file DT.KB is supplied and can be installed with the command "MODCON DT.KB". You do not need to use it if your standard function and cursor keys work in DT.

Disk Tools will begin by showing you the "vital statistics" for the chosen disk drive. You may never need the information presented on this screen, but you can always return to it if you do need it. From this screen you can exit Disk Tools by pressing Function Key 7.

At the bottom of the screen is a short description of the purpose of each of the Function Keys [F1] - [F7], and a status line showing the current disk drive, directory path (MS-DOS 2.11 only), cluster and sector. Throughout this document the wide Function Keys across the top of your keyboard will be referred to by number and in brackets (e.g. [F1] for Function Key 1).

Disk Tools uses the Function Keys to make it easy for you to move around on the disk, looking at various parts of it before making changes. Any time the Function Keys are displayed at the bottom of the screen you may press one of the keys [F1] - [F7] to perform its function. For example if you are in the Directory with the cursor pointing to your file, you can press [F5] to view the first sector of that file.

The Function Keys perform the following functions:

[F1] HELP	- displays help for the current screen.
[F2] DIR	- displays the current directory.
[F3] NAME	- changes drive and directory and searches for a file you name.
[F4] FAT	- displays the File Allocation Table.
[F5] SECTOR	- displays the Sector listed on the status line.
[F6]	- prints the directory or FAT
[F7] EXIT	- exits from the current screen.

FUNCTION KEY [F1] HELP: Help is available at almost every point within Disk Tools. Pressing the [F1] key will display a help screen designed to give useful information about the part of Disk Tools with which you are currently working. Thus when you are in the Directory display, [F1] will give you help on what you are seeing and what you can do from the directory display.

FUNCTION KEY [F2] DIR: This function displays the disk directory and allows you to choose a file by pointing to it. The directory is presented one sector at a time, which is 16 entries on a normal MS-DOS disk, but may be fewer on a RAM disk. Use the WORD <- and -> keys to page through the directory one sector at a time.

The file name, extension, size, date, and time displayed are the same as you would see if you simply executed the DIR command from DOS. However in the Disk Tools directory you will also see deleted and hidden entries, as well as two other fields (PTR and Attributes).

Deleted files have had the first letter of the file name replaced with a hex E5, which is displayed as the character sigma (all other characters in a file name are upper case). The PTR field indicates the starting cluster of the file (i.e. where the file is located on the disk). The file Attribute may be any combination of Read-Only, System, Hidden, Archive, Volume ID, Directory, or deleted. If you are unsure of what any of these attributes mean, please see your MS-DOS manual for an explanation. Disk Tools shows you both the number which is the attribute, and a list of what that number means. An attribute of 0 means a normal every-day file.

When you enter the Directory display, the cursor is on a message telling you that the RETURN key displays more file entries. If you press the down cursor key, the cursor will move down to the first file name. At the same time the status line at the bottom of the screen (which had been displaying the current directory sector) will display the starting cluster and sector of this file.

Moving the cursor up and down with the cursor keys allows you to choose any file on the screen. When you have chosen a file, you can press [F4] to look at its linked list in the FAT, or [F5] to display the starting sectors of the file, or RETURN to go on to the File Access Screen. If you choose a deleted file, Disk Tools will first ask if you want to recover it before going on to the File Access Screen.

Under MS-DOS 2.11 you will see directory names listed as well as file names. You can change directories from this screen by pointing to a directory entry and pressing [F2].

<u>FUNCTION KEY [F3] NAME</u>: Pressing [F3] allows you to enter a new drive, path or file name. If a file name is given, Disk Tools will search the directory for the file, and if found, will take you to the File Access Screen. If the file you name has been deleted, Disk Tools will ask if you want to recover the file.

For example you could enter B:REMIND.COM and Disk Tools would switch to drive B: and search the current B: directory for the file REMIND.COM. Under MS-DOS 2.11 you could enter \XTALK\IN and Disk Tools would switch to that directory on the current drive. You could also combine all three by entering B:\XTALK\IN\REMIND.COM.

If you wish to change diskettes while running Disk Tools, you should first use [F7] to go back to the Disk Information Table (the first screen). Now change disks and use [F3] to tell Disk Tools to read the new drive. Changing diskettes without using [F3] can destroy data.

<u>FILE ACCESS SCREEN</u>: The net result of choosing a file with either [F2] or [F3] is to bring you to the File Access Screen (unless the file is deleted, in which case you are first asked if you want to recover the file). This screen displays all the directory information about the chosen file, and allows you to make changes. A menu is presented giving you the choice of changing:

1) the file name
2) the file size
3) the file date and time
4) the first cluster pointer
5) the file attributes
6) delete the file

You choose one of the above functions by pressing the number indicated (not the Function Keys since they take you to the other Disk Tools screens). Each of the above functions will ask you to enter a new value for that particular field. If you change your mind you may press RETURN to make no change. Or you can enter the new name, or date, etc. and then press RETURN. Number 6 is slightly different. Disk Tools will show you the clusters linked to this file (from the FAT) and ask that you confirm deleting this file. If you answer Y for yes the file will be deleted.

You may make as many changes to this directory entry as you like, even just to experiment. The results of the change are displayed, but none of the changes are made permanent by writing the directory back to the disk until you try to leave the File Access Screen. If you have made changes, and you press a Function Key, such as [F7] to exit, Disk Tools will ask if you want to save those changes to disk. If you answer Y for yes, the changes are made permanent on the disk. If you press N for no the changes are discarded (not written to disk).

FILE RECOVERY SCREEN: When the file you choose with [F2] or [F3] is a deleted file, Disk Tools takes you to the File Recovery Screen. You are asked to supply the first letter of the file name. If you press RETURN at this point, you will move on to the File Access Screen discussed above. If you enter any letter (A - Z) Disk Tools will try to recover the deleted file. You do not have to enter the original first letter of the file name, but that is most likely the letter you want.

Disk Tools will now proceed to recover the file by rebuilding the FAT. The directory entry for your file contains a pointer to the first cluster of this file. Thus we know where to start. Unfortunately, the linked list in the FAT which tells where the file continues was erased when the file was deleted. Disk Tools attempts to rebuild this list by making assumptions that match how MS-DOS creates files.

When MS-DOS writes a file, it starts with the first free cluster on the disk, then continues in the next free cluster, whether it is contiguous or not. Thus your file might end up scattered over a large area of the disk, or it may be in one contiguous group of clusters (called an extent).

Disk Tools first assumes that your file is contiguous. This is a very good assumption for a fairly new disk onto which you have just copied files. Older disks on which files have been deleted and new ones copied are more likely to be fragmented (i.e. the files will not be contiguous).

Disk Tools displays the clusters it is linking together as it recovers your file. If enough contiguous clusters are available for the size of file you are recovering, Disk Tools reports its success and asks if you want to save these changes (i.e. the file recovery) to disk, before taking you on to the File Access Screen.

If Disk Tools runs into a cluster which is not free, then the file must not have been contiguous. In this case you will be asked to provide the next cluster number to link to this file. Disk Tools will helpfully suggest that the most likely number is that of the next free cluster, and tell you what that is. The best thing to do here is to press RETURN to accept the cluster number that Disk Tools has suggested. Instead, you may enter any cluster number you like, or you may press a Function Key such as [F4] to look at the FAT or [F5] to display sectors in order to find the rest of your file. If you enter a cluster number, Disk Tools will continue linking consecutive clusters until it either has enough clusters to hold the file size, or it again runs into an allocated cluster.

The major problem with file recovery under MS-DOS is that there is no automatic way to know if the next free cluster really is part of your file. That information was erased when the file was deleted. Only <u>you</u> can tell if a cluster is part of your file. If the file is readable text, such as a letter or a report, then it is very easy to spot the next cluster of your file just by looking at it. Binary files, such as .COM and .EXE programs are more difficult. Generally you have to run the recovered program to see if it works. But you can still look at the clusters to be certain you are not linking text with the binary file you are trying to recover.

To this end, Disk Tools allows you to look at the File Allocation Table (FAT) with [F4] to see which clusters are free, and to view sectors on the disk with [F5] to see if they contain parts of the file you are trying to recover. You should always view the first sector of each cluster in a recovered file, even if Disk Tools recovers the file without having to ask you to intervene. If a wrong cluster does get linked into your file, you can simply delete the file and try again.

<u>FILE ALLOCATION TABLE [F4]</u>: The FAT is where MS-DOS keeps information on which clusters on the disk are free and which are allocated to files. A cluster is simply a fixed number of sectors in a row. The Disk Tools drive information table will tell you how many sectors there are in each cluster on your disk.

There is one FAT entry for each cluster on the disk. Each FAT entry is a hexadecimal number from 0 to FFF. (In hexadecimal, the number 9 is followed by A (10), then B (11) up to F (15), then 10 (16), 11 (17) etc.) If a FAT entry is 0 then that cluster is free and may be written over when a new file is created. Any number from 2 to FF7 indicates that the cluster is part of a file, AND points to the next cluster in the file. The number FFF is used to indicate the End of File. (FF7 is also used to indicate a bad or reserved cluster if it is not linked to a file.)

As an example of how the FAT works, consider the file COMMAND.COM on a bootable single sided floppy diskette. (You may want to use Disk Tools to actually look at an MS-DOS boot disk while following this example.) In the Drive Information Table (first Disk Tools screen) we see that a single sided disk has 4 sectors per cluster.

Looking at the directory with [F2], we see that COMMAND.COM is the second directory entry, right after the hidden file MSDOS.SYS. The value PTR tells us that the file starts at cluster 2. (Cluster 2 is the first cluster on the disk--0 and 1 are not valid cluster numbers.) Since COMMAND.COM is 19456 bytes long (MS-DOS 2.11) it will use up 10 clusters on this disk. If you use the cursor keys to move the cursor over the file name, the status line at the bottom of the screen will display the starting cluster and sector numbers.

Pressing [F4] displays the FAT. Since we were pointing to COMMAND.COM, the cursor is now pointing to the FAT entry for the first cluster of that file. At the bottom of the screen the status line tells us that we are pointing to cluster 2. Also, a line asks us if we want to enter a new value for Pointer 2.

The number in FAT entry 2 is 3. This means that the next cluster in COMMAND.COM is cluster 3. Note that the number under the cursor is a pointer to the NEXT cluster in the file, NOT the current cluster number. The current cluster is shown on the status line.

The number in entry 3 is 22. This means that the file is non-contiguous. Instead of continuing in cluster 4, it continues in cluster 22. Entry 22 contains a 23, and so on to entry 29 which contains an FFF indicating that the file does not continue anywhere (End of File). While the cursor is on FAT entry 2, all clusters linked together in this file are highlighted. This makes it easy to see just how fragmented the file is. For COMMAND.COM we see that the file is contained in two extents.

Very seldom will you want to change a FAT entry directly. If you do enter any new values, Disk Tools will ask if you want to save the changes to disk when you exit the FAT display.

For a single sided diskette the entire FAT fits on one screen. For larger disks the FAT is displayed one screen at a time. Pressing WORD -> pages to the next screen. The WORD <- key will back up a page. A line at the top of the screen tells you which clusters are displayed on this screen. The status line will tell you where the cursor is, and clusters on this screen will be highlighted if they are linked to the cluster the cursor is on--even if the cursor is off screen.

Using the cursor keys, you may move the cursor to any cluster. The current cluster and sector numbers are displayed on the Status line at the bottom of the screen. The first two entries in the FAT (0 and 1) do not point to clusters on the disk. Rather, these numbers FF8 FFF identify the beginning of the FAT. Pointing to any FAT entry, we can view the contents of those disk sectors using.[F5].

<u>VIEW SECTORS [F5]</u>: Any time you press [F5], Disk Tools will display the contents of the sector shown on the Status Line. If you are in the directory [F2], this is the first sector of the file the cursor is pointing to. If you are in the FAT display [F4], this is the first sector of the cluster the cursor points to. When you are at the Drive Information Table, or in the FAT display with the cursor pointing to the FAT Identification entry, [F5] displays the sector 0 (normally the boot sector).

The sector display splits the screen into two parts. The top 16 lines are used to show the contents of the sector in both a Hexadecimal Dump with 16 bytes per line, and in ASCII (text). This is the same kind of dump that DEBUG displays. The lower part of the screen shows the sector as seven lines of continuous text. If the sector contains text, such as a letter, the bottom display will be the most readable, whereas binary information will be readable in the HEX/ASCII dump.

The HEX/ASCII dump can only display 256 bytes, or half a sector on one screen, while the text display at the bottom shows the entire 512 byte sector. Pressing the space bar will display the second half of the sector in the HEX/ASCII dump.

Use the UP and DOWN cursor keys to move the display forward or back one sector. The WORD <- and -> keys move you forward and back by clusters.

You should always use the Sector display to page through a file that you have just recovered, verifying that every cluster is really part of your file. Note that where your file is non-contiguous, you do not want the NEXT cluster, but rather the next cluster linked to your file. In this case, use the LEFT and RIGHT cursor keys to move to the previous or next cluster within your file.

FUNCTION KEY [F6] PRINT: When in the FAT [F4] display, or the Directory [F2], the [F6] function key will print a copy of the entire FAT or the current Directory on your printer. A hard copy of the FAT is especially useful when recovering a large, discontinuous file, or multiple files.

<u>FUNCTION KEY [F7] EXIT</u>: The [F7] key is always the way to exit the screen you are in. If you are in a sub-function such as changing the file name, [F7] returns you to the previous screen (in this case the File Access Screen). If you are in one of the main function screens, such as the Directory or FAT displays, [F7] returns you to the Disk Information Table--the first screen on startup.

From the Disk Information Table, the [F7] key exits to DOS.

RECOVERING FILES

Most files are very easy to recover using Disk Tools. Often, there will be a "hole" in the FAT (an area of un-allocated clusters) exactly the size of your file, and starting where your file started. In this case all you need to do is follow these steps:

1. Enter Disk Tools with the command:
   ```
   DT filename.ext
   ```
   or:
   ```
   DT d:\path\filename.ext
   ```

2. Disk Tools will search the directory for your file, and take you to the File Recovery Screen. Alternatively, you could use [F2] to find your file.

3. When asked, provide Disk Tools with the original first letter of the file name.

4. Disk Tools will show you the linked list of clusters it rebuilds for your file, then announce the successful recovery of your file and ask if you want to save these changes to disk.

5. Answer Yes to the above question, then use [F5] to go to the Sector Display screen. Here you should see the begining of your file.

6. Use the LEFT and RIGHT cursor keys to view the first sector of each cluster linked in your file. You could also use the UP and DOWN arrows to see each sector in the cluster, but it is only necessary to check the beginning of each cluster.

7. When you have verified that everything linked to your recovered file is part of the original, use the [F7] key to exit Disk Tools.

If you have trouble with any of the above steps, there are still things that can be done to recover all or part of your file.

The most common problem arises when your file is not contiguous. In this case, Disk Tools reports this to you in the File Recovery Screen, when it tries to re-build the FAT linked list. When you first try to recover a file, you should always use the clusters Disk Tools suggests should be linked together. When enough clusters have been linked to account for the file length, and you are asked if you want to save these changes, answer Yes, even if you are not sure these are the right clusters.

Next, step through the file in the [F5] display. Write down the clusters that are not a part of your original file. When you have gone through the entire file, as Disk Tools recovered it, go back to the File Access Screen (using [F2] or [F3] to name your file) and use function 6 to delete the file. Everything is now exactly as it was before you recovered the file.

Now you can recover the file again, only this time skip the clusters that you have listed as not being part of your file. Be sure to repeat the process of viewing the linked clusters untill all of them belong to your file.

Another method that is sometimes useful in making Disk Tools recover your file, is to recover another, smaller file first. Since this allocates clusters to the smaller file, Disk Tools will not try to allocate them to your file, but will skip around them.

If you really feal you know what you are doing, you can edit the FAT directly from the [F4] display. You can really mess up a disk by changing the FAT incorrectly, so be sure to be working with a DISKCOPY of a floppy diskette, or to have backed up everything important on your hard disk.

To recover a file by editing the FAT, make a list of the clusters you want linked to the file by browsing through the Sector [F5] display. Then build the FAT linked list by hand. Use the Change Name function in the File Access Screen to get rid of the E5 character in the file name.

You can also use this method to link random clusters from the disk into a file. Pick a deleted file entry, or a file you don't need from the Directory [F2] display. In the File Access Screen, use function 4 to change the file pointer to the number of the first cluster you want to link. Then edit the FAT from the [F4] display to link together the rest of the clusters you want. Be sure to change the file size to include all the clusters you link together. Using the numbers in the Disk Information table, multiply the number of clusters by the number of sectors per cluster, and then multiply by the number of bytes per sector. Use this number as the file size. (e.g. 3 clusters * 4 sectors * 512 bytes = 6144 bytes.)

You can also edit the FAT in order to allocate some clusters to make sure that they are not used. For example, to keep Disk Tools from allocating a cluster to your recovered file you could mark it with a 1 in the FAT. Then you could change the 1 back to 0 after the file is recovered. The number FF7 is used by MS-DOS to mark bad clusters. These clusters will not be freed by CHKDSK and DOS will never write on them.

Sub-directories that have been removed with the RMDIR or RD command can be recovered using Disk Tools. Since DOS does not record the size of the directory in the directory entry, Disk Tools will only recover the first cluster of a sub-directory. This is usually all there is. If you have a very large sub-directory that spans clusters, you will have to find the next directory cluster using the VIEW [F5] command and then link it to the first directory cluster by editing the FAT [F4]. Simply point to the first directory cluster and change the FFF you find there to the number of the next cluster. Then change THAT cluster pointer to FFF.

FIXING THE BIG MISTAKE

You're working on your dual floppy system, copying some files from the diskette in drive A. Now you walk over to your Victor 9000 with the internal hard disk. You put in the floppy and enter DEL A:*.*. You've just erased every file on your hard disk Volume A.

Yes, Disk Tools can recover the files, but it will be difficult. When you need to recover just one file, there is generally a hole in the FAT that exactly fits your file. At the very least, you know your file isn't where all the other files are located. But when all the files have been deleted you're lacking all those clues.

The best strategy to use in this case is to recover the easiest files first. You have to be careful not to allocate the wrong cluster to a file, since you will need that cluster later. Therefore I recommend that you recover your files in the following order:

1. All one cluster files.
2. Text files, starting with the smallest.
3. .BAS files, starting with the smallest.
4. .COM & .EXE files, smallest first.

The Disk Information Table displayed when you start up Disk Tools will tell you the cluster size in sectors and the sector size in bytes. Thus on a Double Sided floppy a cluster is 4 sectors of 512 bytes each. This means any file less than 2048 bytes fits in one cluster. Since we always know the starting cluster of a deleted file, we can recover these one cluster files with complete certainty.

Next you should recover all text files, including letters, documents, source code, etc. These are easy to recover since you can use the [F5] View Sector function to read the file and be certain it is correct. It is also best to start with the smallest files and work up.

After that, any .BAS files should be recovered. This is because you can recover the file and load it in BASIC where it is readable. You should do this to verify the correctness of each .BAS file you recover.

Finally recover the binary files (extensions .COM and .EXE). You cannot read these files, but you can try running them to see if the recovered file works. Often you will not even need to recover these files as you will have backup copies on your original product disks.

A NOTE FROM THE AUTHOR

I want to thank everyone who buys my programs for their support. But I'm not sure all of you realize that when you give away a copy, your friend gives away two copies, and each of his friends give away more copies.

Low price does not seem to discourage people from stealing software. Copy protection does, but makes the software harder to use. Instead, I have devised a copy labeling system. Every copy of Disk Tools has a serial number registered to its owner. This serial number is displayed on start up and is also encoded in an unspecified number of places within the program. This will allow me to track pirated copies back to the original owner.

Note that this will be no impediment to others giving away the copy you gave them. They won't care that they are spreading around a program with your name on it. It is up to you to keep your copy secure. Remember, when you give away software to someone, they are not the only ones stealing it--you are.

Please don't give away copies of my software. The next time someone asks, suggest that for a $3500 machine they can afford to spend $85 on some very useful utilities. Besides, ordering from me gets them on the mailing list for future offerings of both free and low-priced utilities.

Ultimately YOU will be the beneficiary of making others pay for their software. I am attempting to make White Crane Systems a source of quality software designed specifically for the Victor 9000. While companies like Lotus and others are abandoning the Victor I am trying to expand my support for this superior machine. You have already helped by purchasing this program. Please make others pay for their copy.

SST+

INTRODUCTION TO SCROLL SYSTEMS' SST+

From Michael Wishnietsky's PUB Bulletin Board

#278 (of 290) CH CARLISLE, on 12-MAR-86 01:58
Subject: SST+ ver.1.2

Scroll Screen Tracer is a high-powered replacement for Debug. Since I'm not an assembly language programmer I find SST+ a little intimidating, but there are a few functions that I use fairly often.

Display/Patch

To look at or modify any file that can fit in memory just type

```
    SST <filename>
    D
```

You get a full screen display in standard Debug format with hex on left and corresponding ASCII characters on right side of screen. Works just like a word processor.

Cursor control:

- horizontal cursors move 1 hex digit left or right
- vertical cursors scroll the screen up and down
- scroll key moves one page up and down
- tab key moves cursor from hex display to corresponding position in the ASCII display or from ASCII back to hex display
- ^R (alt-R) homes the cursor to the address given in the D command, 100 assumed if omitted.
- ^T (alt-T) tags the cursor position, and
- ^G (alt-G) returns cursor to tagged position.

Display control:

- ^A (alt-A) toggles screen display from hex/ASCII to only ASCII so that you get a lot more information on the screen - useful if you're interested only in the text content of the file.

Updating:

. ^O (alt-O) toggles from display mode into overtype mode; anything you type will overwrite hex digit/character at cursor.

Exiting display mode:

. a carriage return gets you out of display/overtype mode.

Changes are made in memory only, not on disk. If you want changes saved then use the Write command same as Debug. SST+ can write any file including .EXEs.

You don't have to memorize any of the above keys because SST+ displays a one line menu at top of screen giving most or all of the keys. Typing ? will get you a help screen applicable to the menu displayed.

Unlike word processors, SST+ has no insert mode; moving things around in a program will screw up all the addresses.

I use the display mode for looking at programs, patching, or replacing unwanted messages or carriage return/line feeds with binary zeroes.

Search

Unfortunately you can't search in display mode like a word processor does. However SST+ searching is much easier than Debug. Eg:

. SST <this document>
. sLcx "SST" [lists all locations of string "SST"]
. d [point cursor at one of the locations listed above] <cr>
. s [repeat previous search]
. d [point cursor at another location listed above] <cr>

The first search is from beginning of this file

(loaded at 100h) for a length equal to the size of this file (the length is in register CX). The search lists the addresses where it found "SST", up to 12 addresses per line.

To display the memory just type D, point cursor at the first address listed by search and then hit carriage return.

To display all memory areas containing "SST" just repeat the search without arguments (SST assumes same arguments as previous search) and then display another address.

Faster to do than to explain. Why? Because in the above example we never had to type an address or register value. Same commands as Debug but with lots of short-cuts. Like searching for assembly language instructions rather than the equivalent hex digits.

Unassemble

Same as Debug except that

- . down cursor/space bar unassembles next instruction
- . scrl down unassembles next screenful of instructions

Trace

Debug is adequete for most things but hopeless at tracing. If you're a raw beginner and you want to try a little tracing then forget it, or buy SST+.

Typing T puts you into full screen single step trace mode. Hitting the space bar executes one instruction and holding down the space bar runs the program in super-slow mode which is as fast as I dare to go.

You need a good assembly language reference manual and Microsoft's MSDOS Programmers Reference Manual ($40) that documents function calls & interrupts.

Disk Display/Modify

The disk function lets you edit the disk directory and FAT, although this feature is nowhere near as slick as White Crane's Disk Tools. SST+ is useful if you encounter some problem that Disk Tools can't fix.

Conclusion

I've described about 10% of what SST+ can do. Scroll Systems mentions that the manual is a reference and not a tutorial. To use the remaining 90% of SST+, I'll have to do some book learning and assembly language programming. Scroll Systems pitches SST+ as an assembly language interpreter (use like BASIC - no Assembler/Linker required), powerful debugger, disk utilities - all in one.

Do you need it? For tracing or assembly language programming, yes. Especially if your time is worth more than $1.00 per hour. For the other stuff, Debug is adequete and free although SST+ is nicer.

SCROLL SYMBOLIC TRACER

An 8086 Family Symbolic Debugger/Interpreter

by
Murray Sargent III
Lillard Darwin Sanders III

8087 display/modify facility by
Matt Derstine

Scroll Systems, Inc., 5530 N. Camino Escuela, Tucson, AZ 85718

Resident SST!

7-53

SCROLL SYMBOLIC TRACER

An 8086 Family Symbolic Debugger/Interpreter

by
Murray Sargent III
Lillard Darwin Sanders III

8087 display/modify facility by
Matt Derstine

Scroll Systems, Inc., 5530 N. Camino Escuela, Tucson, AZ 85718

Version 2.0
January 14, 1988

This manual is a combination of the SST 1.4 manual and four new chapters, Chaps. 11 through 14. As such references to the special features of SST 2.0 are not found in Chaps. 1 through 10. A subsequent manual will integrate these features.

Printed in the United States of America

This manual was prepared and printed using the Scroll Systems *PS™ Technical Word Processor* on an Imprint Technologies *LightWriter* laser printer.

Chapter 1
INTRODUCTION

If you work with or want to learn assembly language, SST™ 2.0 is for you. Similarly if you need to debug programs written in assembly language and in higher-level languages, SST can be invaluable. SST is a screen-oriented, upward-compatible replacement for the ubiquitous DEBUG.COM distributed with DOS. Use SST like DEBUG and enjoy access to a relaxed syntax, numerous extensions, ready help information (just type Function Key 1), and instantaneous full screen displays. In particular, the trace and display functions are much more powerful than DEBUG's. SST is to DEBUG much as a word processor like WORDPERFECT is to EDLIN. In addition to assembly language and source-level support for programming under MSDOS, SST 2.0 includes special support for Microsoft *Windows* (see Chap. 13) and the Intel 80386 microprocessor (see Chap. 12).

On an 80386 microprocessor system, SST can run in protected virtual address mode outside the usual DOS space (the first megabyte of RAM). In this mode SST has power beyond all but the most expensive hardware debuggers. For example, you can break on any input/output operation to any port or on read/write/execute access of any 4-kilobyte page of memory. Furthermore your program cannot corrupt SST's memory or code since SST lives in "hyperspace" above the first megabyte of RAM! This incredibly powerful mode of operation requires the SST DOS Extender Option.

SST, The Integrated Debugger

SST is an integrated debugger that combines RAM, disk, screen-font, and code display facilities with syntax is used for all modes, making them easier to use than a set of unrelated programs. SST also incorporates an assembly language interpreter that allows you to write and debug COM files much as you create BASIC files using a BASIC interpreter. The COM files so generated can run stand alone or under SST's supervision and run at full machine speed, unlike other interpreter code. Type a or A to run the Auto demo of the Function Key 7 to see how the interpreter and other features work and consult Chap. 8 for further information about the interpreter.

In addition, SST 2.0 can single step source code written for example in the C programming language. At any point you can work directly in the high-level language, display mixed high-level code and the corresponding assembly mnemonics, or display the mnemonics alone. See Chap. 11 for more information on source-level debugging.

SST runs in both resident and nonresident modes. If you can afford the extra RAM, we recommend having a resident copy running to give you instant access to the built-in calculators, system extensions, ready debugging, and trapping of errors such as divide overflow. Enter SST at any time by typing Ctrl-Enter or pressing an NMI button. Nonresident use is valuable for debugging programs and running the interpreter.

This manual is primarily a reference to SST and does not have much tutorial material. We recommend you take a guided tour through many of SST features by running the Auto demo offered when you type Function Key 7 in the SST COMMAND MODE. You might also find one or more of the many books now available on 8086/8088 assembly language helpful. In particular, the book *The IBM Personal Computer from the Inside Out* by Sargent and Shoemaker (Addison-Wesley) contains several chapters on assembly language. The "Sample Program" section of Chap. 2 helps to explain how to load and trace a program using labels.

This first chapter introduces many of SST's features and explains how to use this manual. Subsequent chapters explain the features in greater detail.

Full Screen Display Mode

The *display command* displays a delimited area of memory if both the start and end addresses are given. However if neither or only one address is specified, an instantaneous full screenful of memory is displayed. This screen can have the usual hex/ASCII format or a pure ASCII format. The cursor arrows, PgUp, PgDn, space bar, and backspace move the cursor around. It is possible to scroll rapidly (four seconds per 64K in ASCII mode) through all of memory scanning for text. A variety of hot keys allow you to use the information at the cursor as pointers to move around memory and to define blocks on which to operate. For a demonstration, run SST, type Function Key 7 followed by `d` or `D`. Type Function Key 1 for context-sensitive help on the display mode. More complete discussion is given in Chap. 7.

Trace Mode

The *trace command* allows streamlined screen-oriented execution of programs in single step or under control of breakpoints. Single-stroke hot keys are used to advance execution. The current instruction (at cs:ip) is highlighted by a reverse video bar. Whenever execution goes outside the instructions displayed, the screen is instantaneously redrawn with the appropriate new instructions. A conditional jump or loop that will be successful is identified by an arrow pointing in the direction of the jump and the target offset is displayed in boldface if it's on the screen. A small display window can display a selected portion of memory (arrow keys, PgUp, and PgDn can scroll this window, and Ctrl-U and Ctrl-D change its size), or it can track the memory locations referenced during the trace. A program stack window displays RAM starting at the top of the stack (given by the register pair ss:sp) and identifies the stack words by one of three readout offsets.

Function Key 5 zooms the stack and display windows into DISPLAY MODE, where you can overtype their RAM. Similarly the Edit hot key lets you overtype the register values. Function Key 6 moves the cursor from one window to another, allowing you to scroll the display, stack, program output, and trace windows. When in the trace window, the cursor is used for setting breakpoints, starting assemblies, and moving the instruction pointer. When the cursor is in the program-output window, Ctrl-U moves the window height Up, while Ctrl-D moves it down. When the cursor is in any other window, Ctrl-U and Ctrl-D move the memory-examine window height Up and Down, respectively.

In continuous trace mode, the tracking-memory mode produces an impressive dynamic screen display that often reveals how a program works. The multiple-step Undo option is particularly valuable. This allows you in effect to execute backwards, discovering why registers or memory locations got their values, or how you got to the current instruction. With it you can single-step into a subroutine, change your mind retracing backwards, execute the subroutine at full speed, and continue single stepping afterwards. For an example of how to trace your own programs, see the "Sample Program" section of Chap. 2. Chapters 7 and 14 describe the TRACE MODE further.

Trace Mode Demonstration

For a demonstration, run SST, type Function Key 7 followed by `t` or `T`. This starts continuously tracing a built-in piece of code. You can pause execution by typing the space bar. Subsequent space bars single step execution; i.e., advance execution one instruction at a time. Other single-stroke commands include those to single-step, break at the current instruction, break at the current instruction after executing it a specified number of times, fast execution (e.g., call or loop) at full machine speed, slow execution which allows single-stepping **int** calls (normally executed in Fast mode), and no execution useful for skipping unwanted instructions. A program can be traced continously as in the T demo with full screen updates, or run three times faster in a Quiet mode that only updates the register values. These continuous trace modes are interrupted by typing a character or by a reference to a memory location protected by the p command. The continuous mode is also interrupted by an illegal op code or by an op code belonging to a higher-level microprocessor (e.g., an 80286 instruction executed on an 80186 or 8088). Type Function Key 1 to see a help screen defining the TRACE MODE hot keys.

Back Tracing

SST also lets you backtrace program execution up to twenty steps by default. This can be very handy when you find the program somewhere and can't figure out how it got there. Just type u or U for Undo and watch your program execute backwards in time. As for Super-Trace, this has to be seen to be believed! To change the number of backtrace steps use the /U*n* switch when starting SST (see Command Line Parameters in Chap. 3). Note that currently the backtrace (undo) feature restores only the 80286 subset of the 80386 machine state and it cannot undo values output to an I/O device.

8087/80186/80286/80386 Support

The *assemble and unassemble commands* recognize all 8086/8087 mnemonics. SST supports the 80186, 80286, and 80386 extensions as well. The *search command* can search for assembly language instructions as well as hex bytes and string literals. See Chap. 12 for discussion of additional 80386 support.

SST fully supports the 8087 numeric coprocessor with stack displays in TRACE MODE. Registers and memory can be changed by simple assignment statements using ordinary scientific notation, and all status information is displayed. An 8087 floating point calculator is also built in as described in Chap. 5.

Labels

SST supports the full link MAP for the DOS LINK.EXE linker. By reading in the map (specify /MAP option on the LINK.EXE list file entry), you can refer to program line numbers and external labels (declared EXTRN in .ASM files and external names in general declared in compiler source files). See the load label option in Chap. 7. You can also read in variable names for one segment. Variable names are not treated as generally as desired, partly due to the need for telling SST what segment should be assumed for variable references. Something like the MASM.EXE ASSUME directive is needed. Improvements along these line are planned.

Calculators

Both a Polish suffix hex calculator and a floating-point calculator are included. String literals and decimal values (indicated by a decimal point) are supported in the HEX CALCULATOR, SEARCH, and ASSEMBLER MODEs. Register variables can be used in calculator expressions, and register and flags can be assigned values by direct assignment.

The floating point calculator requires an 8087 numeric coprocessor to be installed. It provides 80-bit trigonometric, exponential, hyperbolic, and arithmetic functions. The results can be inserted into the keyboard input queue to obviate the need to retype them perhaps with errors.

Command editing supports the Function Key 3 DOS Edit function (repeat to end of last command), although DOS is bypassed for all operations other than disk. This allows most DOS functions to be traced and leads to much faster response (up to 100 times faster than DEBUG!). In addition, the left and right arrow keys, Home, End, Del, and several Ctrl keys can be used for editing.

Synergy

The program serves both to teach people new to assembly language how the machine works, and to aid the advanced programmer in finding program bugs. It typically requires 1/20th the time to find a bug with SST as compared to DEBUG or SYMDEB, and sometimes a few minutes with SST can literally save you days of debugging with DEBUG. The program runs in about 100K RAM and is written in optimized 8086 assembly language (some 80286/80386 code is used in special sections off limits to smaller microprocessors). It has many features not found in debuggers requiring two or more times as much memory. SST's relatively small size per feature is due partly to tight coding, and partly to a careful integration of facilities that allows the various components to take advantage of one another (synergy!).

Menu Line

Initially SST operates in COMMAND MODE. Depending on the command chosen, SST may switch into one of several other modes, namely ASSEMBLE MODE, ASSEMBLE INSERT MODE, UNASSEMBLE MODE, TRACE MODE, XAMINE MODE, DISPLAY MODE, OVERTYPE MODE, CALCULATOR MODE, DISK DISPLAY/OVERTYPE MODEs, FONT DISPLAY/OVERTYPE MODEs. A ↵ (Enter) returns to COMMAND MODE (two ↵'s return in ASSEMBLE MODE).

The current mode is displayed on the *menu line* at the top of the screen. For example, in COMMAND MODE, the menu line reads

```
COMMAND MODE:  F1 0-9 Asm Cmp Dsp Exam Fill...
```

The menu line is also used to report some errors and special conditions.

Register Window

A register window is usually displayed at the top of the screen just below the menu line. In COMMAND MODE, the command `rØ` toggles this window on and off, and in TRACE MODE, the hot key 0 turns it on and off. Other windows appear for various commands, and pop-up help screens always appear immediately below the register window. This special window has the form:

```
ax=n bx=n   ds=n es=n cs=n ss=n   bp=n        ds:[n]=n
dx=n cx=n   si=n di=n ip=n sp=n   NS NZ NC    PE + EI
```

The *n*'s in this figure represent 4-digit hexadecimal numbers. The third line in general displays the menu for whatever mode is currently active. Here the COMMAND MODE menu is showed in part. This particular menu is displayed when SST is started and whenever you type a ↵(double ↵'s in ASSEMBLE MODE).

The register window groups the registers according to their typical usage in 8086 code. The **ax**, **bx**, **cx**, and **dx** registers are the general accumulators that can also be split into pairs of 8-bit registers like **ah** and **al**. The segment registers **ds**, **es**, **cs**, and **ss** are shown directly above the 16-bit registers with which they are commonly paired. Specifically, the addresses **ds**:[**si**] and **es**:[**di**] are used with the powerful 8086 string instructions. The **cs**:[**ip**] address gives the current instruction, and the **ss**:[**sp**] address gives the top of the program stack. In addition, **ds**:[**bx**] and **ss**:[**bp**] are common addresses, so it is handy to have the corresponding registers near one another.

On the top line the ds:[*n*]=*n*, which appears only when a memory reference occurs and displays the value and address of such a reference. If the value is a byte value, only two hexadecimal digits are displayed.

After the sp=*n* field on the second line, the flag values are displayed. For example, if the Zero flag is set to 1, you see "Z". If it is reset to 0, you see "NZ" as shown in the figure. This notation corresponds to the instruction mnemonics used by the unassemble and trace commands. Note that since the TRACE MODE reverse video bar for the current instruction indicates whether a conditional jump will occur, it isn't nearly as important to consult the flags as it is with DEBUG.COM. PE means that the last instruction that affects the parity flag found Even Parity, while PO stands for Odd Parity. A + or - indicates the direction in memory that repeated string operations go. The instruction CLD (CLear Direction) gives a plus sign (+), which is the usual direction for most programs. If Interrupts are Enabled, you then see EI, while if they are Disabled, you see DI. The two remaining flags, OV (Overflow Flag) and AC (Alternate Carry) occur less often and are only displayed if they are on. This choice helps to reduce screen clutter and separates the principle set of flags (Sign, Carry, and Zero) from the others.

Following the Interrupt flag value, four SST status values are displayed in reverse video if their corresponding functions are enabled. These are **E** for active Echo output (see `n>` command), **S** for active Super-Trace conditions (see trace command), **T** for Tracking memory display window (see trace command), and **V** for 80286 protected Virtual address mode (see `vm` command).

80386 Register Window

By default on 80386-based computers, the register window displays the complete 32-bit 80386 register values in the form

```
eax=n ebx=n   ds=n ss=n   ebp=n   ds:[n]=n
edx=n ecx=n   fs=n gs=n   esp=n   NS NZ NC
esi=n edi=n   es=n cs=n   eip=n   PE + EI
```

In COMMAND MODE, the `r3` command and `r1` switch to the 80386 and 8086 register sets, respectively, while in TRACE MODE, the hot keys 1 and 3 perform these switches. The `dr` command displays special 80386 registers (see Chap. 12).

Disk and RamFont Editors

SST contains a disk editor invoked by the command **disk** in COMMAND MODE. The idea is that in place of the segment specification for RAM, you type a sector number. The facility, described further in Chap. 9, has a variety of options to facilitate moving around a disk.

SST has a RamFont editor that allows you to create and modify the characters sets that appear on your computer display. This facility requires the use of the Hercules Graphics Card Plus or the IBM Enhanced Graphics Adapter, or other boards compatible with one of these. See Chap. 10 for further discussion.

Mouse Support

The RamFont editor and the DISPLAY and TRACE MODEs can use the mouse to move the cursor around. To enable the mouse, you have to run the appropriate MOUSE program at the DOS command level, and then tell SST that it should use the mouse by typing the **mouse** command in SST's COMMAND MODE. The mouse allows you to move around the display screens rapidly and to edit character fonts.

Super-Trace

SST has a pair of exceedingly powerful conditional break facilities for advanced users. The first is the Super-Trace mode, which traces program execution at about one tenth full speed and after each instruction it checks an arbitrary set of conditions specified by the user in assembly language. If the result of these conditions sets the Zero flag, tracing is halted; otherwise the trace continues. This allows a very rapid execution search for any desired machine state. It implements in software features that have been hitherto performed only by expensive hardware tracing boards, and has generality that the hardware methods cannot match. The user conditions can even call user-supplied subroutines, allowing specialized monitoring such as program execution profiling. The use of ordinary assembly language for the user conditions combines the highest execution speed, the simplest implementation and documentation, and the greatest power available in the computer. Because of the great flexibility of the method, you have to be careful not to include a command that will crash the computer. Hence we consider the Super-Trace to be a facility for advanced users, although simple Super-Traces can be run by beginners (see Chap. 3 demonstrations).

Conditional Breakpoints

Alternatively, breakpoints can be associated with the same arbitrary set of conditions as the Super-Trace. For these, execution proceeds at full speed until the computer attempts to execute the instruction at one of the user-defined breakpoint locations. The user's conditions are then checked. If they succeed in setting the Zero flag, program execution is halted and control is returned to SST. Otherwise execution proceeds again at full speed. If no breakpoint is encountered, you can usually recover control by typing Ctrl-Enter.

How to Use this Manual

This manual tells you how to use SST. It should be used in combination with a book or reference manual on assembly language for the Intel 8088/8086 microprocessor. The book *The IBM Personal Computer from the Inside Out* by Murray Sargent III (SST author) and Richard L. Shoemaker (Addison-Wesley Publishing Co., 2nd Edition, 1986) is one of several such books. If you are already familiar with assembly language and DEBUG.COM, you may just want to glance at this introductory section, at Chap. 4 on Syntax, and then refer to the command descriptions of Chap. 7 when the built-in help messages are too terse. If you are learning assembly language, read the Help section (Chap. 2), the Demonstration section (Chap. 3), run the Auto demo of Function Key 7, read the Syntax section (Chap. 4), and read your book on assembly language. Try out the built-in demonstrations to get a feel for how memory looks and how a program runs. Assemble some simple code of your own and trace its execution with the trace command. You'll learn assembly language in a fraction the time required by traditional methods.

Chapter 2
HELP FACILITY

In all modes, typing Function Key 1 displays an appropriate help screen in a pop-up window just below the register window. Typing any key (except for Function Key 1 itself in ASSEMBLE MODE) replaces the screen text that was covered up by the help window. In particular, typing Function Key 1 instead of a command displays the menu

```
@scii    Asm      Baud   Comp   Display  Exam   Fill
Go       Hex      In     Klear  Load     Move   Name
Out      Protect  Quit   Reg    Search   Trace  Unasm
Vector   Write    Xam    YGDT   Zam

For more help, type command letter followed by Function
Key 1

0-9 '"  start calc entries        exp = value | exp1 exp2 op
address = [segment:] offset       range = address1 address2
```

This gives the names of the simple commands available under SST. To run a command type the first letter of the command name followed by appropriate arguments. If you type Function Key 1 in the middle of typing a command, a terse help message for that command is displayed in a pop-up window below the register window. Typing any character gets rid of the help window, replacing the text it covered up. If you type the command character immediately followed by the Function Key 1, then the command line is erased when the help window goes away. If you type more characters on the line, the command line remains, ready for further typing. This allows you to get help whenever you need it. For more information on each command, please see the corresponding page in Sec. 7, which is ordered alphabetically by command.

The Function Key 1 help summaries are (for the complete a F1 displays, see assemble command below):

Brief Command Definitions

`and` *range list*	And (bitwise) memory in *range* with *list*
`a` [*address*]	Assemble. Op codes (186/286 capitalized, F1 → 8087) are:
`a` [*address*]	Assemble. 8087 op codes (F1 → 8086) are:
`b` *rate* [,*channel*]	Set the baud rate of serial channel
`b`*c* *list*| *	*c* breakpoints in *list*, where *c* = `c`, `d`, `e`, for Clear, Disable, Enable
`bl`	List breakpoints
`bs`[*n*] *address* [*m*]	Set breakpoint [*n*] at *address* [skip *m* passes] Note: `b` breakpoints are sticky unlike g's
cd *path*	Change Directory to *path*
`cls`	Clear Screen
`close`	Close all files
`c` *range address*	Compare memory in *range* to memory at *address*
`cpu`	Display info about computer
`date`	Display date
`dir` [*template*]	Display filenames matching *template*
`d` [*address*]	Display full screen of memory
`d` $address_1$ $address_2$	Display memory from $address_1$ to $address_2$ Can echo to file - see `n>`
`d/`*c*	display labels (*c*=segment paragraph), variable names (*c*=v), or user strings (*c*=u)
`dos`*n*	Execute DOS (int-21h) function ah=*n*
`echo`	Toggle screen echo on/off
`erase` *filename*	Erase file *filename*

`e` *address*	Examine *address* in Byte mode
`e` *address* [/*type*]	Examine *address* by *type* = `b`, `d`, `i`, `l`, `o`, `q`, `s`, `t`, for packed BCD, Double float, Integer, Long int, Quad int, O binary, Single float, Temp float, respectively (Needs 8087)
`e` *address* [/*string*]	Examine with structure template *string*
`f` *range list*	Fill memory in *range* with *list*
`g` [=*address*] [$address_1$]...	Go execute at **cs:ip** or =*address* with breakpoints at $address_1$, $address_2$, ... Same followed by @ allows conditions to be typed
`h` *value*	Convert **hex** *value* to binary
`h` $value_1$ $value_2$	Calculate $value_1+value_2$ and $value_1-value_2$
`ini`	Run first sst.ini file in DOS path
`i` *portaddress*	Input byte from *portaddress*
`int21`	[*n*] Display int-21 function definition(s)
`iv`	Initialize MDS Genius display
`j`	(No command)
`k`	Klear screen
`k`*n*	Klear next *n* lines
`k`*n m*	Klear from line *n* to line *m*
`kf`	Klear floating point (8087) registers
`ki` ↵ *c*	Display keyboard input code *c*
`list` [*address*]	Unassemble screenful starting at *address*
`llist` [*address*]	Unassemble to printer starting at *address*
`l` [*address*]	Load file named by `n` at cs:100 or *address*
`l` *address drive* $sector_1$ $sector_2$	Load absolute sectors $sector_1$ thru $sector_2$ at address *address*
`ll`	Load program labels (use `n` to name .MAP file)
`lm`	Load program labels for a .COM file
`lv`	Load variable names (name .LST file)
`m` *range address*	Move memory in *range* to *address*
`n` *filespec*	Name load or write file by *filespec*
`n>` *filespec*	Name echo file by *filespec*
`n>`	Toggle echo to file
`n=`	Display current name file

n *string* = "..."	Define user *string* (2 letter names)
new	Reset labels and paramters to starting values
not *range*	N (bitwise) memory in *range*
or *range list*	Or memory in *range* with *list*
o *portaddress list*	Output *list* to *portaddress*
pause *n*	Pause a time proportional to *n*
prompt	Toggle path prompt
p	Turn off memory protection
p *address*	Protect memory *address*, i.e., Stop trace if memory *address* is referenced
p *range*	Stop trace if memory *range* is referenced
q	Quit - return to DOS
q*c n*	Set screen attribute for window *c* = a, h, n, r, s, x, z for Assemble, Help, Normal, Register, Stack, Xam, Zam, respectively
qg*n*	Set SST video RAM segment = *n*
qi*n*	Set SST interrupt mask = *n*
ql*n*	Set lines/page = *n*
qo*n*	Set SST display origin to line *n*
qol	Display in lower half of 66 line screen
qp*n*	Set 6845 CRT I/O port = *n*
qs	Swap IBM displays for SST alone
qs1	Swap displays for both SST and DOS
qs3 (2)	Turn screen save on (off)
qu*n*	Set undercover debugger port = *n*
qy*n*	Set # lines Xam window = *n*
r [*register*]	Display [change] registers. Can change registers by =
rr	Restore registers to initial values
rm	Go to real *address* mode
rn	Return NMI interrupt to preceeding owner
ren *file*$_1$ *file*$_2$	Rename *file*$_1$ *file*$_2$
s *range list*	Search memory in *range* for *list*
s *range* @	Search *range* for assembly language
s *range*	Search *range* for last string
s	Repeat last search
system	Quit to DOS
time	Display time
t [*address*]	Trace program starting at cs:ip or *address*

t @	Specify Super-Trace conditions
type *filename*	Type (browse) file *filename*
u [*address*]	Unassemble code at last address or at *address*
u *range*	Unassemble code in *range*. Use **n>** to echo to file
n [**ax** [**bx** [**cx** [**dx**]]]]	Execute interrupt vector *n* giving optional register values
vm	Switch to 80286 protected virtual address mode (AT only)
width*n*	Set screen width (40 or 80)
w [*address*]	Write file named by n from cs:100 or *address*
w *address drive sector*$_1$ *sector*$_2$	Write absolute sectors *sector*$_1$ thru *sector*$_2$ from *address*
wl	Write labels
x *address*	Start trace examine window at *address*
xor *range list*	Xor memory in *range* with *list*
y	List 80286 GDT entries, one/space bar
y n	List GDT entry *n*
y *n address* [*access* [*length*]]	Define GDT entry 68<*n*<D8 at *address*, access = *access*, length = *length*
z	ezamine 8087 status

For definitions of command syntax and words like address, see Chap. 4 on Syntax. Typing \ followed by Function Key 1 displays the current disk drive:directory (to obtain this information continuously in COMMAND MODE, type the **prompt** command). Typing a decimal digit followed by Function Key 1 displays help for the calculator (see Chap. 5 for more information):

Hex number	Convert to decimal
decimal number	Convert to hex
exp$_1$ *exp*$_2$ *op*	Calculate *exp*$_1$ *op* *exp*$_2$ (*op* = +-*/&!)

ASCII/EBCDIC Chart

When debugging it is often very handy to have ready access to the ASCII codes. These are usually instantly available in a pop up screen by typing @. In some situations the @ would be used for other purposes, such as in an assembly language comment, searching for assembly language, and supertracing. Hence @ doesn't give the pop up menu as the second or later character of the command line. It also works in most non COMMAND MODEs, and shows you a hexadecimal display of all 256 extended ASCII codes. Type any key other than another @ and you're back to the screen displayed before you typed the @.

The @ option has four pop-up screens. To get to the next one, type @. The pop-up screen following the initial hexadecimal ASCII screen is a decimal ASCII display. The third screen is an EBCDIC (Extended Binary Coded Decimal Interchange Code used on IBM mainframes) display with hexadecimal codes, and the fourth screen is an EBCDIC with decimal codes. Further @'s repeat this sequence of four screens.

In RamFont modes (see Chap. 10), you can see the ASCII and EBCDIC charts displayed with different fonts by typing the desired font number.

Sample Program

To illustrate the loading and tracing of a program, the SST distribution diskette includes a simple program to get and display console input. The program is as follows:

```
public  ci,co,console_loop

;Simple  console  echo  program  that  illus-
trates
;SST label facility

CR       = 13
LF       = 10

cseg     segment
         assume            cs:cseg

console_loop:
         call   ci          ;Get  next  charac-
                            ter
         mov    dl,al       ; from console
         call   co          ;Display character
         cmp    dl,CR
         jnz    console_loop
         mov    dl,LF       ;If CR, output
         call   co          ; LF automatically
         jmp    console_loop

ci:      mov    ah,7        ;21h        direct
                            console
         int    21h         ;   input   without
                            echo
         ret

co:      mov    ah,2        ;21h       display
                            output
         int    21h
         ret

cseg     ends
         end    console_loop
```

You can run the sample CONSOLE program either by typing it in in ASSEMBLE MODE (see assemble command in Chap. 7), or by assembling and linking it with the Microsoft assembler. For the latter you need to have Microsoft's MASM.EXE available in your current directory or in some subdirectory specified by the **path** command in your AUTOEXEC.BAT file. If you're using DOS 2.0 or later and don't know about the **path** command, immediately go read about it in the DOS manual, since it can simplify your life considerably.

Suppose for the sake of illustration that the DOS prompt is C:\\> and SST's prompt is ▸. Then at the DOS prompt type

```
C:\>masm console;
C:\>link console,,console/map;
C:\>sst
▸nconsole.map
▸ll
▸nconsole.exe
▸l
▸t
```

This puts you into the SST TRACE MODE all ready to trace your simple console program. SST allows you to see what you program displays on the screen in several ways. For the present case, just type the TRACE MODE **W** option, to give yourself a DOS window on screen. Then start single stepping your way through the program by typing the space bar. When you reach the **int 21h** for the `ci` subprogram, the console pauses to let you type in a character. Type something other than the space bar, so that the `co` routine will display something you can see in the DOS window. Notice that the ASCII code of the character you type for the `ci` subroutine is returned in the **al** register (low byte of the **ax** register). The program then moves this character into the **dl** register. You can watch this action by looking at the register window at the top of the SST display screen.

After single stepping for awhile try some of the SST options like **D** for Don't single-step subroutine, **B** for Break when back at the current instruction, and **G** for break (Go) at the address you type in. Working with this simple program can teach you a great deal about the TRACE MODE. Return to COMMAND MODE at any time by typing the Enter key or the Esc key.

For simple programs like this one, the DOS window is fine, but for more typical programs, the whole screen is needed. If you have two screens on your computer, you can put SST on the one your program isn't using (see Sec. "Multiple Screens" in Chap. 3). Alternatively you can turn on the screen save option discussed in Sec. "Screen Save Option" in Chap. 3. For this, just type `qs3` at SST's COMMAND MODE prompt, and return to TRACE MODE. In single stepping most instructions, you'll notice no difference with the screen save option enabled, but whenever you do something that SST cannot know whether the screen will be accessed (e.g., you use an explicit or implied breakpoint), you'll notice a momentary flashing of the screen. This is because SST restores the entire screen for the user program.

Any time you want to switch to the user screen, type `v` or `V` for View program screen in TRACE MODE. To return to SST's screen type any key.

INT21 Command

SST automatically comments some unassembled instructions, such as DOS calls (**int 21h**), 8087 emulation interrupts (**int 34h - int 3dh**), and immediate byte constant instructions like `mov al,41H`. In addition the **int 21h** definitions are displayed when you type the **int21** [*n*] command in COMMAND MODE. If the optional *n* is present, the definition for that entry point alone is displayed. If *n* is missing the next hexadecade of **int 21** entries is displayed. These features are very handy for working with code that makes DOS calls.

Help Command

For more information, type `help` in COMMAND MODE. This displays the file called SST.HLP, which has a variety of help imformation.

Notes:

Notes:

Chapter 3
RUNNING SST

The first thing to do with your SST is to see it in action! Run SST and you'll see the sign on help message in the main part of the screen and the COMMAND MODE window at the top. The 8086 registers are displayed in this window followed by the COMMAND MODE menu. Type Function Key 7, type `t` or `T` for the Trace demo, and stare at the continuous TRACE MODE in amazement! What you'll see is a dynamic screen trace of the execution of a program, revealing how the registers, stack, and memory referenced by the program change. In this continuous trace mode, the program executes about 40,000 times more slowly than normal, which gives you a chance to see what's going on. For comparison with the Super-Trace described below, notice how the **di** register increments slowly (due to the **stosb** instruction) as the program runs.

Typically the program runs much too fast to understand what's going on, so to stop execution, type the space bar. Successive depressions of the space bar single-step the program, always showing you the latest state of the machine. You can see what effect the instructions have on the register, flag, and memory contents. The demo uses the "tracking display" window in ASCII mode, so that you always see an 80 hex byte memory window around the last memory location referenced by the program. The size of this window is programmable - see the `qy`*n* command in Chap. 7. At the right end of the second display line from the top, you'll see a **T**. This indicates that the memory window is in Tracking mode.

The TRACE MODE menu indicates many other options. Type Function Key 1 to see a help screen that gives brief definitions of most of these options. This help screen is also shown in Sec. 7 under the trace command, along with more detailed descriptions of the options. To get rid of the help screen, type any key.

Backtrace Demonstration

After you've traced program execution for awhile, type `u` or `U` for Undo. This causes the program to undo its steps, literally executing backwards in time. This feature is handy when you the program ends up somewhere and you don't remember how it got there. SST cannot trace backward for ever, or it would unboot your machine! Actually SST doesn't execute backwards, it just restores the preceding machine state for up to 20 back-states by default. To change this number, use the SST/U*n* option described under Command Line Parameters in this chapter.

Display Demonstration

After commands like assemble and load are executed, the Function Key 7 demo option is suppressed to prevent SST from overwriting a program you have loaded in or typed in with the assemble command. If Function Key 7 doesn't work, quit and rerun SST. Type Function Key 7 followed by `d` or `D` to go into the DISPLAY MODE. You can also do this at any time in COMMAND MODE by typing `d` or `D` followed by a ↵. This gives you a full screen display of memory with the register values and a menu at the top of the screen. Type Function Key 1 to see a help screen that gives a brief definition of the menu options. Section 7 under the display command also shows this help screen along with more detailed discussion of the options. Type any key to get rid of the help screen.

Overtype Mode

SST has a number of other options, including Ctrl-O, which toggles between OVERTYPE and DISPLAY MODE. This mode allows you to overtype the memory location at the cursor position. If you do this by mistake, type Ctrl-U to Undo the overtype. Hopefully you didn't overtype something important, like a keyboard interrupt vector (crash!). SST allows you to do absolutely anything with your computer, so be careful. SST isn't PASCAL, which usually prevents you from doing something you might later regret. This kind of freedom is desirable since it allows you not only to identify a program bug, but also to try out a possible fix without reassembling or recompiling and relinking. This can save you considerable time, but it does require a bit of care.

Try out the various options, scroll through all of memory, and learn about your machine as only hands-on interaction allows.

Command Line Parameters

When invoking SST from DOS, you typically type

```
C:\>sst filename other_parameters
```

SST then loads the file *filename*, and places the *other_parameters* in the command line area reserved in the program prefix, just as DOS's COMMAND.COM does.

In addition you can specify several useful options as switches following the SST. These options allow specifying the total amount of SST RAM work area, the number of back states for the TRACE MODE Undo option, and making SST resident. The Resident option is described in the next section.

```
C:\>sst/n
```

saves 1024**n* bytes of RAM for SST. A minimum of 9K is required. No specification results in 9 K bytes (/9). More RAM is automatically allocated as needed when labels are read in.

```
C:\>sst/un
```

saves room for *n* backsteps in TRACE MODE. The default for SST is 20.

Resident Operation

Sometimes it's handy to have SST in memory for ready access in the event of a problem, or just to see the ASCII chart or convert between hexadecimal and decimal. One way to do this is to follow SST on the DOS command line by the /R switch

```
C:\>sst/r
```

This loads SST and returns to the DOS prompt with SST resident. To have SST take control, type Ctrl-Enter or press an NMI button. This method doesn't let you have the chance to set up special SST features such as loading in program labels. If these other features are needed, try the q/R option described in Chap. 7.

Screen Save Option

When going between SST and a program, it's handy to be able to see the screen display generated by the program. The SST/R resident mode option described in the preceding section does this automatically. More generally to start up the screen save option, use the qs3 option described in Chap 7. Then whenever you return to a program using a go command, or use the View option in TRACE MODE, you can view the program screen rather than SST's.

Multiple Screens

For extensive debugging it's very useful to have more than one screen. This is particularly true when debugging graphics programs. SST can be put on whatever screen you desire using various q commands. In particular on the IBM PC, the monochrome and color/graphics adapters are so widespread that SST has been setup to switch very easily between the two. Type the command (▸ stands for the SST COMMAND MODE prompt)

```
▸ qs
```

to switch between them without telling DOS. Type qs1 instead to switch screens telling DOS as well. Hence you can load SST from DOS on either screen and switch back an forth as the need be and exit SST on either screen.

In addition, SST has special support for 66 line displays of the Micro Display Systems variety. See the q section of Chap. 7 for more details on these options.

Configuring SST

With the abundance of different screen sizes, character attributes, and other machine characteristics, it is time consuming to configure SST appropriately each time you run it. To help out, SST can be reconfigured using the q commands of Chap. 7 along with appropriate DOS commands. We illustrate this procedure using the screen attribute specification which allows setting the screen attribute (color, reverse video) for various SST windows. This command is defined by

```
▸ q*c n*
```

which choses a screen attribute *n* for the window *c* = a, h, n, r, s, x, z for Assemble, Help, Normal, Register, Stack, Xam, Zam, respectively. In particular, experimenting with the qr *n* (set register window attribute) is a great way to learn about screen attributes.

You can configure SST.EXE to your tastes by SSTing a copy of SST.EXE and typing the q commands followed by */n*. When done, type w or W to Write out the modified SST.EXE file. For example, on the IBM color/graphics display, to set up

> Red background, yellow foreground register window
> Blue background Xam window
> Green background stack window
> Red foreground assembly language
> Yellow foreground normal window
> Magenta help screen

type the following DOS and SST commands:

```
C:\>copy sst.exe newsst.exe
C:\>sst newsst.exe
▸ qr40/n
▸ qx10/n
▸ qs20/n
▸ qa4/n
▸ qn6/n
▸ qh5/n
▸ w
▸ q
```

Here the commands following the SST prompt ▸ are typed in SST's COMMAND MODE. The q's store the configuration information in your program NEWSST.EXE, the w or W writes the modified NEWSST.EXE to disk, and the final q command quits SST. Then type

```
C:\>newsst
```

Your file NEWSST.EXE now has these characteristics as you'll see by typing Function Key 7 followed by t or T to see the Trace demo.

Echoing Output to a File

You may want to save disassembled code or hex/ASCII formatted binary displays on disk files. With DOS 2.0 or later, such a file could also be the printer. For this purpose, some SST commands can echo what they send to the screen to a disk file of your choice. For explicit discussion, see the n command in Chap. 7.

Super-Trace Demonstration

With SST you can Super-Trace execution at one tenth full speed looking for a condition of your choice to occur. For example, type ↵ to return to COMMAND MODE, type `t@` ↵, which transfers control to the assembler for specifying Super-Trace break conditions, and type the assembly language instruction

```
cmp di,600
```

followed by two ↵'s, which turns on the Super-Trace mode. At the right end of the second display line, you'll see a S to indicate that Super-Trace conditions are in effect (they apply to breakpoints as well). Now things are very different. Each time you type the space bar, the trace stops only if the condition di=600 is satisfied! Hence the trace sometimes hesitates between single-stepping, since the computer has to execute many instructions in between. You can run the Super-Trace continously by typing `c` or `C`. To turn Super-Trace off, type ↵ to return to COMMAND MODE, type `t@` followed by two ↵'s, which turns the conditions off, and then type `t` or `T` ↵ to return to TRACE MODE. Notice that the S at the end of the second screen line goes away. Type `c` or `C` to start up the Continuous trace mode again, and notice how slowly the **di** register changes. This should convince you that the Super-Trace mode is really super! The Super-Trace executes about 10 times more slowly than normal, compared to the continuous trace, which executes 40,000 or more times more slowly than normal. DEBUG traces 270,000 times more slowly than normal and cannot be read when running continuously.

Another interesting supertrace is to have SST load your favorite word processor (see load command in Chap. 7) and supertrace for a condition that will never be met like `or sp,sp`. Since the stack pointer is never 0 for working programs, SST will simply run your program in slow motion. To stop, type Ctrl-Enter, or tell your program to quit. Super-Trace is described further in Sec. 7 under the trace command.

Terminating User Programs

Program terminations of all kinds (**int 20h**, **int 27h**, and **int 21h** with **ah**=0, 31h, and 4Ch) are intercepted by SST, which then gives a menu offering to 1) Restore registers to their initial values, 2) go into TRACE MODE, or 3) return to COMMAND MODE leaving the registers as they are. The restore option acts like DEBUG.COM, allowing you to rerun programs easily. The other two allow you to investigate the circumstances that led to program termination. You can also restore the registers at any time by typing `rr` in COMMAND MODE.

On terminating execution of a program, SST closes file handles 5 through 20, and releases all memory owned by the program's Program Segment Prefix.

SST initialization

The SST.INI file in the current directory is automatically executed when SST is loaded. This file can consist of a set of COMMAND-MODE commands that customize SST as you desire.

The COMMAND MODE **ini** command executes the first SST.INI file that it finds by doing a DOS path search.

Notes:

Chapter 4
SYNTAX

SST accepts the commands in the same format as DEBUG, so users of DEBUG can continue with their usual methods. In addtion, many DOS or BASIC like commands are supported and the two kinds of commands live together remarkably peacefully. The DEBUG-style alphabetic commands are identified by a single command letter that can be preceded or followed by optional blanks and tabs. Most command take one or more arguments separated by a blank, comma, or tab. The syntax has been relaxed in several ways to streamline command entry and execution. The semicolon (;) can be used in place of the colon (:) for specifying segment register values. As in DEBUG, the segment register names (cs, ds, es, and ss) can be used to specify address segments. SST allows the program registers (ax, **bx**, **cx**, **dx**, **al**, **ah**, **bl**, **bh**, **cl**, **ch**, **dl**, **dh**, **si**, **di**, **bp**, **sp**, and **ip**) to be used as well in place of hexadecimal values. Five-digit addresses refer to the entire one-megabyte address space, and 6-digit address correspond to extended RAM available on the IBM PC AT and compatible computers. If no segment register is given, ds is assumed for all commands except for assemble, go, load, trace, unassemble, and write commands, which assume cs.

Syntax Type Fonts

In this document we indicate characters typed by the user using the `courier` font, which resembles the standard screen characters and we display output from the program in ordinary print. Each command is defined and described with examples, and is compared to its DEBUG version. The Enter key is indicated by ↵, and is used to terminate a command line, and to terminate the execution of certain commands like the DISPLAY and TRACE MODEs. The usual SST prompt character is shown as ▸ (similar to the SST prompt on the IBM PC screens.) Variables are given in *italics*.

Each command is introduced with a SYNTAX specification. In these specifications, square brackets [] are used to surround optional fields. For example,

▸ `a` [*address*]

means that the letter `a` is typed following the SST prompt ▸, optionally followed by the address *address*. The text following the syntax specification then defines what `a` alone means and what `a` followed by an address means.

In the syntax specifications, the word *range* stands for two addresses separated by a comma or blank. The first address can have an optional segment specification. The segment used for this first address is automatically used for the second (the second address for the protect command can have its own segment). The range can be used in commands like display, fill, and compare. For example,

▸ `d` [*range*]

displays the range of memory *range*. This range can be given in one of three forms:

1. $address_1$ $address_2$
2. $address_1$ l *count*
3. Ctrl-B

The first form specifies the address explicitly either with hexadecimal values of the form [*segment*:] *offset*, or with labels (see Label section in this Chapter). The second form specifies the number of bytes including the first one pointed to by the address *address*$_1$. The third method uses a block defined by the Ctrl-T and Ctrl-E DISPLAY MODE options (see Block section in this chapter).

If during execution you type Function Key 1 anywhere after the command letter, the short syntax definition is displayed in a pop-up window under the register window. Type any character to get rid of this help screen. If you type Function Key 1 immediately after the command letter, both are erased when the help screen goes away. If you have typed more than two characters when you type Function Key 1, then those characters are left, letting you continue your command after seeing the help screen. The assemble command works a bit differently in having two help screens. The first time you type Function Key 1 a terse syntax definition is displayed, followed by the full set of possible 8086/80186/80286/80386 instruction mnemonics. Typing Function Key 1 again toggles the Assemble help screen to the 8087 mnemonics. Typing any other character replaces the screen text and cursor to that present before you asked for help. This lets you check the spelling of a mnemonic in the middle of typing an instruction.

Specifying an Argument Using Cursor Arrow Keys

If the desired hexadecimal value for an argument (usually an address) appears on the screen due to displaying or unassembling, you can insert this value into your command line. Use the cursor arrow keys to move the cursor to the start of the value and then type a blank to continue the command or a ↵ to run it. Both the address at the cursor as well as the character you type are inserted into the command string. This is particularly useful for beginning a trace at a disassembled instruction displayed on the screen, for setting a breakpoint at a disassembled instruction, or for starting a disassembly at an address found by searching for instruction mnemonics. For example, to start execution at the current instruction (cs:ip) and break at an address displayed on screen, type

```
▸ g
```

with no ↵, move the cursor to the desired address on screen, and type ↵. If you want a second breakpoint, press the space bar instead of the ↵, move the cursor to the second address, and then type the ↵. If you just want to copy in the segment value from one part of the screen, move the cursor to the start of the segment value and type a backspace. This inserts the segment, the colon and only three digits of the address offset, three more backspaces delete the rest of the offset to make room for the value you want. If you're an accurate typist, inserting a segment value this way isn't that useful, but it illustrates how whatever you insert this way can be further edited, unless you type the terminating ↵.

You can use the arrow-key insertion method to assemble on top of code, or to set a single temporary breakpoint. You can also do these things directly from the UNASSEMBLE and TRACE MODEs.

DOS Commands

A subset of DOS commands has been built into SST allowing you to change the default drive and directory path, to display directory filenames, to type files, and to erase files. The display runs about three times as fast as DOS, and the **type** command at least 20 times as fast. Hence the **type** command runs in a special paged mode.

To change the current DOS drive letter, type the desired drive letter followed by the command colon (or semicolon). For example,

```
▸ a:
```

switches to drive A and displays a screen showing the current drive (A) and directory path on that drive.

To change the default directory path on the current drive type `cd\` followed by the path name. Hence

```
▸ cd\sst
```

changes to the subdirectory \SST, and displays a screen to that effect. If no such directory exists, the screen shows the active directory instead.

To display the current default drive and subdirectory, type \ Function Key 1 in COMMAND MODE or type the **prompt** command in COMMAND MODE to continually see this information.

To display DIRectory information, type **dir** followed by the desired filename template. The total number of bytes in the files matched is displayed in decimal for all values if you have an 8087, and up to 64K if you don't (larger values are then displayed in hex).

To display a file, type

```
▸ type filename
```

which allows you to browse up and down a file with search capability.

To erase a file, type

```
▸ erase filename
```

You will be asked to confirm before SST erases the file. The "ase" in "erase" is optional.

To leave SST, you can type **bye**, **quit**, or **system**, in upper or lower case. For a complete summary of these commands, see "Interpreter Commands" in Chap. 8.

Editing Command Lines

SST has a line edit facility patterned after the PMATE editor that works both in COMMAND and ASSEMBLE MODEs. While typing in a command or assembly language statement, you can use the left and right arrow keys to move around the line. Typing ordinary characters simply inserts them at the cursor position. Keystrokes are identified as follows:

Home	move cursor to beginning of line
End	move cursor to end of line
Ctrl-O	move one word left (Ctrl-← is an alias)
Ctrl-P	move one word right (Ctrl-→ is an alias)
Ctrl-Q	delete word to left
Ctrl-W	delete word to right
Ctrl-K	delete (Kill) from cursor to end of line
Del	deletes character under cursor
Backspace	deletes character before cursor

Modifying Edit Command Characters

If you prefer, you can load in a file to reconfigure the Ctrl character commands. The distribution diskette includes such a file called WSKEY.INI, which changes the command to correspond to MicroPro's WordStar editor. This file contains the single command

```
key 1e01, 1f13, 2004, 2106, 0, 2308, 2207, 1414, 1519
```

Each entry specifies the desired IBM PC character code for an edit function. These functions appear according to the order:

LeftWord, LeftChar, RightChar, RightWord
DeleteLeftWord, DeleteLeftChar, DeleteCursorChar
DeleteRightWord, DeleteToEndOfLine

Hence in the WordStar example above, the first entry 1e01 (Ctrl-A) corresponds to moving left one word, while the second entry 1f13 (Ctrl-S) corresponds to moving left one character.

Labels

The load map generated by the DOS LINK.EXE program with the /MAP list option can be loaded by SST to identify locations in memory by name. This works with large and small memory model programs and greatly facilitates debugging programs. Once defined, the labels can be used in place of hexadecimal addresses. For example you can type

```
▸u alpha
```

to start unassembling at the double-word address `alpha`.

To load in program labels, name the .MAP file with the name command (Chap. 7), and type the `ll` command (see Chap. 7 load command). This automatically reads the labels in starting at the point in the .MAP file identified by the words "by Value" and relocates them relative to the origin of the .EXE module (program prefix segment paragraph + 10).

To load .MAP labels relative to some other paragraph, type `ll`*n*, where *n* is the desired paragraph number. This option is useful for debugging resident programs.

To load .MAP files for use with .COM files, type `lm`, which relocates relative to the Program Segment Prefix (PSP) rather than to the .EXE module paragraph (10h paragraphs lower)

SST has limited support for program variables with the `lv` option. This option loads the variables defined by the part of a MASM.EXE listing for a single segment. The program scans for the word "segment" and sets up program variable names up to the corresponding ends pseudo op.

The `ll`, `lm`, and `lv` options use the user program area to load in the .MAP and .LST files and hence overwrite whatever program might have been loaded in. Hence to debug a program, load in the label files first, and then the program.

Labels can be displayed by segment paragraph number by the `d/`*n* command of Chap. 7. Variable names are displayed by `d/v` and user strings by `d/u`.

The `ll` option can also read in label files generated by the `wl` option, which writes the entire SST label linked list to disk, program labels, variable names, and user strings. See the write command in Chap. 7.

For the more technically oriented, we note that internally SST stores all labels in a two-level linked list format. The outer level linked list consists of one or more entries having a segment paragraph value (one word), followed by a one word segment label string length, followed by a string of that length. In turn, an outer level string contains one or more inner level linked lists, each describing a label. An inner level linked list has the same format for its entries, with a one word offset value followed by a one word string length, followed by a label string of that length. The special segments for `lv` and `lw` are identified by the outer level paragraph numbers 0FF00H+"V" and 0FF00H+"W", respectively, which are not likely to occur as program paragraphs. When you use the `wl` option, a double linked list of this form preceded by the special word 0FFFAh is written to the file named by the name command. In writing the file, the paragraph values less than 0FF00h are unrelocated by subtracting the Program Segment Prefix paragraph + 10h. This file can subsequently be reread by the `ll` option, which adds the Program Segment Prefix + 10h paragraph back in. In this way, the labels can be used when your program is loaded in a different place on subsequent occasions.

Blocks

It's often useful to scan through memory using the DISPLAY MODE and then examine part of memory in a different way, or write it out to disk. For such purposes, SST has a limited form of the word processor block facility. Specifically any time you type Ctrl-T in DISPLAY MODE, the address pointed to by the cursor is saved in a double-word Tag location.

You can examine the memory at the Tag location using the examine command of Chap. 7 by typing Ctrl-B instead of an address. The Ctrl-B so used displays as a little box.

Similarly if you type Ctrl-E in DISPLAY MODE, the address pointed to by the cursor is saved in a double-word End tag location. You can go back and forth between these two locations by typing Ctrl-G.

You can use the block of memory between the Tag and End-tag locations in a number of ways. You can write a block to disk by naming the desired file with the name command and then typing

```
▸ w Ctrl-B
```

in COMMAND MODE. The Ctrl-B can also be used in place of two addresses with the compare, fill, move, and search commands. For example,

```
▸ f Ctrl-B "abcd"
```

fills the memory defined by the previous Ctrl-T Ctrl-E DISPLAY MODE option with the string "abcd". The compare, fill, move, and search options are limited to 64K, but the write option is limited only by the RAM and disk sizes.

User Strings and Keyboard Macros

User strings are definable with two character alphanumeric names. These strings can be used for defining memory structures or *templates* for use with the examine memory command (see Chap. 7), and for user input (macros without arguments). The facility makes it much easier to read the values of data structures stored in memory. To define any user string, type

```
▸nst="..."
```

This defines (names) the string `st` to have the value ... .

You can also define 40 function key values by assigning strings to **f0 - f9**, **c0 - c9**, **s0 - s9**, and **a0 - a9**, which define the unshifted, Ctrl'd, Shifted, and Alt'ed functions, repectively. For example,

```
▸f9="
dd;"
```

defines Funtion Key 9 to switch back to COMMAND MODE if it's not there already, and to display the 8088 interrupt vectors in double-word format.

To use a string in place of keyboard input (limited macro facility), type the command **$** followed by the string name. Hence the sequence

```
▸nst="dd;"
▸$st
```

generates a full screen double-word display of the 8086 interrupt vectors, just as if you had entered the command `dd;` directly. This facility is particularly useful for abbreviating multiline command sequences that you use often. For example in debugging SST itself, we might use the string

```
▸nls="nsst.map
ll
nsst.lst
lv
nsst.exe
l"
```

which we keep along with other useful strings in a macro file called SSS. Then typing

```
▸nsss
▸ll
▸$1s
```

loads in the current labels, variable names, and SST.EXE for debugging.

Alternatively, you can put these commands in a file and run them using keyboard redirection. This method is really simpler. For example we might debug SST.EXE by using the file S

```
nsst.map
lm
nsst.lst
lv
nsst.exe
l
qs3
```

and then invoke this file by the keyboard redirection command

```
▸n<s
```

Chapter 5
CALCULATOR MODE

In addition to the alphabetic commands described in Chaps. 7 through 13, if a decimal digit or 8086 register name starts the command, the line is assumed to be calculator input to a hexadecimal Polish suffix calculator. This calculator supports 32-bit arithmetic with the arithmetic operators "+-/*", the binary logical operators "!&", the logical not "~", and the store operator "=". See also the more limited hex command, which is included to be compatible with DEBUG.COM. If you have an 8087 or 80287 numerical coprocessor, you can use the floating-point calculator described below. As an example, typing

```
▸9 9*
```

followed by a ↵ displays

```
▸9 9*= 51
```

Operands must be separated by at least one blank, and the operators must follow their respective operands without intervening blanks (the operators are treated as special operand delimeters)

Converting between Hexadecimal and Decimal

Typing a single value followed by a ↵ with no operators displays the corresponding decimal value. Similarly, typing a decimal value (identified by trailing period) followed by a ↵ displays the corresponding hexadecimal value. Examples are

```
▸ØACE= 2766.
▸127.= 007F
```

To convert from hexadecimal to binary, in COMMAND MODE type h or H followed by the hexadecimal value.

To convert decimal to binary type h or H followed by decimal value [illegible]

Use of Register Values

Typing

```
▸ds 4*
```

displays the value of the **ds** register multiplied by 4. More complicated expressions like

```
▸ØFE 4 ds*&
```

can be used. This one calculates the infix expression (4*ds)&0FEh. If ds=0BCh, this gives 0f0h. Note that a leading 0 is required to indicate that 0FEh is a number and not a fill command.

To store the value of an expression into one of the 8086 registers, use the = operator at the end of the expression. For example, to set **bx** = 2***ax**+**cx**, type

```
▸ax 2* cx+ bx=
```

Floating Point Calculator

SST has an 8087 reverse-Polish-notation (RPN) floating point calculator that supports +-*/^%, trig, hyperbolic, exponential, and other functions described below. The calculator requires the presence of the 8087 numeric coprocessor and issues a message encouraging you to install one if you haven't. The calculator has several major advantages over a pocket calculator:

- the large screen allows substantially more information to be displayed
- it's much faster
- it produces 16-decimal place answers (unless you specify less precision)
- in SST resident mode, the result of a calculation can be inserted directly into the keyboard input queue as if you had typed it.

This last feature adds speed and accuracy to your calculator functions. The 8087 has 8 stack locations, and SST's 8087 calculator is limited to this depth. Some functions, such as **sin** and **cos** require a total of three stack locations, which must come from the 8. Trig functions assume that their arguments are in degrees until instructed otherwise by the **rad** command.

For quick calculations, you can use the calculator directly in COMMAND MODE by starting the line with a "." Quick help in COMMAND MODE is given by typing . Function Key 1. For example, we have

```
▸ .16 2^= 65536

▸ . 45 sin= 0.707106781186548
```

8087 CALCULATOR MODE

For more extended calculations, switch into the 8087 CALCULATOR MODE, by typing "." in COMMAND MODE followed by ↵. In CALCULATOR MODE, the leading period is not used and a special menu line shows some calculator functions along with the degree/radian and stack status's. CALCULATOR MODE remains on until you type the Esc key. For quick help, just type Function Key 1 at any time.

When SST is resident (SST/R), type Ctrl-. to interrupt the program you're running and enter CALCULATOR MODE directly. Perform whatever calculations you want and return to your program by typing **bye**. If you want to insert the result of the calculation into the keyboard input queue as if you had typed it in, type the Ins key. This inserts the result accordingly and returns to your program. For example, to insert the value of pi (3.141592653589793) accurate to 16 decimal places at the cursor position in your word processor, type Ctrl-., pi Ins. Done! In fact that's exactly what I just did to get that value into this help file.

Calculator Stack

Ordinarily the 8087 calculator works with its own stack, saving the complete 8087 state before using the 8087 and restoring the 8087 state when the calculation is completed. Alternatively, the stack-on command uses the current user 8087 stack and status. This stack-on mode is handy for manipulating the 8087 stack while debugging, and it displays the stack continually. The user stack is the one examined by the z command and displayed by the 7 option in TRACE MODE.

For either 8087 stack, the **x***n* function eXchanges the stack top with the *n*th ($0 \leq n \leq 7$) stack location. For example, **x1** exchanges the stack top with the next stack location. Calling the stack top *x* and the next location *y*, the **x1** command exchanges the contents of *x* and *y*. For either 8087 stack, the **p***n* command pushes the value of the *n*th stack location onto the stack. In particular, **p0** duplicates the stack top.

In the stack-on mode, **s***n* works the same way as **p***n*, but in the stack-off mode, **s***n* pushes the *n*th user stack location. This allows you to use the user 8087 stack for input without affecting that stack. You can also store a result back into the *n*th user stack location by typing **s***n***=**.

If you type in an integer with no functions or operators, the calculator displays it in hexadecimal. To convert from hex to decimal, type in the hexadecimal number followed by `h` or `H` and ↵. The advantage of using the 8087 calculator instead of the usual SST hex calculator is that full 64-bit arithmetic is supported instead of 32-bit.

To get an idea of how to use the calculator, enter CALCULATOR MODE and start typing commands. Note that if you push too many values onto the stack, you'll start seeing ??, meaning "Not a Number" (NaN), which is one of several illegitimate values tagged by the 8087. Similarly once you get an infinite answer (perhaps by dividing by 0), subsequent functions continue to yield infinity even if you divide it into a finite number. You can pop illegal values off the stack using the **pop** command, or store new values on the stack with the **=** command. To reinitialize the user stack (all stack locations empty), type **init**.

8087 Built-in Functions

The following gives a brief description of the built-in 8087 functions. In the discussion, x refers to the 8087 stack top, while y refers to the next stack location. For example, "x = absolute value of x" means that the absolute value of the stack top replaces the stack top. Push value pushes the value indicated onto the 8087 stack. Thus pi pushes the value of π (3.141592653589793) onto the stack.

abs	x = absolute value of x
acos	x = arc cos(x)
acosh	x = arc cosh(x)
asin	x = arc sin(x)
asinh	x = arc sinh(x)
atan	x = arc tan(x)
atanh	x = arc tanh(x)
bye	return to DOS, or if resident, return to interrupted program
cabs	$x = x*x + y*y$, and stack is popped once
cexp	x = cos(x), y = sin(x) (stack is pushed once)
chs	$x = -x$
cls	clear screen
cos	x = cos(x)
cosh	x = cosh(x)
cot	x = cot(x)
csc	x = csc(x)
deg	interpret trig arguments in degrees (default)
e	push e (2.718281828459045)
exp	x = exp(x)
hex	display x in hex (automatic if you type in a number alone)
init	reinitialize the user stack
int	x = integer(x) (chopped down)
inv	$x = 1/x$ (INVerse)
ln	x = ln(x)
log2	x = log base 2 (x)
log	x = log base 10 (x)
in	push number typed in by user
pi	push π (3.141592653589793)
pop	pop stack (throw away x, so that y becomes x)
prec	set display precision = x

rad	interpret trig arguments in radians
ranf	push next random number in sequence determined by seed
sec	x = sec(x)
sech	x = sech(x)
seed	seed for random numbers = x
sin	x = sin(x)
sinh	x = sinh(x)
sq	x = x*x
sr	push x from last calculation
sqrt	x = sqrt(x)
stack	on (off) - turn user 8087 stack on (off)
tan	x = tan(x)
tanh	x = tanh(x)
todeg	$x = 180 * x/\pi$
topol	x = sqrt(x*x + y*y), y = atan(y/x)
torad	$x = \pi * x/180$
torec	x = x*cos(y), y = x*sin(y)
z	display full 8087 status/stack values

The % operator computes the percent difference between x and y, that is, $y = 100*(x-y)/y$, and then it pops the stack.

Notes:

Chapter 6
INTERRUPTS

Hardware and software interrupts play important roles in 8086/8088 based computers like the IBM PC. Hardware interrupts are used to maintain the date and time of day, keyboard buffering, some disk control operations, 8087 numeric coprocessor exceptions, and optional serial and parallel input/outut data transfers. Software interrupts are used to connect programmer routines with operating system routines, to handle arithmetic exceptions such as divide overflow, and to handle single-step and breakpoint operations. To control these operations effectively, SST takes over many of these interrupts and restores them to their previous values upon returning to DOS. For example under IBM DOS and alleged DIVIDE OVERFLOW interrupt is identified as such and the machine either halts or returns to DOS, in either case preventing you from finding out where and whether an overflow occurred or whether a software interrupt occurred instead. SST gives the same message if a divide overflow really did occur and in any event leaves you pointing to the instruction that caused the interrupt. You can then investigate the conditions that caused the problem and return to DOS to correct your program accordingly.

More specifically, unless you specify /L (for tread Lightly) when invoking SST on the DOS command line, SST takes over interrupts 0 (divide overflow), 1 (single-step), 2 (nonmaskable interrupt), 3 (breakpoint), and 4 (overflow). On 80186/80286-based machines, SST also takes over 5 (for trapping the **bound** instruction), 6 (illegal op code), and on 80286's 7 (80287 not available). On all machines it takes over **int** 9 (keyboard), **20h** (return to DOS), **21h** (main MSDOS software interrupt), **22h** (terminate address), **23h** (Ctrl-Break), **24h** (Critical Error Handler), and **27h** (return resident to DOS).

SST takes over the keyboard input interrupt 9 to see if a Ctrl-Enter has been typed. If so, SST takes control, which allows you to stop a runaway program. For any other key combination, SST transfers control to the keyboard routine active at the time SST was loaded. SST takes over the MSDOS interrupt 21h to intercept DOS exit requests (**ah**=0, 31h, and 4Ch). If these are encountered, SST takes control issuing the message "Program terminated normally." If not, SST transfers control to the MSDOS program active at the time SST was loaded.

Hardware Interrupts

For hardware interrupts, the corresponding interrupt programs can be invoked either by a real hardware interrupt or by software executing an **int**, far **call**, far **ret**, of far **jmp** instruction. SST identifies the software **int***n* cases as such, and otherwise gives the appropriate hardware interrupt message. For example, if your program executes an **int 0** instruction, you'll see the message `int 0`, rather than code leading to the interrupt using the **u**nassemble command.

Another example on the IBM PC is the message "PARITY CHECK 2", which allegedly means that some memory location may have caused the error. You can run a memory test program to check your memory, but the interrupt could have been caused by a software bug, namely an **int**, **call**, **ret**, or **jmp** that uses interrupt vector 2, the nonmaskable interrupt vector. Running under SST, you can check the origin of the problem, and either go fix your program or your memory accordingly.

NMI Button

Very usefully, the NMI interrupt can be caused by your shorting the I/O Channel CHK line to ground with an NMI button (connect normally open push button switch to top and bottom I/O Channel pins closest to rear of PC). This is a very powerful way of giving SST control when ordinary maskable interrupts have been disabled.

Interrupt Mask Control

Particularly in debugging multitasking systems that use the system clock to switch tasks, it is important that SST can control the interrupt mask active when SST is active. Otherwise, SST could regain control and immediately lose it to some other task. For this purpose, the command

▸ `qi` *mask*

sets the interrupt controller mask used when SST is active to the value *mask*. For example on the IBM PC, to allow only keyboard interrupts, use

```
▸ qiFD
```

You can try this option out if you have a resident clock routine such as that given in the book, *The IBM Personal Computer from the Inside Out.* This routine shows the seconds ticking away in the upper right corner of the screen. When the 0FDh interrupt mask is used, SST stops the screen update whenever SST is active, and then restarts it upon return to the user program.

DOS and 8087-Emulation Interrupt Definitions

SST automatically comments some unassembled instructions, such as DOS calls (**int 21h**) and the 8087 emulation interrupts (**int 34h - int 3dh**). In addition the **int 21h** definitions are displayed when you type the **int21** [*n*] command in COMMAND MODE. If the optional *n* is present, the definition for that entry point alone is displayed. If *n* is missing the next hexadecade of **int21** entries is displayed. These features are very handy for working with code that makes DOS calls.

Examining Interrupt Vectors

To facilitate examining interrupt vectors, the suffix "i" on an address *address* automatically implies the address 0:4**address*. Hence the command

```
▸ dd17i
```

displays the 8088 interrupt vector table with the cursor position at 0:5Ch, which has the double-word vector to **int 17h**. You can then type Ctrl-F to begin disassembling at this routine.

To exit: F7, then Display

To examine memory location: D

Chapter 7
COMMAND DESCRIPTIONS

This section describes the basic SST commands summarized under Sec. 3 on Help as

!&.<>	0-9	@scii	Asm	Baud	Comp	Display
Exam	Fill	Go	Hex	In	Klear	Load
Move	Name	Out	Protect	Quit	Reg	Search
Trace	Unasm	Vector	Write	Xam	YGDT	Zam

It addition, this chapter summarizes many DOS-like commands as given in the Table of Contents. Commands specific to the interpreter, the disk editor, and the RamFont editor are described in Chaps. 8, 9, and 10, respectively. The remaining commands are discussed in this and later chapters. They are introduced by a brief syntax specification (see Chap. 4) followed by an explanation of the command and some examples.

! SHELL Command

The command ! loads and executes copy of the DOS COMMAND.COM. The syntax is

▸ ! [*filename* [*parameters*]]

If the optional *filename* and *parameters* field are present, the file is executed with the command-line parameters specified and control is returned to SST. If the ! appears alone, COMMAND.COM retains control giving the user the usual DOS command line prompt. Type any commands desired and return to SST by typing the DOS command **exit**.

Note that SST and whatever you're debugging remain resident during this process, substantially reducing the amount of RAM left to the new COMMAND.COM process, relative to the one used to run SST in the first place. Many DOS commands such as **dir** are built into SST, so in many cases you may not have to use the shell command.

& Address Command

The & *label* command returns the address of the variable or label *label*. If *label* is not in the symbol tables read in (see **LL** command), an error message is issued. Type the **d/s** command to see what segments have symbol tables.

. Calculator Command

The . command switches to floating-point CALCULATOR MODE. See Chap. 5 for further description.

0–9 Calculator Command

Commands beginning with 0–9 invoke the 32-bit hex calculator option as described in Chap. 5.

< Command

The < *filename* command redirects keyboard input to the file *filename*. This is very useful for reading in script files to define symbol tables, function keys, and initialization commands.

> Command

The > *filename* command defines the file *filename* to be used for echoed output.

@ Command

The @scii command displays hexadecimal and decimal ASCII and EBCDIC charts as described in Chap. 2.

A Command

The **a** command assembles 8086 and 8087 mnemonics. In SST it assembles 80186 and 80286 extended mnemonics as well. It has the syntax:

▸ **a** [*address*]

This starts assembling instructions typed in by the user at the address given following the a, or at the last address used (initially cs:100) if no address is given. The menu line (third line from screen top) changes to

```
ASSEMBLE MODE:  F1 Esc
```

The help screens displayed by Function Key 1 are discussed below. The first time an a ↵ is typed you see (suppose cs=1234)

```
▸ a
1234:100          _
```

After you terminate an assembly language instruction with a ↵, the screen displays the corresponding machine language and goes on to the next line. The assembly language mode is terminated by two ↵'s in a row. Thus to add the first ten integers one types and sees

```
▸ a
1234:100  B90A00        mov   cx,a
1234:103  31C0          xor   ax,ax
1234:105  01C8          add   ax,cx
1234:107  E2FC          loop  105
1234:109
▸
```

You can screen trace the operation of your program by typing `t` or `T` followed by ↵ and single stepping by typing the space bar. From the TRACE and UNASSEMBLE MODEs, you can invoke the assembler on the line given by the cursor by typing the hot key "a".

Assembler Syntax

In the a option, all numbers are assumed to be hexadecimal unless identified as decimal by a trailing period, or identified as a string literal by enclosing in single or double quotes. Extra spaces and tabs can be be freely inserted to improve readability. Memory references are usually chosen to be byte or word according to the register that appears in the instruction. Hence the instruction `mov [100],ax` moves a word to the location 100, while `mov [100],al` moves a byte. If no register is mentioned, such as in immediate transfers like `mov word ptr [100],10`, the usual modifiers **word ptr** and **byte ptr** can be used. These can be abbreviated by **word** and **byte**, respectively, or simply by **w**, and **b**,. For example,

```
shl word ptr [100],1
shl word [100],1
shl     word       [100],1
shl w,[100],1
```

all assemble the machine instruction that shifts the word at location **ds**:100 left one bit position. Either a byte or a word specification must be given. In general syntax accepted by DEBUG.COM is accepted by SST as well.

In ASSEMBLE MODE, Function Key 3 displays code for instruction at current address for editing. You can also edit a sequence of instructions in a row by using the **edit** command described in Chap. 8.

8087 Instructions

For the 8087 instructions, the specifications **d**,, **q**,, and **t**, are available to mean double word, quad word, and ten byte (temporary real – **fld** and **fstp** instructions only), respectively. The usual assembler notation **dword**, **qword**, and **tbyte** followed optionally by **ptr** is also accepted. The stack registers can be referenced by their full names **st**(*i*) with *i*=0 to 7. For simplicity they can also be referenced by **st0** to **st7**, with **st** alone meaning **st(0)**. Most simply, they can be referenced by 0 to 7, an unambiguous specification since no immediate instructions exist on the 8087.

The arithmetic instructions **fadd**, **fmul**, **fsub**, and **fdiv** can appear with no arguments, in which case the operand field **st(1),st(0)** is implied and a pop occurs. For example, **fadd** means

```
faddp st(1),st(0)
```

This abbreviation makes arithmetic instructions without operands work as in a Polish suffix calculator.

8086/80186/80286 Mnemonics

To see all the mnemonics recognized by the assembler, type the command a F1. This displays the help screen

```
a [address]      Assemble. Op codes (186/286/386
                 capitalized, F1 → 8087) are:

aaa     aad     aam     aas     adc     add     and     Arpl
Bound   Bsf     Bsr     Bt      Btc     Btr     Bts     call
cbw     Cdq     clc     cld     cli     cmc     cmp     cmpsb
Cmpsd   cmpsw   Cts     cwd     Cwde    daa     das     db
dec     div     dw      Enter   esc     hlt     idiv    imul
in      inc     Insb    Insd    Insw    int     into    iret
Iretd   ja      jae     jb      jbe     jc      jcxz    je
jg      jge     jl      jle     jmp     jmps    jna     jnb
jnc     jne     jng     jnl     jno     jnp     jns     jnz
jo      jp      jpe     jpo     js      jz      lahf    Lar
lds     Lfs     lea     Leave   les     Lgdt    Lgs     Lidt
Lldt    Lmsw    lock    lodsb   lodsw   loop    loope   loopne
loopnz  loopz   Lsl     Ltr     mov     movsb   Movsd   movsw
Movsx   Movzx   mul     neg     nop     not     or      out
Outsb   Outsd   Outsw   pop     Popa    Popad   popf    Popfd
push    Pusha   Pushad  pushf   Pushfd  rcl     rcr     rep
repe    repne   repnz   repz    ret     rol     ror     sahf
sal     sar     sbb     scasb   scasw   seg     Sgdt    shl
Shld    shr     Shrd    Sidt    Sldt    Smsw    stc     std
sti     stosb   stosw   Str     sub     test    Verr    Verw
wait    xchg    xlat    xor
```

Typing any character other than Function Key 1, restores what was on the screen before you asked for help. If you type Function Key 1 again, you see the 8087 mnemonics:

a [address] Assemble. 8087 op codes (F1 → 8086) are:

f2xm1	fabs	fadd	faddp	fbld	fbstp	fchs	fclex
fcom	fcomp	fcompp	fdecstp	fdisi	fdiv	fdivp	fdivr
fdivrp	feni	ffree	fiadd	ficom	ficomp	fidiv	fidivr
fild	fimul	fincstp	finit	fist	fistp	fisub	fisubr
fld	fld1	fldcw	fldenv	fldl2e	fldl2t	fldlg2	fldln2
fldpi	fldz	fmul	fmulp	fnop	fpatan	fprem	fptan
frndint	frstor	fsave	fscale	fsqrt	fst	fstcw	fstenv
fstp	fstsw	fsub	fsubp	fsubr	fsubrp	ftst	fwait
fxam	fxch	fxtract	fyl2x	fyl2xp1			

For use of these commands, see the Int$_e$l *iAPS 86/88, 186/188 User's manual*, the Int$_e$l *iAPX 286 Programmer's Reference Manual*, the Microsoft *Macro Assembler* manual, or the book *The IBM Personal Computer from the Inside Out*, by Sargent and Shoemaker.

If an error is found, the assemble command indicates the offending letter or field by an Up arrow followed by the word error as in

```
        mov     ax,q
                  ↑    error
```

You can then use the command edit keys (see Chap. 4) to fix the error, or type Esc to return to COMMAND MODE.

Labels and Comments

You can label instructions as you type them in with the assemble command. Terminate the labels with a colon (:), and they are inserted into the label table for use in later assembles, unassembles, examines, displays, etc. A colon alone deletes the label at the current program counter. A new label replaces an old one. You can save the labels you type and/or read in with the wl command.

Use the `ll` command to read in labels written with the `wl` command.

You can also define labels by reading them in from the **/MAP** option (`ll` command) on the linker and **/LST** option (`lv` command) on the macroassembler. Currently a macroassembler listing is the only way to insert variable names into SST unassembles. The assembly language interpreter (Chap. 8) can have variables of its own by using the **db** and **dw** pseudo-ops, or by storing a value into an as-yet undefined variable.

Program comments are supported by the assemble and unassemble commands and are defined by a starting semicolon. If you type a semicolon at the start of a line, the comment that follows will automatically be appended to the instruction for that line. If you type a semicolon alone, any existing comment for the line is deleted. If you type a new comment, it replaces the old one.

Program Mode

The **end** pseudo op stops TRACE MODE unassembly beyond the end, giving a more readable screen. This pseudo op can only be used in .COM file mode, i.e., with cs equal to the program prefix segment. When **end** is in effect, you can edit, insert, and delete instructions in the middle of your program. See Chap. 8 on the built-in assembly language interpreter.

The a option acts essentially the same way as for DEBUG.COM with the additions of instant help messages (just type Function Key 1), of displaying the assembled machine language, of recognizing the 80186 and 80286 extended mnemonics, use in search strings, with the change to lower case for increased readability, use of labels and comments, and with the PROGRAM MODE allowing insertion and deletion of instructions.

SST also uses the assembler for the Super-Trace and conditional breakpoint facilities.

AND Command

The logic operations **and**, **or**, **xor**, and **not** operate on memory ranges much like the fill command. The **and**, **or**, and **xor** have the same syntax as the fill command, while the **not** command simply inverts all bits in the range. The syntax is

‣ **and** *range list*
‣ **or** *range list*
‣ **xor** *range list*
‣ **not** *range*

The bytes in the list *list* are **anded**, etc., with the bytes in the range *range*. The **and** operation can be used to kill the high bit on bytes in WordStar files or the parity bit included by some communications programs. For example, to kill the high bit in the file WORD-STAR.DOC, type

```
‣ nwordstar.doc
‣ 1
‣ and 100 1 cx 7F
‣ w
```

To reverse video font t5, type

```
‣ xor t5 11000 FF
```

B Command

This command manages breakpoints and setting the baud rate. To set the baud rate, type a command of the form

‣ **b** *rate* [,*channel*]

where *rate* = one of 110,150,300,600,1200,2400,4800,9600. The optional channel value of 1 specifies COM1 and 2 gives COM2.

If no number is given, the message

Reboot System (Y/N)?

is displayed. Typing `y` or `Y` reboots the system without erasing memory (works only on older IBM PC's at the moment).

Breakpoint Commands

In addition to the 10 normal *g* breakpoints (see go command) that go away upon reentry to SST, you can define up to 10 sticky breakpoints with the breakpoint command. To set sticky breakpoints type a command of the form

▸ **bs**[*n*] *address* [*m*]

which sets breakpoint [*n*] at the address *address* and skips *m* passes by this address. No blanks can occur between *n* and the `s` if *n* is given.

To clear breakpoints in *list* or all (*), type a command of the form

▸ **bc** *list*

or type

```
▸ bc  *
```

The list *list* here refers to one or more digits 0 to 9, and the * refers to all ten possible breakpoints.

Similarly to disable breakpoints in *list* or all (*), type a command of the form

▸ **bd** *list*

or type

```
▸ bd  *
```

and to enable **breakpoints** in *list* or all (*), type a command of the form

```
▸ be list
```

or type

```
▸ be *
```

To list your breakpoints and their characteristics, type

```
▸ bl
```

For example,

```
▸ bs0 1234:5678
```

defines and enables sticky breakpoint 0 at the address 1234:5678.

```
▸ bd0
```

disables sticky breakpoint 0. Hence if you go to the program, this breakpoint will not be set. The `be0` command can reenable the breakpoint.

After defining one or more sticky breakpoints, type the `bl` command to see a pop up window telling you about the status and location of the breakpoints. This facility is compatible with SYMDEB.EXE's except that the latter uses **bp** instead of `bs` to set BreakPoints. SST uses **bp** to refer the **bp** register.

BLINK Command

The command

▸ **blink** [*n*]

controls the Hercules Graphics Card Plus blink/reverse-video attribute. If *n* = 0 or is missing, then bit 7 = 1 of the screen attribute byte specifies a high-intensity background. If *n* = 1, then bit 7 = 1 of the screen attribute byte specifies blinking. Use `blink 0`!

BYE Command

The command

▸ **bye**

quits to DOS. **system** and **quit** do the same thing.

BYTE Command

To facilitate both source-level and assembly language debugging, SST includes commands to define typical data types. The syntax for the **byte** type is:

byte [[[far]] *]] *variable name address*

For example,

```
byte alpha 305
```

adds the symbol `alpha` of type **byte** (8-bit unsigned integer) to the segment specified by the **ds** segment register at the offset 305h. You can specify any other segment. The optional * generates a near **ptr** to a variable of the type **byte**. The optional **far** generates a far pointer to a variable of the type **byte**.

C Command

This command compares the contents of one memory block to that of another memory block (like DEBUG). It is useful for checking that two copies of a program are identical and have not been changed, for example, by a crash of the system. The command expects the first block start and end addresses and the second block start address as arguments. Syntax:

▸ **c** *range address*

For example,

```
▸c500,595,2000
```

compares the contents of memory from address 500h through 595h against the contents of memory from address 2000h through 2095h. Any differences will be shown on the display. Thus if location 527h was 00 while 2027h was FFh in the above example, the display would show

```
▸c500,595,2000
2347:527  00 FF
```

assuming all other locations in the blocks are identical.

If **ip**=235, cs=0200, es=3000 and **di**=495, then

```
▸ccs:ip E000 es:di
```

compares 0200:235 up to 0200:E000 against the memory block starting at 3000:495.

This command works the same way in DEBUG, and allows general register names to be used in the address fields.

CD Command

The **cd** command changes the directory as for DOS. Such changes cause the new path name to appear in a pop-up window. Alternatively, type the **prompt** command (described later in this chapter) to display the current path on all COMMAND MODE command lines (as for the DOS **prompt** command). The instead of the ▸ prompt, you see something like

```
C:\PS>
```

To change the current drive, type a command of the form *a*: or *a*;, where *a* is the desired drive letter.

CHAR Command

To facilitate both source-level and assembly language debugging, SST includes commands to define typical data types. The syntax for the **char** type is:

char [[[far]] *]] *variable name address*

For example,

```
char alpha 305
```

adds the symbol `alpha` of type **char** (8-bit signed integer) to the segment specified by the **ds** segment register at the offset 305h. You can specify any other segment. The optional * generates a near **ptr** to a variable of the type **char**. The optional **far** generates a far pointer to a variable of the type **char**.

CLOCK command

SST can display a time-of-day clock at right side of menu once a second. The command

▸ **clock** *status*

turns this clock on or off if *status* = `on` or `off`, respectively. If *status* equals an address instead of `on` or `off`, the word value at that address is displayed once a second. In addition to reporting the time of day, this feature lets you know if your machine has totally crashed.

CLOSE Command

The **close** command closes all opened files except those with handles 0 through 4.

CLS Command

The **cls** command acts as for DOS to clear the screen, here leaving the register window on top. This command also works in CALCULATOR MODE.

CONFIRM Command

The **confirm** command is typically used in demonstration script files to bypass the need for the user typing "y" or "n" to confirm an action. The syntax is

▸ **confirm** *status*

where *status* = `off` turns off the need to confirm, while the value `on` turns it back on (default is on).

CONT Command

The **cont** command continues operation where left off. It is the same as the **g** command without arguments, except that **cont** requires no confirmation if no breakpoints are implemented.

CPU Command

The **cpu** command gives you information about your computer. For example when you type

```
▸ cpu
```

on a typical IBM PC AT, you might see

```
CPU: 80286/80287
RAM: 0 1 2 3 4 5 6 7 8 9 A B C D E F
Serial Ports: 3F8 2F8
Parallel Ports: 3BC 378 278
DOS: 3.1
Speed Relative to PC I: 3.0
BIOS ROM: 01/10/84
```

Here the underlined values in the RAM entry are shown in reverse video on screen and signify that RAM exists in those 32K memory banks.

The program used to measure the speed relative to the IBM PC I is as follows:

```
comment |SPEED - display null-terminated string ds:[si]
followed by ratio of loop speed on machine to that of
IBM PC I.  ax, bx, cx, dx, si changed
|

speed:   cli                 ;No interrupts
         call   mark         ;Get ax = timer start count
         mov    cx,800h
speed2:  push   ax           ;Delay
         pop    bx           ;bx = original timer count
         loop   speed2
         call   mark
         sti                 ;Interrupts back on
         sub    bx,ax        ;bx = binary interval count
         call   tom          ;Display speed message
         mov    ax,0c100h    ;IBM PC I loop time (0c100)
         xor    dx,dx
         div    bx           ;ax = integer ratio
         push   dx           ;Save remainder in 0f000's
         call   dlbyte       ;Display integer part
         mov    al,"."       ;Display decimal point
         call   co
         pop    ax           ;ax = remainder
         mov    cx,10d       ;Convert to tenths
         mul    cx
         div    bx
         jmp    dlbyte       ;Display tenths' part and ret
```

```
comment |MARK - return ax = timer 0 count.   ax
changed
|

timer0    =      40h        ;8253 timer 0
timctl    =      43h        ;8253 timer control port

mark:     mov    al,0       ;Latch count
          out    timctl,al
          in     al,timer0
          mov    ah,al
          in     al,timer0
          xchg   al,ah
          ret
```

CSRSIZE Command

The command

```
▸ csrsize xxyy
```

begins the cursor on character raster row *xx* (starting with row 0 on the top) and ends the cursor on row *yy*. Hence the command

```
▸ csrsize b0c
```

gives a two-line cursor starting on line 11 (decimal) and ending on line 12.

D Command

This command displays the contents of memory in hex/ASCII (like DEBUG), pure ASCII, word, or double-word formats. In the hex/ASCII mode, sixteen bytes of memory are displayed per line with the starting address of the line given as the first entry on the line. In the pure ASCII mode, 64 (40 hex) bytes are displayed per line, allowing one to see four times as much memory on screen as with the hex/ASCII mode.

The command

```
▸ d [range]
```

displays the memory in the range *range*. For example,

```
▸ d100,200
```

displays memory from address 100h to address 200h inclusive. Usually this syntax is compatible with DEBUG.COM. However if one of the characters "`abdpw`" follows the `d` *with no intervening blanks*, that character is interpreted to choose the DISPLAY MODE ASCII, BYTE (hex/ASCII), Double word, triple-nibble (for 12-bit FAT displays), and Word, respectively. The mode so chosen is used by subsequent display commands until overruled. A particularly handy special case is the command

```
▸ dd;
```

which displays the interrupt vectors down at 0000:0000 in double word format. You can then use the cursor keys, blanks and backspaces to move to the desired interrupt vector and type Ctrl-F to start unassembling at the interrupt handler entry point. Note that `dd`*n*`i` displays the interrupt vector table with the cursor at interrupt vector *n*.

The output of this start/end display option can be written to a file of your choice (see `n>` *filename* command). Under MSDOS 2.0 and later versions, this file can be the printer instead of a disk file. The display command for a *range* quits with Ctrl-C and pauses with Ctrl-S. The maximum range length is 8000h.

Linear Address Display

A linear address display is available for Real Mode and protected 386 mode operation. In COMMAND MODE, type

▸ **dx** *offset*

where on an 80386 the offset can be 32-bit. Using this command in protected mode on the Compaq *DeskPro 386*, you can see that this machine wraps its address space at 16 megabytes, in true AT compatible form. If an address greater than 1 megabyte (100000H) is used, the "x" is implied.

This display mode is very handy for looking beyond 1 megabyte.

Screen Display

If no address or only the start address is specified, an instantaneous screenful of memory is displayed with a register/menu window on top. The offset at the cursor is displayed in the register window and the registers are displayed as discussed under the register command below. SST has the menu

```
DISPLAY MODE: F1 ←↑↓→ Tab ASCII Tag Word Dbl
              Near Far Cont Ovr Undo
```

The Tab entry means the Tab key. The A stands for Ctrl-A. To type this, type a or A while holding the Ctrl key down. This toggles the DISPLAY MODE between hex/ASCII and pure ASCII formats (see below). Typing Function Key 1 displays the help screen (Ctrl characters are shown in reverse video on screen)

←↑↓→	move cursor (csr)	PgUp/Dn	scroll screen
Tab	csr: HEX ↔ ASCII	ASCII	toggle ASCII/HEX
Byte	Byte format	Shift	invert case at csr
Tag	Tag csr position	End	End block at csr
Go	tag ↔ end	Restore	Restore csr
Z	display last char read	@scii	ASCII screen
Ovr	toggle OVERTYPE	Undo	Undo last overtype
Word	csr = Word ptr	Double	csr = Dword ptr
Near	csr = Near unasm ptr	Far	csr = Far unasm ptr
Cont	tgl Continuous update	↵	go to COMMAND MODE
V	unassemble at csr		

Cursor Movement and Memory Display Format

The cursor arrows, PgUp, PgDn, space bar, and backspace move the cursor around memory, scrolling the screen to keep the cursor on the middle line of the display (except near 0000:0000). Memory can be displayed in two formats, hex/ASCII and pure ASCII. The hex/ASCII format displays 16 (10h) bytes of memory per line, in hexadecimal form on the left and middle of the screen, and in ASCII on the right, as shown on the following page. The ASCII column replaces control characters (those with codes less than 20h) by periods. The pure ASCII format displays 40h bytes per line, with all control characters except code 0 by periods. Code 0 shows up so often that SST represents it by a o character.

Vertical motions can extend arbitrarily far up or down. In the pure ASCII mode, the autorepeated PgUp command displays each 64K RAM of memory in about 4 seconds, making it easy to scan all of memory for text. If you scroll towards lower addresses in memory than those displayed at the start of the display command, the address segments decrease by 10h. This allows you for example, to examine the bytes in the program prefix of an .EXE file with the program prefix segment displayed. If you scroll up in memory beyond the segment you started with, the segment displayed is increment by 1000h. If you display memory near or at the start of physical memory

(0000:0000), the 0000 segment is automatically used. This area of memory contains the 8086/8088 interrupt vectors, which consist of 4-byte pointers to the programs that handle the interrupts.

Stack segment displays automatically display stack frames pointers in reverse video, and referenced locations in boldface (as in TRACE MODE).

In hex/ASCII mode, the Tab key switches back and forth between the hex and ASCII display columns.

Ctrl-A switches back and forth between the hex/ASCII and pure ASCII formats (used also for X window in TRACE MODE).

Ctrl-B switches to the Byte (hex/ASCII) format

Ctrl-E Ends the block at the location pointed to by the cursor. The other end of the block is defined by Ctrl-T.

Ctrl-G Goes back and forth between the Tag and End tag locations.

Ctrl-S inverts (Shifts) the case of the character at the cursor.

Ctrl-T Tags the cursor location for use by the Ctrl-G command and for defining one end of a block that can be used in compare, display, examine, fill, move, search, and write commands (see Block section of Chap. 3).

Ctrl-Z displays last char read in by the last load command executed

As for all commands, ↵ (or Esc) returns to COMMAND MODE (ASSEMBLE MODE typically takes two ↵'s)

Memory Pointers

SST uses certain control characters to specify that the bytes starting at the cursor are to be used as pointers into memory for subsequent display and disassembly. This helps one to move around in memory without typing addresses.

Ctrl-C Continuously updates the display. This is optional, since some screens glitch with this process, and the keyboard response may be slowed down. This feature is handy to watch areas of memory being changed by interrupt-driven routines, such as the time-of-day clock, and the keyboard input buffer. Try using it while displaying 40:0 on an IBM PC.

Ctrl-D displays memory at the Double-word address indicated by the cursor.

Ctrl-F starts disassembling memory at the Far address indicated by the cursor. This is very handy for looking at the code of an interrupt handler. Display memory at 0000:0000 (just type the display command `dd;`, which acts like `dd0000:0000`), move the cursor to the desired interrupt vector and typed Ctrl-F. This displays the first instruction of the interrupt handler. Typing the space bar or PgUp displays subsequent instructions.

Ctrl-N starts disassembling memory at the Near address indicated by the cursor.

Ctrl-O toggles between DISPLAY MODE and OVERTYPE MODE as described below.

Ctrl-U Undoes the last overtype in case you type something by mistake.

Ctrl-W displays memory in the current segment starting at the offset given by the Word-address indicated by the cursor.

Overtyping Memory with SST

By typing Ctrl-O, you toggle between DISPLAY MODE and OVERTYPE MODE. In the latter when the cursor is located in the hex columns, typing hex digits overtypes those in memory. When the cursor is on ASCII columns, typing characters with blank or larger ASCII codes overtypes memory. Control characters must be entered as hex values. Since you may overtype memory by mistake and not know what value you overtyped, the Ctrl-U option allows you to replace the value of the last location overtyped. Use the overtype facility with caution. You may overtype something you don't mean to.

SST display syntax is upward compatible with Microsoft's SYMDEB, except that the 8087 modes remain with the examine command, and SYMDEB lacks the full screen mode. The command syntax sometimes gives different results from DEBUG.COM, but the result is always clear and the extra power is worth the change.

Binary Editor

If a non-com, non-exe file smaller than 64K bytes is read in, an elementary binary mode edit capability exists in addition to the usual overtype capability. The Del key deletes the byte at the cursor from the RAM image of the file, decrementing the user cx value accordingly. The Ins key inserts a binary null at the cursor, incrementing the user cx value accordingly. Such a file can be overtyped and bytes can be inserted and deleted, regardless of the file content, i.e., the file can have arbitrary binary values. The **w** command then rewrites the file with the new length back to the same place on disk (unless you use the **n** command to change the filename).

Displaying Labels

Labels can be displayed by a command of the form

▸ **d/*n***

where *n* is the number of a segment paragraph. Often the desired labels are in the current code segment, in which case type

```
▸ d/cs
```

To display program variable names, type

```
▸ d/v
```

To display user strings, type

```
▸ d/u
```

To find out the address (segment:offset) of a given label, type *&label_name*.

DATE Command

The **date** command in COMMAND MODE displays the current system date.

DEL Command

A command of the form

▸ **del** *filename*

deletes the file *filename*. This form could also have the unlikey interpretation as a Display command, starting at the offset 0Eh with the number of bytes specified after the "L". This possibility is superseded by the **del** command.

DELAY Command

A command of the form

▸ **delay** *n*

causes the code

```
mov       cx,n
loop      $
```

to be executed each time most characters are displayed on the screen. This slows down SST displays, mostly to help in the debugging of SST itself.

DELETE Command

A command of the form

▸ **delete** *n*

deletes the instruction at the offset *n*. This works only in Program mode (see Chap. 8).

DIR Command

Typing **dir** in COMMAND MODE acts very much like typing `dir/W` at the DOS command prompt, but also displays labels, and system, hidden, and directory files. You can follow the **dir** command with an arbitrary filename specification including path and asterisks. For example, the command

```
▸ dir .asm
```

or just

```
▸ dir.asm
```

displays all files in the default directory on the default drive with the extension .ASM. The filenames displayed are alphabetized. At the end of the display the total byte count for all file whose filenames are displayed is given in decimal if that number is less than 65536 or if the computer has an 8087. Otherwise the count is given in hex.

DISK Command

The command **disk** changes the display source for the **d** command from RAM to disk. See Chap. 9 for details. The command **ram** switches the display source back to RAM.

DOS Command

The **dos***n* command is the equivalent to typing `v21`*n*00 at the COMMAND MODE prompt. The only advantage is that if you type it incorrectly you won't execute some undefined interrupt vector by a mistake.

DOUBLE Command

To facilitate both source-level and assembly language debugging, SST includes commands to define typical data types. The syntax for the **double** type is:

double ⟦⟦far⟧ *⟧ *variable name address*

For example,

```
double alpha 3Ø5
```

adds the symbol `alpha` of type **double** (8087 64-bit floating-point) to the segment specified by the **ds** segment register at the offset 305h. You can specify any other segment. The optional * generates a near **ptr** to a variable of the type **double**. The optional **far** generates a far pointer to a variable of the type **double**.

DR Command

On 80386-based computers, the **dr** command displays the 80386 debug registers 0, 1, 2, 3, 6, and 7, the translate-lookaside registers 6 and 7, the control registers 0, 1, and 3, and the extended flags register. This command does not work when SST is run as a V86 task. It does work if SST is run in real or protected mode.

DWORD Command

To facilitate both source-level and assembly language debugging, SST includes commands to define typical data types. The syntax for the **dword** type is:

dword [[[[far]] *]] *variable name address*

For example,

```
dword alpha 3Ø5
```

adds the symbol `alpha` of type **dword** (32-bit unsigned integer) to the segment specified by the **ds** segment register at the offset 305h. You can specify any other segment. The optional * generates a near **ptr** to a variable of the type **dword**. The optional **far** generates a far pointer to a variable of the type **dword**.

E Command

This command examines or modifies memory on a byte by byte basis (like DEBUG), in various floating point formats, and according to user defined templates. The display command can also be used change memory, but the examine command is occasionally preferable, since the screen remains largely unmodified and 8087 data types are supported. To execute the command, type `e` or `E` plus an address and then hit the space bar. The system will respond by displaying the contents of memory at that address. Syntax:

```
▸ e address
```

For example, typing `e2000` followed by a ↵ results in

```
▸ e2000 00-
```

if the contents of location 2000h are 00. Typing in a value *nn* at this point will change the contents of location 2000h to *nn*h, while hitting the space bar will leave the contents of that location alone and display the contents of the next higher location. One can continue entering new values or hitting the space bar as long as desired. The command is terminated by hitting ↵. For example,

```
▸ e2000 00- 11-10 22- 33- 44-1210 55- 66-
```

would change locations 2001h and 2004h to 10 and leave the other locations unchanged. Note that the keyboard input routine uses only the last two characters typed before the space bar is hit. Thus in the above example, `12` was entered by mistake in location 2004h and then corrected by immediately typing `10` before the space bar was hit. One can also correct a mistake in the previous byte by pressing the backspace key to re-display the preceeding byte.

Floating Point Values

When an 8087 is installed, the SST examine command can also be used to examine and change long integer, packed BCD, and floating-point values in memory. If `e` [*address*] is followed by a / and one of letters `b`, `d`, `l`, `o`, `p`, `q`, `s`, `t`, or `w`, memory is examined in the following formats respectively:

BCD (10 bytes)
Double precision real (8 bytes)
Long integer (4 bytes)
O binary (1 byte)
P binary (2 bytes)
Quad integer (8 bytes)
Single precision float (4 bytes)

Temp precision (10 bytes)
Word integer (2 bytes)

For example, typing the command `e100/d` followed by a ↵, you might see

```
▸ e100/d - 0
```

At this point if you type a number in like `1.2345`, the 8 bytes at location 100h would be changed to the floating point number 1.2345. Subsequent typing of the space bar examines subsequent 8-byte double precision floating point quantities.

Structure Templates

Structure templates are used to display memory in customized formats that reveal the data in its natural form, rather than in one of the usual uniform formats like hex/ASCII. Such layouts include linked lists and data structures with mixed data types. The templates used to describe these data structures are mixtures of 1) alphanumeric names that begin with a letter, 2) single decimal digits, 3) $*n*, where *n* is a value < 100, 4) **$b** or **$w**, 5) >*n*, and 6) string literals '...'. The string names, contents of the string literals, and all other bytes are displayed as is. The digits and $ fields have special meanings as follows:

1, 5-9	display the next 1, 5-9 bytes in hex
2	display the next 2 bytes as a 16-bit word
3	display the next 3 bytes as a 24-bit word
4	display the next 4 bytes as a double word (segment:offset)
$*n*	display the next *n* ASCII characters from memory
$b	display the character string following the next byte in memory with length given by that byte
$w	display the character string following the next word in memory with length given by that word
$z	display null-terminated character string
$$	display $-terminated character string

>*n*	go forward the next *n* bytes
>b	go forward the number of bytes specified by the next byte in memory
>w	go forward the number of bytes specified by the next word in memory
>z	skip null-terminated character string
>$	skip $-terminated character string

<*n*	go backward the next *n* bytes
<b	go backward the number of bytes specified by the next byte in memory
<w	go backward the number of bytes specified by the next word in memory
<z	go backward to preceeding null-terminated character string
<$	go backward to preceeding $-terminated character string

=*n*	go to offset *n* in current segment
=w	go to offset specified by the next word in memory
=d	go to address specified by the next double word in memory
=s	go to offset 0 in segment specified by the next word in memory

/b	8087 BCD format (10 bytes)
/d	8087 Double precision (8 bytes)
/l	8087 Long integer (4 bytes)
/q	8087 Quad integer (8 bytes)
/s	8087 Single precision (4 bytes)
/t	8087 Temporary real (10 bytes)

The / options require the 8087.

For example, to read out an 80286 descriptor data structure with the macroassembler form

```
dscptr   struc                ;Descriptor
sglen    dw      ?            ;Segment max length
sgbase   dw      ?            ;Segment base low word
         db      ?            ;Segment base high byte
access   db      ?            ;Segment access byte
reswrd   dw      ?            ;Reserved word
dscptr   ends
```

define the string **ds** by

```
▸nds="sglen   2
sgbase  3
access  1
reswrd  2"
```

Then use the examine command as follows

```
▸ e address/ds
```

After the first ↵, subsequent space bars display the next structure entry in memory. User strings along with program labels, comments, and variables are all saved together by the `wl`, and can be reread by the `ll` command.

Useful DOS Examine Templates

A set of useful examine templates is given on the SST distribution diskette in the file STRUCT. These templates include those for the EXE header, the Program Segment Prefix (PSP), the File Control Block (FCB), the Extended FCB, the Drive Parameter Table (DPT), the Device Header, and the Bios Parameter Block. We are indebted to Guy Gordon of White Crane Systems for donating these templates for SST users.

The examine command is upward compatible with DEBUG.COM, except for the use of the backspace and extra digits in arguments. The command adds the ability to examine and change floating-point values in IEEE format and by user-defined structure templates.

ECHO Command

The echo of display and unassemble output to the printer or echo file set up by the n>*filename* command can be controlled by the **echo** command. Type

```
▸ echo on
```

to turn it on and

```
▸ echo on
```

to turn it off.

EDIT Command

A command of the form

▸ **edit** *address*

enters the ASSEMBLE EDIT MODE for the instruction at the address *address*. See the assemble command and Chap. 8 for further discussion.

EGA *n* Command

A command of the form

▸ **ega** *n*

determines the Enhanced Graphics Adapter's line/page mode. $n = 43$ chooses the 43-line mode and $n = 25$ chooses the 25-line mode.

ERASE Command

SST includes a subset of the DOS file commands for speed and convenience (see Chap. 4). One of these is the **erase** command, which is typed in COMMAND MODE in the form

```
▸ erase filename
```

After getting this command, SST asks you to confirm that you really want to erase the file *filename*. If you type y or Y, SST erases the file; otherwise it does not. You can interrogate the directory with the **dir** command in COMMAND MODE.

The alternate DOS form for erase, **del**, can be used in SST. This form could also have the unlikey interpretation as a Display command, starting at the offset 0Eh with the number of bytes specified after the "L". This possibility is superseded by the **del** command.

F Command

This command **fills** a block of memory with a constant of one or more bytes (like DEBUG). Syntax:

```
▸ f range list
```

For example,

```
▸ f1ØØ,1CØ,FF
```

fills memory locations **ds:100h** through **ds:1C0h** with the hex value FF.

The command can also fill a block of memory with a list of assembly language instructions, handy for checking speed of execution. For this option, the list is replaced by @ ↵, which transfers control to the assembler. Type in the instructions you want followed by two ↵'s in a row. This fills the memory range you give repeatedly with the instructions you give. For example,

```
▸f 100 1FE @
1234:0100            loop 100
1234:0102
```

fills 100h through 1FEh with a loop to here instruction. This sort of fill command is useful for finding out how fast pieces of code run (see Sec. 9-4 of *The IBM Personal Computer from the Inside Out*).

The command is upward compatible with DEBUG.COM, and adds the ability to fill memory with a set of instructions.

FILES Command

The **files** command is an alias for the **dir** command and has the same syntax.

.se"FLOAT Command"

To facilitate both source-level and assembly language debugging, SST includes commands to define typical data types. The syntax for the **float** type is:

▸**float** [[[[far]] *]] *variable name address*

For example,

```
▸float alpha 305
```

adds the symbol `alpha` of type **float** (8087 32-bit floating point) to the segment specified by the **ds** segment register at the offset 305h. You can specify any other segment. The optional * generates a near **ptr** to a variable of the type **float**. The optional **far** generates a far pointer to a variable of the type **float**.

FONT Command

The **font** *address* command displays memory in the special screen font format described in Chap. 10 on the Ramfont editor.

G Command

The **g** command allows a user to go execute a program with breakpoints (like DEBUG). These breakpoints go away when SST regains control. Sticky breakpoints are also available as described under the **breakpoint** command in this chapter. In addition, the SST go and sticky breakpoints can be made conditional, that is, whether execution is stopped when the instruction at a breakpoint is reached can be made to depend on a set of conditions specified by the user. When the go command is executed, SST loads the values in the register storage area into the proper registers, and then jumps to the requested program address.

The go breakpoint syntax is:

▸ g [=*address*] [*address*]...

For example,

```
▸ g1000
```

starts execution at cs:**ip** and breaks if cs:1000 is reached.

```
▸ g=1000,1020,es:1230
```

starts executing at cs:1000h, and breaks at cs:1020h or es:1230h, the program will return to the monitor, printing the register values. All register contents at the time of the breakpoint are saved. All previously set go breakpoints are cancelled when any breakpoint is reached. Breakpoints must be set only at locations corresponding to the first byte of an instruction. Additional breakpoint facilities are built into the screen trace mode, and greatly reduce the frequency that you need to use the unconditional go command. The breakpoint facilities use the **int 3** instruction, and only work in RAM.

Conditional Breakpoints

To make the breakpoints depend on a set of conditions, follow the **go** command specification (i.e., just before the ↵) by "@". This transfers control to the assembler to allow the set of conditions to be entered. The conditions are expressed by an arbitrary set of assembly language instructions. These instructions could in principle invoke software interrupts, call user subroutines, and do anything else that the machine can do. The conditions are specified the same way for the Super-Trace and for conditional breakpoints. Hence after a conditional breakpoint succeeds, a subsequent trace will automatically be a Super-Trace (note the S at the end of the second line from the top of the screen), unless the conditions are turned off with either a `g@` ↵ ↵ or a `t@` ↵↵ command.

There are three basic guidelines to writing conditional breakpoint code:

1. The **ax** and **bp** registers are saved before the user code is executed. It is your responsibility to save and restore any other registers you wish to use. The **bp** register is initialized to point to the program stack, with **bp-2** giving the user **ax** value, **bp+0** giving the **bp** value when the breakpoint was encountered, **bp+2** giving the **ip**, **bp+4** giving the cs, and **bp+6** giving the flags. The **ax** register is initialized to the first word of the current user instruction. For example, if the current instruction is a **ret**, then **al** = 0C3h, something you can break on.

2. The area reserved for user code is 40h bytes long. This is more than enough if you plan to type conditions in hand, but could be easily exceeded if you load a .COM file into the condition memory (to find the address of this memory, type `g@` ↵ ↵ or `t@` ↵ ↵, which in addition to displaying the condition

memory address turn off any active conditions). If you want to access a large amount of condition code such as a program profiler, make it a resident routine and access it through an **int** instruction.

3. Program execution is interrupted if the instructions set the Zero flag to 1, that is, if a **jz** instruction would jump. Otherwise program execution continues.

Writing Conditional Code

Chapter 3's section on "Super-trace Demonstration" illustrates the simple condition

```
        cmp     di,600
```

which succeeds if and only if **di**=600. Sometimes it is easy to express a condition that yields NZ rather than Z. For example, suppose you want the condition that **di**≠600. For this use the code

```
        cmp     di,600
        lahf
        test    ah,40
```

Here the `lahf`, `test ah,40` complements the Zero flag. Since SST saves **ax** for you, you don't have to worry about clobbering the **ah** register. More complicated conditions often require some conditional jump instructions as well.

If you want to stop supertracing on the next **ret** instruction, use the condition

```
        cmp     al,0C3
```

H Command

This command is included for primarily for compatibility with DEBUG.COM. The SST calculator provides a much more powerful facility. The **hex** command adds and subtracts two hexadecimal numbers. Syntax:

▸ **h** *value*$_1$ *value*$_2$

If the *value*$_2$ is missing, the binary equivalent of *value*$_1$ is displayed. To get hex/decimal conversions, see Chap. 5 on the "Calculator" section.

Examples:

```
▸h345,abc  E01  F889

▸h1Ø= 00010000
```

This command is upward compatible with DEBUG.COM's, adding the hex to binary conversion facility (used mostly for tutorial purposes).

HELP Command

The **help** command subjects the file SST.HLP to the SST **type** command. This allows you to browse/search through the SST.HLP file looking for online help.

I Command

This command displays the binary value read from any input port (like DEBUG). Type `i` or `I` followed by the port number. Syntax:

▸ **i** *portaddress*

Thus

```
▸i2Ø
```

inputs a byte from input port 20h. The input value is displayed in binary, i.e., if the value obtained from the input port was 45h, the display would show

```
▸i20  45=01000101
```

INI Command

The **ini** command reads and executes the file called SST.INI. This can be used to initialize SST for your standard set of parameters (see also the q command).

NOTE: If an sst.ini file exists in the default directory, it is automatically executed before SST displays its signon message.

INSERT Command

The command **insert** *n* enters the ASSEMBLE INSERT MODE at offset *n*. This works only in Program mode. See the assemble command and Chap. 8 for further details.

INT Command

To facilitate both source-level and assembly language debugging, SST includes commands to define typical data types. The syntax for the **int** type is:

▸ **int** [[[far]] *]] *variable name address*

For example,

```
▸int alpha 305
```

adds the symbol alpha of type **int** (16-bit signed integer) to the segment specified by the **ds** segment register at the offset 305h. You can specify any other segment. The optional * generates a near **ptr** to a variable of the type **int**. The optional **far** generates a far pointer to a variable of the type **int**.

INT21 Command

DOS **int 21h** entry definitions are displayed when you type the **int21** [*n*] command in COMMAND MODE. If the optional *n* is present, the definition for that entry point alone is displayed. If *n* is missing the next hexadecade of **int 21** entries is displayed. In addition unassembled **int 21h** instructions are commented with the corresponding entry descriptions. These features are very handy for working with code that makes DOS calls.

IV Command

The **iv** command initializes the screen RAM of the Micro Display Systems VHR monitor. This is a full screen display with 66 lines and graphics that can overlay the text-mode screen.

J Command

No commands currently begin with J.

K Command

The **k** command is used to clear (klear) the screen, to give a program stack trace, to reset the 8087 registers and status, and to get keyboard input codes.

K - Stack Frame Display

A special stack readout may enable you to trace subroutine calls. Most higher level languages support a recursive subroutine linkage convention that creates a stack frame for each call. Unless the compiler is optimizing code, each subroutine call saves the current value of **bp** on the stack and points **bp** at the saved value. Use of **bp** then allows access to the subroutine arguments which have been pushed onto the stack before the call and to local variable storage on the stack which is allocated upon entry to the subroutine. Immediately above the save **bp** value is the Near or Far return address.

To display such a stack trace, type

```
▸ k
```

in COMMAND MODE. Note that the k command can take arguments, which cause it to do other things as described in Chap. 7. For SST to display the trace, the numbers it encounters on the stack must make sense as stack frames. The **bp** register must contain a value at least as large as the **sp** register. The saved **bp** values must be greater than the current **bp** value and must increase monotonically. As soon as one of these requirements is violated, the trace terminates.

SST stack traces use the Microsoft Windows convention to determine if a return address is Near (16-bits) or Far (32-bits). Specifically, if the saved **bp** value is even (as it is whenever loaded as a frame pointer into **bp**), the return address on the stack is assumed to be a Near address, that is, 16-bits long. In contrast for a Far (32-bit) return address, the Microsoft Windows subroutine initialization code saves the **bp** value + 1, i.e., an odd value. SST's stack trace therefore assumes that an odd saved **bp** value signals the presence of a Far return address. The instruction preceeding that at the return address given in this fashion is examined to see if it is an appropriate **call** instruction. If so, a call trace display is given and the next frame is examined. SST also displays up to eight stack values in between the frames.

For a more general call trace, special coordination between SST and the .EXE symbol tables is needed. CodeView, for example, has such a facility.

Klearing the screen and 8087

The k command clears the screen (except for the register window at the top). This is useful when starting to assemble code following displays or traces that are irrelevant to your assembly. The **cls** command is an alias. This command does not exist in DEBUG.COM.

To **k**lear screen lines *n* through *m* (*n*=0 is screen top), type a command of the form

▸ **k** *n,m*

To **k**lear the floating point (8087) registers, type

```
▸ kf
```

To display the keyboard input code *c*, type a command of the form

▸ **ki** ↵ *c*

where ↵ stands for the Enter key.

KEY Command

The **key** command modifies the control characters used to edit command lines. See the section *Modifying Edit Command Characters* in Chap. 4.

KEYBOARD command

To deal with hostile keyboard environments, SST has a built-in **int** 9 keyboard encoder. When debugging programs under Microsoft Windows or in Protected Virtual Address Mode, this keyboard facility is ordinarily on. Otherwise is is left off, and SST gets its keyboard input from **int** 16, unless keyboard redirection of some sort is enabled. A command of the form

▸ **keyboard** *status*

enables or disables the built-in keyboard support if *status* = `on` or `off`, respectively. The facility doesn't handle the enhanced keyboard new keys yet.

KILL Command

The **kill** command works like the **erase** command to erase disk files, but allows the filename to be quoted (as in BASIC).

L Command

This command loads a file or absolute sectors (like DEBUG)

▸ L [*address* [*drive* *sector*$_1$ *count*]]

If the drive and sector specifications are missing, it loads file named by **name** command (see below) at the address specified on the `l` command line. If the address is missing, the file is loaded at cs:100. If the file has an .EXE extension, it is loaded as an .EXE file with appropriate address relocation and segment register initialization.

For example to **name** and **load** a program called TEST.COM, type

```
▸ntest.com
▸l
```

You can then type `t` or `T` to trace program execution, `u` or `U` to unassemble some code, or `d` or `D` to display the program in hex/ASCII.

If you make changes in the program, you can write the revised version back to disk using the write command. Be sure the **bx** and **cx** registers have the values they had when you loaded the file, since they determine how many bytes will be written. With SST you can not only read .EXE files as can DEBUG, but also write them back to disk. This is very handy for patching your favorite system programs.

The following error messages can occur:

File not found
Error in .EXE file

Load Labels

The load command is used also to load in program labels, variable names, and user macro and examine template strings. These facilities are described in greater detail in Secs. "Labels" and "User Strings and Keyboard Macros" in Chap. 4.

To load in program labels, name the .MAP file with the `n` command (Chap. 7), and type the `ll` command (see Chap. 7 load command). This automatically reads the labels in starting at the point in the .MAP file identified by the words "by Value" and relocates them relative to the origin of the .EXE module (program prefix segment paragraph + 10).

To load .MAP labels relative to some other paragraph, type `ll`*n*, where *n* is the desired paragraph number. This option is useful for debugging resident programs.

To load .MAP files for use with .COM files, type `lm`, which automatically adds 100h to the label offsets.

SST has limited support for program variables with the `lv` option. This option loads the variables defined by the part of a MASM.EXE listing for a single segment. The program scans for the word "segment" and sets up program variable names up to the corresponding **ends** pseudo op.

The `ll`, `lm`, and `lv` options use the user program area to load in the .MAP and .LST files and hence overwrite whatever program might have been loaded in. Hence to debug a program, load in the label files first, and then the program.

LIST Command

The command

▸ **list** [*address*]

unassembles the code starting at the address *address* if specified, at 100h if a .COM file, or at the initial **cs:ip** if an .EXE file. The command goes into the unassemble full screen mode automatically.

LLIST Command

The command

▸ **llist** [*address*]

acts as the **list** command and echos the output to the lister.

LOAD Command

The **load** command is used to load in an SST Program saved by the **save** command. See Chap. 8 for details.

LONG Command

To facilitate both source-level and assembly language debugging, SST includes commands to define typical data types. The syntax for the **long** type is:

▸ **long** ⟦⟦far⟧ *⟧ *variable name address*

For example,

```
▸ long alpha 305
```

adds the symbol `alpha` of type **long** (32-bit signed integer) to the segment specified by the **ds** segment register at the offset 305h. You can specify any other segment. The optional * generates a near **ptr** to a variable of the type **long**. The optional **far** generates a far pointer to a variable of the type **long**.

M Command

This command moves a block of memory from one location to another (like DEBUG). The command expects original start address, original end address, and destination start address as arguments. Syntax:

▸ **m** *range address*

For example,

```
▸m2200,2280,1000
```

moves the contents of memory contained in the block **ds**:2200h through **ds**:2280h (inclusive) to **ds**:1000h through **ds**:1080h. If the original and destination memory blocks do not overlap, the original memory block is left undisturbed. However the two memory blocks can overlap with no ill effects. For example,

```
▸m2200,2275,2203
```

moves the contents of **ds**:2200h through **ds**:2275h up by three bytes in memory to **ds**:2203h through **ds**:2278h.

This command moves a maximum of 64K in any one move. As elsewhere, segment values can be used to override the default value in **ds**:

```
▸mcs:2200,2275,es:2203
```

moves cs:2200 through cs:2275 to the block starting at es:2203.

MAP Command

MSDOS 2.0 and later has a linked list distributed throughout the low 640K bytes of RAM that defines how the RAM is allocated. This list consists of 5-byte paragraph-aligned entries. These entries immediately preceed the memory block (also paragraph aligned) that they describe. The first byte is an "M" for all control blocks except for the last, which has a "Z" for the first byte. MZ are the initials for one of the principal architects (Mark Zibowsky) of MSDOS, and also appear as the first two initials in an .EXE file. The second and third bytes give the 16-bit paragraph that owns the block, and the fourth and fifth bytes give the length of the block in paragraphs.

Specifically, the command

⊦ **map**

displays the five bytes of all DOS memory-control-blocks each followed by a two-line display of the start of the memory block they describe.

MOUSE Command

The command

⊦ **mouse** *status*

turns mouse control on if *status* = on and off if *status* = `off`. The mouse is used to control cursor motion for the FONT, DISPLAY, and TRACE modes.

MSW Command

The command

⊦ **msw** *n*

exclusive or's the value *n* with the SST byte that controls SST Microsoft Windows debugging. See Chap. 13 for details.

N Command

This command names a file for reading or writing using MSDOS **int 21** routines (like DEBUG). Syntax:

▸ **n** *filespec*

names the file given by the file specification *filespec*. For example,

```
▸ nmyfile.exe
```

sets up the load command to be able to read in the file MYFILE.EXE.

The name command stores the name at 81h in the program prefix as does DEBUG.COM and sets up the first and second filenames at the 5Ch and 6Ch File Control Block areas in the program prefix. Note: when you load a program and then go to it, it may use its own name unless you issue a second name command to rewrite the one used to load the program. For example, if you load PS's editor module, PS.COM (a version of PMATE.COM with menu macros), and then go to it, it will open a file called PS.COM, which looks bizarre to say the least, since it's a binary file. To avoid this, type `n` ↵, which gives no parameters, or type `n` followed by the desired parameters. `n=` displays the current command line.

Saving Display and Unassemble Output to File

To define a file for output from the display and unassemble commands, type

▸ **n>** *filespec*

This sets up the file given by *filespec* to receive display and unassemble output when those commands are typed with a range, e.g., `d20,30` or `u100 l 30`. If the filename already exists, you are asked the question, "Overwrite existing file (Y/N)?" In this case, the file is opened for display and unassemble output if and only if you type `y` or `Y`. The `n>` output can be toggled on and off by typing

```
▸ n>
```

i.e., without a filename. If `n>` echo output is enabled, a ◼ shows up at the end of the second line from the top of the screen. If you attempt to toggle the file echo on without having defined a file for output, you'll see the message

```
Echo file undefined
```

Defining User Strings

To define a user string, type a command of the form

▸ **n***st*="*string*"

This defines (names) the string *st* (two letters long) to have the value *string*. User strings are used for examine memory templates (see e command) and keyboard macros as described in Chap. 4.

This command is upward compatible with DEBUG.COM's and adds file echo capabilities.

NEW Command

The **new** command restores the registers to the values used upon running SST, deletes all labels, and enables demostration and program facilities. Essentially **new** returns SST to its initial state.

NMI Command

The command

```
▸ nmi status
```

enables (*status* = on) or disables (*status* = off) NonMaskable Interrupts on return from SST to the user program.

NOT Command

The command

```
▸ not range
```

not's all bits in the bytes in the range *range*.

O Command

This command allows you to output any value to an output port. Type o or O followed by the port number and the desired value to output in hex. Syntax:

```
▸ o portaddress list
```

For example,

```
▸ o20,7F
```

outputs the value 7Fh to output port 20h. The list can have many bytes given by combinations of hex and quoted values. Hex values larger than 0FF are treated as word values.

Output to ports 3F8 and 2F8 wait for the TRHE bit (bit 5 of port 3FD and 2FD, respectively) to go high, indicating that the serial port is ready to transmit.

This command is upward compatible with DEBUG.COM's, and includes the ability to output words at a time and to handshake on the IBM PC serial ports.

OPCODE Command

The command

▸ **opcode** *n*

displays the (or a) mnemonic for the byte opcode given by *n*.

OR Command

The command

▸ **or** *range list*

or's the bytes in the range *range* with the bytes in the list *list*. The *list* is repeated as often as necessary to cover the complete range. This command is similar to the fill command, but **or**'s the list into memory rather than overwriting the bytes in memory.

P Command

The **p** command allows you to protect memory from being referenced in continous and quiet tracing. It has the syntax

▸ **p** $address_1$ $address_2$

where both addresses can have segment specifications. This allows up to the full megabyte of memory to be protected. If $address_2$ is missing, only $address_1$ is protected. In the continuous and quiet trace modes, if protected memory will be referenced by executing the next instruction, the trace halts and the message "Protected Memory Referenced, Continue Trace (Y/N)?" appears. If you type y or Y, the trace continues; else COMMAND MODE takes over. To turn memory protection off, type the p command with no addresses.

If you can type appropriate commands for Super-Trace, you can catch an undesired memory reference much faster than with this mode. If you can get near the bad reference with breakpoints, the continuous traces may give you just what you want.

PAGE Command

The **page** command is used to control page protection on 80386-based machines running in protected mode. See Chap. 12 for details. See also the **p** command.

PAUSE Command

The `pause`*n* command pauses a time proportional to *n* whenever a Ctrl-\ is encountered in the keyboard input. This is used to pace demonstrations such as that invoked by the A option on Function Key 7.

PP Command

The **port protect** command

‣ **pp** *n* [*status*]

protects the port *n* from being read or written when running in 80386 VM mode (see Chap. 12 for details). The optional *status* turns port protection on (*status* = `on`) or off (*status* = `off`).

PROMPT Command

The **prompt** command changes the usual COMMAND MODE prompt to a DOS-like pathname prompt. For example, if you're in subdirectory \BIN on drive C, the COMMAND MODE prompt becomes

```
C:\BIN>
```

This can be a bit confusing if that's the same prompt you use for DOS itself, but in any event you see the register window at the top of the screen, indicating that SST is active.

Q Command

This command is used to return to DOS and also to define a number of SST system parameters controlling screen attributes and other machine characteristics. To return to DOS, type

```
▸ q
```

which asks if your want to leave SST. If you type `y` or `Y`, you get back to the DOS system command prompt, just like DEBUG.COM. You can also return to DOS by typing any of the commands **quit**, **bye**, or **system**. These commands do not require confirmation.

To return resident, i.e., with SST available by typing Ctrl-Enter or by pressing an NMI button, type

```
▸ q/R.
```

If you want the user screen to be saved in this mode, be sure to use the `qs3` command below before typing the `q/R`.

Screen Characteristics

Various screen characteristics are defined by the following `q` commands:

• `q`*c n*	Set screen attribute for window *c* = a, h, n, r, s, x, z, for Assemble, Help, Normal, Register, Stack, Xam, Zam
• `ql`*n*	Set lines/page = *n*
• `qy`*n*	Set # lines Xam window = *n*

To see an example of configuring the colors attributes for various SST windows, see the section "Configuring SST" in Chap. 3

The `qn`*n* command is used to set the normal screen attribute and also to control screen "snow" for IBM color/graphics type displays. SST automatically recognizes the IBM color/graphics display and eliminates most of the snow. `qn80`*xx* sets the normal screen attribute to *xx* and suppresses deglitching (good, e.g., for COMPAQ, which doesn't have snow). This speeds up screen displays by about a factor of three. `qn40`*xx* forces deglitching (good for AT&T 6300)

The `qs`*n* command with *n* missing or less than 4 is used for various screen save and switching options described below. To set the stack window attribute to a number less than four, set the high bit of the word to 1. For example to set the stack attribute to green on a black background, type

```
▸ qs8002
```

Screen Save

SST's screen saving facility is described in Sec. "Screen Save Option" in Chap. 3. The command

```
▸ qs3
```

turns on the screen save option if enough room is allocated (see `sst/`*n* option in the "Command Line Parameters" section of Chap. 3 to increase this allocation).

The command

```
qs2
```

turns off the screen save feature. The COMMAND MODE or TRACE MODE V option switches to the saved user screen. Typing any character thereafter returns to the mode before the V option was chosen.

Where to Display SST

SST display output can be sent to many different places to allow maximum flexibility in debugging programs that themselves use the screen. The options allow you to display SST output in various parts of a given screen (particularly useful on screens larger than 25 lines), on different screens, and in aribitrary parts of memory for use with nonstandard video RAM and multitasking window programs like TOPVIEW (note SST doesn't currently run under TOPVIEW, but we hope to make it do so soon).

To set the origin of the SST display output for a given screen to line *n*, type

```
▸ qon
```

To display SST output in the lower half of 66 line screen, type

```
▸ qol
```

To swap IBM monchrome and color/graphics displays for SST alone, type

```
▸ qs
```

To swap IBM monchrome and color/graphics displays for both SST and DOS, type

```
▸ qs1
```

SST choses the screen RAM segment by consulting **int 10h** on the IBM PC machines. To overrule this choice, you can set the segment to the paragraph *n* by a command of the form

```
▸ qgn
```

Similarly the 6845 CRT controller port is chosen according to **int 10h** information, but can be overruled by a command of the form

‣ **qp***n*

which sets 6845 CRT I/O port = *n*. The IBM PC monochrome display has the value 3B4h and the color/graphics display has the value 3D4h, both of which are recognized automatically by SST. To configure on some other machine, e.g., the Toshiba T300, you need this command (e.g., `qp90`)

Interrupt Mask

The command

‣ **qi***mask*

sets the interrupt controller mask to the value *mask*. This is useful in debugging multitasking systems in which the system clock might be used to switch tasks after SST gains control. For example, to allow only keyboard interrupts use `qiFD`.

Undercover (Periscope) Debugger

To allow the periscope undercover hardware debugger to gain control on receipt of a NMI interrupt instead of SST, type the `rn` command to return the NMI interrupt to the vector active before SST was loaded and type a command of the form

‣ **qu***port*

which instructs SST that the Periscope NMI button port is the value *port*.

QUIT Command

The command

‣ **quit**

returns to DOS. **bye** and **system** do the same thing.

R Command

The **r** command allows you to examine and change the contents of the 8088 registers and flags as for DEBUG.COM. Under SST the r command is basically useless, since the register and flag value are always displayed in the register window at the top of the screen, and the values can be changed by more easily by simple assignments like

```
ax=100
```

The usual DEBUG command

```
▸ r
```

displays the current register values in the COMMAND window. This is useful for echoing the results of a debug session to a file or printer.

Changing Register Values

As for DEBUG,

▸ r *register*

displays the contents of the register *register*. Entering a new value *nn* from the keyboard enters a new value for the register. For example, entering `rdx` when dx=0000 results in the display

▸ `rdx` 0000-

If *nnnn* is now typed, dx will have the new value *nnnn*h. If you do not want to modify the value, hit the space bar to display the next register or hit return to terminate the command.

Alternatively, the command

```
▸ register=value
```

sets the register *register* equal to the value *value*. Valid register names are **ax, bx, cx, dx, al, bl, cl, dl, ah, bh, ch, dh, si, di, bp, sp, eax, ebx, ecx, edx, esi, edi, ebp, esp, ds, es, cs, ss, fl,** and **ip**.

With SST the 8087 floating point stack values can be changed by typing

```
▸ s n=value
```

where s*n* can be s0 through s7.

The flags Auxiliary Carry, Carry, Parity, Sign, Zero, Direction, Interrupt Enable, Overflow, and Trap can be set to 1 or reset to 0 by typing their leading letter followed by **f** as in

```
▸ cf=1
```

which sets the carry flag to 1.

The register command is upward compatible with DEBUG.COM's. Its display differs in that the most recent values alone are always displayed at the top of the screen. This approach is much easier on the eyes than DEBUG.COM's, which constantly scrolls the screen. Note that in TRACE MODE, SST can retrace up to 20 steps (or more - see "Command Line Parameters" in Chap. 3), allowing you to see earlier values of the registers.

Real Mode

SST can run in the 8086/8088 Real Address Mode or on the IBM PC AT in the 80286 Protected Virtual Address Mode (see Y286 command in this chapter). To return to real mode after running in Virtual Mode, type the command

```
▸ rm
```

This restores SST segments to those appropriate for a .COM file.

Restoring Registers and NMI Interrupt

To restore registers to their values when the last .COM or .EXE file was loaded, type the command

```
▸ rr
```

This is handy for rerunning a program after examining how it terminated.

To return the NMI interrupt vector back to the program used before SST was loaded, type the command

```
▸ rn
```

We find this handy for debugging SST itself, and it is also useful for tricky situations when a special hardware debugger has advantages over a software debugger.

RAM Command

The **RAM** command returns the display source to RAM from disk as established by the disk command. See Chap. 9 for details.

RAMFONT Command

The **ramfont** command sets the Hercules Graphics Card Plus RamFont control byte. See Chap. 10 for details.

REDIT Command

The **redit** command puts the cursor into the register window, where you can overtype register values. The facility is also available in TRACE MODE by typing the "e" hot key.

REN Command

The command

⊦ **ren** *file*$_1$ *file*$_2$

renames the file *file*$_1$ to *file*$_2$ like the corresponding DOS command.

RUN Command

The **run** command restores the registers to their initial values and transfers control to the program entry point.

S Command

This command searches for a string of characters or bytes (like DEBUG), or for a string of assembly language instructions. Syntax:

```
▸ s range list
```

where *list* can be composed of one or more strings of the form "..." and bytes consisting of one or two hexadecimal digits. For example with ds=1234,

```
▸ s100 4000 1A 3E "abc"
```

would display

```
1234:856  A0C  FFE  3254
```

if the string of five bytes 1A 3E 61 62 63 starts at the locations 1234:856, 1234:A0C, etc. If the string of bytes is not found, the message

```
String not found
```

is displayed.

For convenience, two abbreviated forms of the search command are included. Typing s or S with a range only searches that range for the last string entered. This allows you to change the range of your search. Typing s or S alone repeats the last search. This is handy after the original search hits are scrolled off the screen.

The search command quits with Ctrl-C and pauses with Ctrl-S.

Searching for Assembly Language

To search for assembly language instructions, type @ ↵for the list field. This leads to the same kind of display and entry as given by the assemble command. The list of instructions is terminated by a hitting ↵ twice as for the assemble command, and is followed by a display of all addresses (if any) where the instructions given are found. For example,

```
▸s cs;0 1000 @
1111:0088 mov       ax,1
1111:008B push      ax
```

searches the memory from **cs**:0 to **cs**:1000 for the instructions that push a 1 onto the stack (with the SST running on an 80186 or 80286-based computer, you can type push 1 for this, but that might not correspond to the code you're searching). Here the 1111:0088 is a sample starting location of SST's search string memory. SST's data segment value 1111: will almost certainly be something different when you run SST.

Being able to search for assembly language mnemonics is very useful for debugging programs consisting of many separately assembled modules, since you typically only know the addresses of code relative to the module origins.

Searching for Jumps/Calls to Location

SST also allows you to find all jump and call references to a particular program offset within a range of memory. Type

▸s *range* j *n*

This lists the offsets (relative to the segment register given by *range*) of instructions that jump to the offset *n*. The instructions checked for are: near/far direct **call**, short/near/far direct **jmp**, the 17 conditional jumps like **jz**, and the three loops. Indirect jumps and calls are not checked.

This option differs from the corresponding DEBUG option in that twelve addresses are displayed per line instead of one, so as to use up less screen display, and in its ability to search for assembly language instructions.

SAVE Command

The **save** command is used to save an SST Program as a COM file with a special header allowing labels and comments to be saved as well. Subsequently, such COM files can be loaded in by the **load** command. See Chap. 8 for details.

SNOW Command

The infamous Color/Graphics Adapter snow can be controlled by the command

▸ **snow** *status*

which turns CGA snow check on (*status* = on) or off (*status* = off).

SYSTEM Command

The command

▸ **system**

returns to DOS. **bye** and **quit** do the same thing.

T Command

This command traces program execution with full screen displays (unlike DEBUG). Syntax:

▸ **t** [*n*]

turns on the TRACE MODE. If *n* is missing, the trace starts at cs:ip; if *n* is present, the trace starts at *n*. TRACE MODE displays the registers as for the **r** command followed by the menu

```
TRACE MODE: F1 Break # Go Cont D!Fast Slow
        Jmp Nop Re Win T!Xam 8087
```

and a screenful of machine language and disassembled instructions starting at the starting trace address. The current instruction line is highlighted by a reverse video bar. Each depression of the space bar single steps the program. Typing Function Key 1 displays the help screen

Break	Break at IP	# *n*	break at IP after *n* passes
Go adr	break at adr	Line	break at next source Line
Here	break at cursor	Point	toggle break-Point at cursor
Fast	break following IP	Slow	trace IP
Space	Single step	Don't	call single step
Undo	last instruction	Nop	skip IP
Jmp	Jump unconditionally	Kick	IP to cursor
Quiet	trace	Cont	Continuous trace
@scii	display ASCII screen	+-&	source/asm/mix
Ice	stack	Offset	change stack readout offset
Overtyp	stack/xam window	Asm	at cursor
Edit	registers	Z	8087 status
ASCII	toggle ASCII vs HEX	Txam	toggle Tracking
F5	zoom window	F6	change window
↑↓	scroll window	PgUp/Dn	scroll window
W/X	toggle Program/Xam	View	program window
2/7/8	toggle menu/8087/stack	0/1/3	no/8086/80386
Redraw	screen	↵	→ COMMAND MODE

TRACE MODE Hotkeys

In alphabetical (ASCII) order, the hot keys are defined as follows:

Ctrl-A toggles ASCII vs hex/ASCII display modes in the memory eXamine window.

Ctrl-B forces hex/ASCII (Byte) display mode in the memory eXamine window.

Ctrl-D moves the top bar of the program-output down if the cursor is in the program-output window. If the cursor is any other window, the memory eXamine window's top bar is moved down.

↵ returns to COMMAND MODE

Ctrl-O zooms the cursor's window directly into OVERTYPE MODE. Typing the Esc key, the Enter (↵) key, or Function Key 5 again returns to TRACE MODE.

Ctrl-U moves the top bar of the program-output up if the cursor is in the program-output window. If the cursor is any other window, the memory eXamine window's top bar is moved up.

Space single steps the program, and ↵ returns to COMMAND MODE.

> Note: on 8088's manufactured after 1981, single step doesn't stop until two instructions after a segment modification instruction like `mov ds,ax`. On the 80386, single-step may step two iterations of a rep string instruction.

# displays

```
    # iterations =
```

to which you type the number of times, *n*, the current instruction should be allowed to execute before breaking (equivalent to typing *n* **B**'s for break).

&-+ switch between assemble/source modes. - switches to pure assembly mnemonics, + switches to pure source code, and & displays mixed source and assembly mnemonics.

0-8 toggle the display of SST windows as follows:

> **0** - toggle register window
> **1** - set 16-bit register display
> **2** - toggle menu window
> **3** - 32-bit register display
> **7** - toggle 8087 window
> **8** - toggle program stack window
> **x** - toggle Xamine window
> **w** - toggle program output window

If option 7 is used with no 8087, the error message "No 8087 installed" is displayed. The 8087 condition codes and register stack are displayed below the program stack. See the zamine command for a complete description. Since ordinary decimal and scientific notation is used, this display makes debugging 8087 code fairly easy (with DEBUG.COM it's essentially impossible).

@SCII displays the ASCII help screens (see Chap. 2 for full description).

Assemble (a or A) switches to ASSEMBLE EDIT MODE at address at the cursor. The Enter key enters the current instruction and goes onto the next. The Esc key returns to TRACE MODE without entering the current line.

Break executes the instruction at **cs:ip** and then sets a breakpoint the execution on the next encounter (RAM only).

Continuous runs continuously until a key is depressed. The continous trace stops if protected memory (see protect command) is referenced, or if an illegal op code is encountered, or if an instruction for a higher-level machine is attempted, e.g., running a **pusha** (push all) on an 8088.

Don't single-step calls executes call instructions at full speed by setting a breakpoint following the call instruction. On other instructions it simply single steps like the space bar. This differs from the Fast hot key, which executes any current instruction at full speed by putting a breakpoint after that instruction. The don't hot key allows you to see the general flow of a routine without getting sidetracked down subroutines. The Don't option marks the first 10 subroutines it encounters as Don't-trace subroutines. These subroutines can be returned to trace mode by using the Slow option.

Edit (e or E) switches into the register window and allows you to overtype register values, and the Zero, Carry, and Sign flags. Use the arrow and tab keys to move around the window. Type the Enter key to enter the new values and continue tracing. Type the Esc or Ctrl-C keys to suppress the new values and continue tracing.

Fast executes the current instruction at full machine speed, breaking when encountering the instruction after the current instruction (RAM only). This is useful for calling a subroutine or finishing a loop instruction without single-stepping through it. Note that if the current instruction is a jump, the fast hot key may amount to a go.

Go *address* sets a breakpoint at the address *address* while remaining in TRACE MODE. This saves the effort of returning to COMMAND MODE and then to TRACE MODE when you know the breakpoint address you need. In addition, **G*s**, **G*b**, **G*c** set temporary breakpoints at [sp], [**bp**], and offset at stack-window cursor, respectively and go. The stack values used are near addresses. To specify far addresses, use **G*fs**, **G*fb**, and **G*cb**, respectively. **G*s** is useful for breakpointing upon returning from a subroutine. Be sure that the return address is in fact at [sp]. Another way to return from a subroutine if you haven't executed too far into it is to Undo back out of the subroutine and Don't call around it.

Here sets a temporary breakpoint at trace-window cursor position and goes. This hot key is available in UNASSEMBLE MODE as well.

Ice *ices* the stack readout offsets at their current values. This iced mode is indicated by reverse video offsets.

Jmp causes an unconditional jump (useful for overruling a conditional jump) to cs:**ip+(ip+1)**.

Kick *kicks* the Instruction Pointer to the address at the cursor. Only the instruction pointer is changed by this hot key. This hot key is available in UNASSEMBLE MODE as well.

Line single-steps with no screen updated until the next source-code line is encountered.

Nop skips the next instruction altogether (but doesn't change the code).

Offset cycles between stack offset value modes. These offsets can be made relative to ss:0, to **sp**, or to **bp**. Stack segment displays show all stack frames in range in reverse video and a referenced location in bold. This feature also works in DISPLAY MODE when the segment displayed is the same as that given by the stack segment register ss. SST also bolds the target offset of a conditional jump that will jump.

Point toggles the sticky breakpoint at the cursor position. This hot key is available in UNASSEMBLE MODE as well.

Quiet toggles quiet continuous trace. This mode only updates the registers and runs about three times as fast as the Continuous trace. The quiet trace stops if protected memory (see protect command) is referenced.

Re Redraws the screen with the currrent instruction at the top. This is handy if the current instruction is displayed at or near the bottom of the display and you want to see the following instructions.

Slow single-steps the next instruction, which lets you trace execution of an int instruction (normally executed in Fast mode).

Txam toggles the Tracking feature of the eXamine window. When tracking is on a T appears at the end of the second line from the screen top. The eXamine window always displays the memory around the last location referenced by the program, and the cursor identifies this location. This is a useful feature and leads to fascinating demos when run continuously.

Undo Undoes the last single step. This can be repeated up to 20 times (or more - see Chap. 3), literally allowing you to see your program execute backwards. This is very useful for recalling the steps that lead to an anomalous condition.

View switches to the user screen if the screen save option is enabled (see the `qs3` command)

Win toggles a Window 15 lines down from the screen top for MSDOS CRT output. This is handy for debugging routines that write a moderate amount of text to the screen using standard system calls.

Xam toggles a two-line memory eXamine window at the bottom of the screen. This window displays 20h bytes in hex/ASCII format and 80h bytes in pure ASCII format. If the window is not in tracking mode (see next option), you can scroll through memory using the arrow keys and the PgUp PgDn keys. The cursor is displayed at the last location referenced in the window.

Z gives you a pop-up full screen of information about the 8087 status. Typing `x` or `X` when this Z screen is present shows you the heX values of the 8087 floating point registers, regardless of whether they are tagged "empty", or invalid.

Trace Mode Window Control

Function Key 5 zooms the stack/xamine windows into DISPLAY MODE. This gives a full screen with full DISPLAY MODE features including overtype capability. The Ctrl-O hot key zooms these windows directly into

OVERTYPE MODE. Typing the Esc key, the Enter key, or Function Key 5 again returns to TRACE MODE.

Funtion Key 6 switches between windows in TRACE MODE. The window with the cursor can be scrolled up and down with arrow, PgUp, and PgDn keys. When in the trace window, the up and down-arrow keys are useful in combination with the Assemble, Here, Point, and Kick hot keys. To toggle windows on and off, see the 0-8 hot keys.

Super-Trace

Section "Super-Trace Demonstration" of Chap. 3 describes a special SST$^+$ facility called Super-Trace. This facility single steps a program in a very tight loop, executing a set of user-specified conditions after each single step. These conditions are written in ordinary assembly language and are assembled by the assemble command module. The requirements for the code are given under the Conditional Breakpoint section of the go command. Basically **ax** and **bp** are saved before entering the user code, and **bp** points at the program stack and **ax** has the first word of the current instruction. No return instruction is necessary, since SST$^+$ automatically supplies the return. If the code sets the Zero flag, SST$^+$ takes over, allowing the user to examine the machine. If the Zero flag is reset to 0, the Super-Trace continues. Typically Super-Trace runs at about one tenth full machine speed, although this depends markedly on how much code the user specifies for conditions. If a whole execution profile routine is called, execution could easily be slowed down another factor of ten.

TIME Command

The **time** command displays the current time of day as calculated by the computer.

TRACE Command

The **trace** command restores the registers to their initial values and goes into TRACE MODE at the program start address.

TYPE Command

The SST **type** command displays a file in a full-screen menu-driven mode that scrolls forward and backward with the PgUp, PgDn, and up and down arrow keys, goes to the start/end of the file with the Home/End key, toggles the display of line numbers with the # key, goes to line *n* with the Line option, and searches forward or backward for an arbitrary literal string.

To display the file *filename* in this mode, type a command of the form

▸ **type** *filename*

Typing the command

```
▸ type
```

with no argument displays the file previously typed at the same location that you left it. To leave the TYPE MODE, press the Esc key.

U Command

This command unassembles machine-language instructions. Syntax:

▸ u *address*

unassembles the instruction at the address *address* and goes into UNASSEMBLE MODE. If *address* is missing, the instruction following the last one unassembled is unassembled, or if no previous unassemble command has been executed the instruction at **cs:ip** is used. Typing the space bar unassembles the next instruction. Typing a PgDn unassembles a whole screenful (except for the register window at the top of the screen). PgUp does a rudimentary Page Up procedure that subtracts a number of bytes from the current unassemble address and unassembles a screeful from that lower address. To be sure you're properly synchronized, type a few space bars. ↵ and Esc go back to COMMAND MODE. The unassemble display format is the same as that for the assemble instruction illustrated above.

Alternatively

```
▸ u range
```

unassembles the instructions within the range specified. This version of the unassemble command works as for DEBUG.COM, while the other two options are different. The output of this start/end unassemble option can be written to a file of your choice (see n> filename command).

USE16/32 Commands

The commands **use16** and **use32** control the 80386 segment D bit of the assembler. The default is **use16**, which generates code according to the standard 8086 segment addressing. The **use32** command switches to the 80386 32-bit addressing mode, which allows access to 4-gigabyte segments, greatly increased indexing facilities, etc. See the Intel *Programmer's Reference Manual* for a detailed discussion of these modes.

V Command

The v command alone (followed by a ↵) swaps to the user screen if enabled by the qs3 option. Alternatively the V option in TRACE MODE flips between SST and user screens (on the same monitor).

When followed by a hexadecimal number, the v command calls an interrupt vector. Syntax:

```
▸ vn [ax [bx [cx [dx]]]]
```

where *n* is the desired interrupt vector number, and the indicated registers are assigned values optionally. Current register values are used for values not given on the command line.

For example,

```
▸ v21,600,,,7
```

rings the bell, since an **ah=6 int 21** instruction outputs the character in **dl** (here 7) to the system console. After returning from the vector command, the user **ax**, **bx**, **cx**, **dx**, and flags are updated to show what called interrupt vector did.

The vector command saves and restores the user screen if the save screen option is enabled by typing qs3 in COMMAND MODE.

Virtual Mode

On the IBM PC AT compatible computers and 80386-based computers, you can enter the 80286/80386 Protected Virtual Address Mode by the command

```
▸ vm
```

and return to Real Address Mode by the command

```
▸ rm
```

The Protected Virtual Address Mode gives you direct access to the 80286's entire 16 megabyte address space as well as to various protected mode features. You can use SST to debug .COM files in this mode, run programs in extended memory up above the Real Address Mode's megabyte, examine memory in extended memory, and so on. See Chap. 12 for discussion of special 80386 protected mode operation.

The user program can change the Interrupt Descriptor Table origin from what SST sets up, but must leave the IDT descriptor in the GDT at offset 90h.

The SST vm command sets up user stack selector = 60, data selectors = 68 and code selector at 70. This works for .COM and .EXE files.

The Ctrl-Enter key interrupts vm operation as in Real Mode and Ctrl-Alt-Del works. In addition sticky breakpoints work. DOS commands in general work in the 80386 DOS extender protected mode (see Chap. 12). In other protected modes, the DOS **dir**, **chdir**, **prompt**, and **type** commands work, although not in a completely general way. After typing vm in COMMAND MODE, which switches to Protected Virtual Address Mode, typing any one of these DOS-like commands invisibly switches back to Real Mode, calls needed DOS commands, displays the desired information, and then switches back to protected mode. The screen pretends that protected mode is always enabled (V at lower right of the register window).

In all protected modes, SST's built-in keyboard (**int 9**) program is used. The time-of-day clock is also turned on to convince you that the machine is still running.

The SST protected mode tracks 80286 Real/Virtual Mode on all trace/breakpoint options. Hence if you leave SST in Real Mode, and SST traps an interrupt occurs in protected mode, SST automatically switches itself to protected mode. Similarly if you leave SST in protected and reenter in Real Mode, SST switches itself to Real Mode operation. This facility is required for operation under OS/2.

W Command

This command writes a file or absolute disk sectors (like DEBUG). Syntax:

▸ w [*address* [*drive sector*$_1$ *sector*$_2$]]

If the drive and sector specifications are missing, it writes the file named by the name command (see above) at the address specified on the l command line. If the address is missing, the file is written starting from cs:100.

.EXE files can also be written provided they are read in first. This allows you to patch an .EXE file. Be sure not to change the segment specification values inadvertently.

The following error messages can occur:

No room in disk directory
Insufficient disk space
Insufficient memory
Error in .EXE file
Read .EXE file before writing

Write is very useful for modifying a disk file. Name the file with the name command, load it with the load command, make the changes you want being sure not to change the values of the **bx** and cx registers, and then type w or W to Write the modified version back to disk.

Write Labels

The labels, variable names, and user strings currently defined can be written to the file named by the last n command by typing

```
▸ wl
```

See Sec. "Labels" in Chap. 4 for further information.

WIDTH Command

The command

▸ **width***n*

sets the screen width to *n* = 40 or 80. The value 40 is nice for big room demonstrations, but is not able to display all features of SST.

WORD Command

To facilitate both source-level and assembly language debugging, SST includes commands to define typical data types. The syntax for the **word** type is:

▸ **word** [[[far]] *]] *variable name address*

For example,

```
▸word alpha 305
```

adds the symbol `alpha` of type **word** (16-bit unsigned integer) to the segment specified by the **ds** segment register at the offset 305h. You can specify any other segment. The optional * generates a near **ptr** to a variable of the type **word**. The optional **far** generates a far pointer to a variable of the type **word**.

X Command

The **x** (eXamine) command sets up the display of 20 hex locations used primarily in TRACE MODE. Syntax:

▸ x *address*

causes bytes starting at hexadecade that includes the address *address* to be displayed and updated automatically in TRACE MODE. When this command is executed, the two-line window immediately shows up at the bottom of the screen and the menu line changes to

```
XAMINE MODE:  ←↑↓→  PgUp  PgDn
```

In this mode, the Xamine window is continously updated many times a second, allowing you to monitor input from interrupt driven devices like the system clock and keyboard. For example, on an IBM PC, type

```
▸ x40:6C
```

and watch the clock tick away. Scroll up in memory to see the keyboard input queue change as you type the right arrow. In either XAMINE MODE or TRACE MODE, the up and down arrows scroll the memory display up (towards smaller memory addresses) and down respectively. The PgUp and PgDn keys scroll up and down by 100 hex at a time.

XOR Command

The command

▸ xor *range list*

xor's the bytes in the range *range* with the bytes in the list *list*. The *list* is repeated as often as necessary to cover the complete range. This command is similar to the fill command, but or's the list into memory rather than overwriting the bytes in memory.

Y Command

Development of protected-mode applications requires a careful understanding of how the Global and Local Descriptor Tables work. In SST Protected Modes, the segment values for ds, cs, es, and ss, along with many internal values have be changed to correspond to values in SST's vm Global Descriptor Table (GDT). Notice after executing the virtual mode command that the segment register values are relatively small numbers. These correspond to entries in the GDT. To read the GDT table, use the y command as follows:

```
▸ y n
```

If *n* is present, list the GDT (or LDT) entry *n*; if not, list 80286 GDT entries one per space bar. RAM and Global Descriptor Table displays beyond the segment limit have the offset field displayed in reverse video.

Defining Global Descriptor Table Descriptors

To define new descriptors to refer to areas of memory of your choice use the command

```
▸ y n address [access [length]]
```

This define a GDT entry 70< *n* <D8 at *address*, access=*access*, with length=*length*. If the *access* and *length* fields are missing, 93h is assumed for the access (writable data segment) and the maximum 80286 segment length of 0FFFFh is used for *length*.

For example, to be able to examine, compare, move, etc. memory starting at the second megabyte in physical memory, type the command

```
▸ y70 100000
```

Then in Virtual Mode, the command

```
▸ d70;
```

will display a full screen displaying this RAM. If you have a VDISK installed, you'll see the VDISK copyright notice.

At present, the SST vm facility is intriguing and useful, but still in its infancy. Hope you enjoy it.

Z Command

This SST+ command displays the complete 8087 state in greater detail than in the TRACE MODE. If you attempt to use the 8087 facility without an 8087, you see the pop-up message

```
No 8087 installed
Better go get one!
```

Trace Mode 7 Option

Typing 7 in TRACE MODE toggles the 8087 window underneath the program stack window. The 8087 window has the 8087 condition codes on top followed by the values of the eight 8087 80-bit registers. They are displayed with **st(0)** on top and with nine decimal places for integers and for typical floating point values, and about five for those requiring exponent notation. For example, you might see a window like

```
1001
__________
 12
 123456
789
 1.25000
000
 -123.456
000
 -1.23456
e-4
  empty
  empty
  empty
```

The 1001 tell you the 8087 condition code bits **c3**, **c2**, **c1**, and **c0**, respectively, which reflect the results of 8087 compare, test, examine and remainder instructions. Other status bits can follow as discussed under "8087 Status Bits" below. The word **empty** means that the corresponding stack registers have not been loaded. Some other special values such as infinity and unnormal are labeled accordingly.

Trace Mode Z Option

For more accuracy in either TRACE or COMMAND MODE's, type z or Z, which gives the full 80-bit values in scientific notation along with some help information. For the example above, you'd see the screen

```
The 8087 is set with projective infinity, full precision, and
round to even

                                     i
stack index = 0      cc=1001        ‾

st(0) = 12                        Status codes:
st(1) = 123456789                   P-Precision exception
st(2) = 1.2500000000000000          U-Underflow
st(3) =-1.2345600000000000          O-Overflow
st(4) =-1.2340000000000e-4          Z-Zero divide
st(5) = empty                                D-Denormalized
                                  operand
st(6) = empty                       I-Invalid 8087 instruc-
                                  tion
st(7) = empty

Letters above the bar indicate normal interrupts, below in-
dicate masked interrupts.  Type x or X for heX display of
registers.

================ Type any key to continue ================
```

Typing x or X replaces the help on the right by the hexadecimal values of the registers. The screen above changes to

```
The 8087 is set with projective infinity, full precision, and
round to even

                                        i
stack index = 0      cc=1001            ______________

st(0) = 12                              0 4002 C000000000000000
st(1) = 123456789                       0 401D 932C05A400000000
st(2) = 1.2500000000000000              0 3FFF A000000000000001
st(3) =-1.2345600000000000              1 4005 F6E978D4FDF3B647
st(4) =-1.2340000000000e-4              1 3FF2 8164EF6DE184EAB8
st(5) = empty                           1 795F DD768A987E5689F2
st(6) = empty                           1 795F DD768A987E5689F2
st(7) = empty                           1 795F DD768A987E5689F2

Letters above the bar indicate normal interrupts, below in-
dicate masked interrupts.  Type x or X for heX display of
registers.

================ Type any key to continue ================
```

Note that the empty registers do have values, although they don't mean anything. The values may be left over from earlier computations. Registers with invalid contents are flagged by "??" which have special hex values identifying the nature of the problem. These special values are easily examined with the zamine X option.

8087 Status Bits

The condition code bits, interrupt request bit, and exception flag bits from the 8087 status word are reported immediately above the register stack. The binary values of the four condition code bits are always displayed at the upper left of the 8087 window. These bits reflect the results of 8087 compare, test, examine, and remainder instructions. The other bits are displayed by letter if they equal 1, and are represented by blanks if they equal 0. A pending interrupt request is displayed as an "i" following the condition codes. The six exception flags are identified by the corresponding capital letters in the following list: Precision, Underflow, Overflow, Zerodivide, Denormalized operand, Invalid operation. For example, you may see a P fairly often, since precision exceptions are not unusual. For a detailed discussion of these bits, please consult one of the Intel manuals on the 8087 or 80287 numeric coprocessors.

This command doesn't exist in DEBUG.COM.

Note that SST has a useful floating-point calculator based on the 8087. See Chap. 5.

Chapter 8
ASSEMBLY LANGUAGE INTERPRETER

SST has a built-in simple assembly-language interpreter. Typically this interpreter mimics the BASIC interpreter, except that it expects assembly language statements instead of BASIC statements. It also differs from previous interpreters in a number of ways, such as having the full power of a screen debugger and using native machine code as the intermediate interpreter language, which can lead to faster programs than those from compilers let alone usual interpreters.

The BASIC-like word commands coexist with the DEBUG-style single letter commands remarkably peacefully. Words like **load**, **save**, **list**, **llist**, **run**, and **delete** are syntactically illegal from DEBUG's point of view, and hence can be used unambiguously directly in SST's COMMAND MODE. The BASIC command **new** is ambiguous, since to debug it means name the file called ew, but if you really want to name such a file you could type n ew, which is not recognized as **new**. A complete list of such command appears under the heading "Interpreter Commands" in this chapter.

To see a demonstration of the interpreter, use the SST Auto demo option given by typing Function Key 7.

Line Numbers

BASIC uses statement line numbers for branching and editing purposes. Similarly the assembly language interpreter instructions are automatically located in memory and can be referred to by their hexadecimal memory offset values. You use these offset values like line numbers to insert, delete, edit, trace, and execute instructions. Since many instructions are longer than one byte, there are many illegal "line numbers" referring to the middles of instructions. The assembly language interpreter tells you if you try to refer to one of these illegal numbers.

To make sure that it knows what's an instruction without undue overhead, the interpreter insists that its code area (code segment) contains only instructions. If you use a **db** or **dw** pseudo op to define variable storage, that storage will automatically be allocated to the program data segment, rather than to the code segment. The interpreter has a very fast algorithm for scanning through a program up to the **end** statement that allows it to check for legal line numbers. This same algorithm is used to insert, delete and overtype instructions, all of which can involve shifting the code up or down in memory. When you make a change in a program, the interpreter *reassembles* the code at about 11,000 instructions a second on an ordinary PC. Actually it doesn't have to completely reassemble the code; it only has to shift the code as needed and update all relative offsets in **jmp** and **call** instructions appropriately.

Labels

The assembly language interpreter allows the use of labels for referring to variables and jump addresses. As you type in or list a program, references to undefined labels are stamped with a "U" to the left of the corresponding machine code. When you resolve these references by typing in a statement with a missing label, the references are filled in. For example, you can type `call alpha`, where `alpha` hasn't been defined previously, and then later type in the subroutine called `alpha`. If you subsequently delete with instruction with the label `alpha`, all corresponding references are stamped as Undefined, until you redefine their target again.

Instructions and Pseudo Ops

The interpreter accepts all instructions in the 8086/80186/80286/8087 repertoire. In addition three pseudo ops are recognized, **end**, **db**, and **dw**. The pseudo op **end** is used to specify the end of the code. Typically you don't have to use the **end** pseudo op, since the interpreter knows where your code ends. However if you want to delete the code from some point through the end of what you've typed in, you can type **end** in sooner. To start over, you should use the word **new** instead, since that also deletes the labels you've typed in.

The **db** and **dw** pseudo ops are used to define program variables and assign them initial values. For example,

```
message  db        "This is a message",0
```

defines memory for the user variable message. The trace command reinitializes all variables to the values given by the **db** and **dw** pseudo ops.

Edit Command

The SST edit facility described in Chap. 4 can be used to edit assembly language instructions. To edit a line, type

```
▸ edit line_number
```

in COMMAND MODE. This switches to ASSEMBLE MODE and automatically calls up the line with the *line_number* (instruction offset) specified. Make the changes you want and type ↵ to go onto the next line. To quit editing, type Esc, which returns to COMMAND MODE.

While in UNASSEMBLE and TRACE MODEs, you can also move the cursor in the trace window to any instruction and switch to ASSEMBLE EDIT MODE by typing the hot key "a".

Interpreter Commands

SST recognizes the following DOS/BASIC-like commands while in COMMAND MODE (see also Chap. 7).

bye	return to DOS
close	close all disk files
cls	clear screen
cont	continue execution at full speed (not tracing)
delete*n*	delete instruction at offset *n*
edit*n*	edit statement at offset *n*
files *template*	list directory with template *template*
insert*n*	insert instruction(s) starting at offset *n*
list [*n*]	list (display) program from start [from offset *n*]
llist [*n*]	print program from start [from offset *n*]
load *file*	load file with filename *file*.COM
new	delete all labels, restore initial registers values
run	run program from start at full speed (not trace)
save *file*	save file with filename *file*.COM
system	return to DOS
trace [*n*]	trace program from start [from offset *n*]

These commands exist in similar forms in DOS or BASIC. SST accepts relaxed syntax. For example, on the **files** command, you can enclose the filename template in double quotes or not as you choose. The **insert** command is added since legal line numbers always correspond to addresses of current instructions. On the other hand, renumbering is automatic, so BASIC's renumber command is superfluous.

In addition to these commands, you have, of course, the standard SST commands, which can also be useful, particularly the `a` (assemble) and `t` (trace) commands.

Notes:

Notes:

Chapter 9
DISK DISPLAY/MODIFY FACILITY

SST has a disk display/modify facility. Essentially the disk display/modify facility works like the memory display/modify facility except that you specify sectors instead of segments. The facility uses a 64K RAM buffer directly following the user program segment prefix, thereby overwriting anything you may have read in there. You can display the sectors in any of the standard SST display formats using the `d` command and scroll through the entire disk if you have enough time.

To switch into DISK DISPLAY MODE, type

▸ `disk` [*sector:offset*]

where the *sector:offset* specification is optional. Leaving it out starts displaying at sector 0, offset 0. To return to memory display mode, type RAM in COMMAND MODE.

Overtyping Disk

You can switch into DISK OVERTYPE MODE by typing Ctrl-O and overwrite the disk image in memory. To update the corresponding sector on disk, type Ctrl-X, which asks if you want to overwrite the sector in question. Type y or Y to confirm the overwrite request and the disk sector will be overwritten.

Pointer Facilities

Some pointer facilities are available to help you move rapidly from one part of the disk to another by using the hierarchical directory structure. These are defined as follows:

Ctrl-P	Display root directory in current display format.
Ctrl-D	Display cluster corresponding to Directory Entry (DE) at cursor or if in FAT to cluster identified by cursor. Saves current disk location so that Ctrl-G returns to this location.
Ctrl-C	Display cluster chain for file described by DE at cursor.
Ctrl-L	List DE in human-oriented form in pop-up window. While this window is active, typing down (up) arrow moves to the next (previous) DE. When pointing at FAT, Link cluster into cluster chain for file chosen by Ctrl-C option.

We recommend displaying the root directory and the subdirectories in pure ASCII format and use the Ctrl-L option to obtain more specific information. When you want to examine a file or subdirectory, position the cursor somewhere on the corresponding Directory Entry and type Ctrl-D.

File Allocation Table (FAT)

The disk space is assigned to files by use of linked chains of clusters stored in the FAT (File Allocation Table). A cluster consists of one or more sectors, 2 on a 360K floppy, 4 on the 20M AT hard disk. The FAT itself starts at sector 1. If you display the FAT in word format on hard disks with more than 10 megabytes or in triple-nibble (dp) format for floppies and smaller hard disks, the disk cluster chain pointed to by the cursor is highlighted. This is both instructive and useful, since you can see how the disk space is allocated. The appropriate display formats are automatically used when the Ctrl-F FAT display command is typed in the DISK DISPLAY MODEs. The offset field is also treated specially to give the cluster value in reverse video, rather than the FAT sector offset.

You can change the cluster allocation if you want to unerase a file or to construct a file from data on the disk. Move the cursor to the directory entry for the desired file, switch into DISK OVERTYPE MODE by typing the Ctrl-O toggle, type Ctrl-C to display its cluster chain, position the cursor at the desired cluster position and type Ctrl-L. To update the disk FAT, type Ctrl-X as described in the Overtying Disk section. You can also modify the chains by overtyping in hex/ASCII mode, but this is hard to decipher, expecially for 12-bit FAT formats (diskettes and smaller hard disks).

The disk display/modify facilility doesn't yet qualify as a full disk utility package since unerasing a file is not yet menu driven. Nevertheless the display capabilities typically exceed other disk utility packages both in sheer speed and in the variety of display formats which include, for example, unassemble and assemble facilities.

Notes:

Chapter 10
RAMFONT EDITOR

SST includes a powerful screen font editor that uses the RamFont features of the Hercules monochrome Graphics Card Plus (GCP model GB112) and of the IBM Enhanced Graphics Adapter (EGA). These enhancements dramatically reduce the effort needed to create new fonts and to debug programs that use RamFont features. The Hercules GCP offers 12 user definable fonts in an 8x16 cell, while the IBM EGA offers 4 fonts in an 8x32 cell. Characters from all 12 fonts can be simultaneously displayed on the GCP, while characters from only two of the four EGA fonts can be simultaneously displayed. SST's RamFont editor works essentially the same way on both cards. In particular, only 16 bytes of the EGA's 32 bytes in a character can be edited and displayed, although rows shifted down below the sixteenth line are preserved. Specifically,

- Font files can be loaded and saved
- Fonts can be easily displayed and modified
- Fonts can be moved and operated on logically
- Hercules RamFont/blink modes can be changed any time

SST's RamFont facilities include those of Hercules' FONTMAN.COM and Victor's EFONT.EXE as subsets, while offering substantially more font visibility and editing power. The embedding of a font editor in a debugger has a number of advantages over other environments, such as using a single move command to turn a small font into a subscript or a superscipt font.

Specifying RAM Addresses

In specifying any address, a leading **t** implies the base RamFont segment (0B400h on the GCP, 0A000h on the EGA) added to the value following the **t** times 100h for the GCP or 400h for the EGA. Hence

```
▸ dt3
```

displays video RAM around 0B700h:0 on the GCP and 0AC00h:0 on the EGA, which is where font t3 resides.

The **t** field can be followed by a colon (or semicolon) and offset as usual.

Any address oriented command can use this feature, including the **load** and **save** commands. Hence

```
▸ nstandard.fnt
 lt7
```

loads STANDARD.FNT into **t7**. We recommend loading the standard font into t7 on the GCP and into t0 on the EGA, since the standard character attributes used by non-RamFont software then automatically look correct.

To facilitate importing standard 16-byte/character fonts to the EGA, a font file exactly 4096 bytes long (256 characters in 16-byte format) read into an EGA font location (t0 through t3) is automatically expanded to the EGA 32-byte/character format. Files written from EGA RamFont RAM are left in 32-byte format.

Note that FONTMAN.COM specifies the character "a" in font 3 as t4:61 rather than t3:61, since FONTMAN numbers fonts from 1 rather than from 0. Although this might be clearer in the abstract, SST's choice is clearer if you want to keep track of where things are in memory and how the hardware is accessed. Just remember that SST starts with font t0 (FONTMAN's t1) and a character with code 41 has top-line offset 410 (fontman's 41). SST's **t** field is the same as the low nibble attribute byte, while FONTMAN's is one higher.

Font Displays

To display a font on the EGA, type

```
▸font [tn]
```

where the t*n* is an optional font specification. If missing, the last display location is used, or on the first font command, the zeroth character in font 0. This causes a 256-character font to be displayed at the right of the screen and the current character within that font to be displayed in the large at the left of the screen. The cursor keys move to other characters, displaying them in the large accordingly. The Home and End keys move to the zeroth and 255th characters in the font. The PgUp and PgDn keys move to the next and previous fonts, respectively.

To display a font on the GCP, you must first be in 48K RamFont mode (the EGA is always in RamFont mode). If not, first switch into this mode by typing

```
▸ramfont 5
```

(or `ramfont 7` for 8-bit wide characters) and then type the desired font command.

The ASCII chart option given by "@" in many modes can also display in different RAM fonts. Type the desired font number.

Copying Fonts and Shifting Characters

The move command can be used to make copies of fonts and to shift them up and down. For example,

```
▸mt2 11000 t3:3
```

moves font 2 to font 3 shifting all characters down three raster lines (use 2000 for the EGA). This feature is handy for making a set of subscripts or superscripts.

You can also copy part of a font by Tagging one end of a block by typing `t` or `T`, moving to the other End and typing `e` or `E`, and then moving to the desired target position and typing `c` or `C` for Copy.

Logic Operations

The logic operations **and**, **or**, **xor**, and **not** are general features of SST+, but are particularly useful in special font manipulations. **and**, **or**, and **xor** have the same syntax as the **fill** command, while the **not** command simply inverts all bits in the range. The syntax is

```
▸ and range list
▸ or range list
▸ xor range list
▸ not range
```

The bytes in the *list* are **and**'ed, etc., with the bytes in the *range*. See the **and** operator in Chap. 7 for an example.

Editing RamFont Characters

Characters are easily edited using the SST font command, which displays the current character in the large on the left of the screen and 256 characters in the small on the right of the screen. The changes you make show up in the large on the left display and in the small on the right display. Both displays have cursors. The cursor on the left is called the cell cursor, and the one on the right is called the font cursor. The font cursor is moved by the cursor-pad keys as described earlier.

You use the cell cursor to specify where you want to make changes in a character. A Microsoft compatible mouse can be used to move the cell cursor and to store a dot or blank at the cell cursor position. To enable the mouse, type the **mouse** command at the DOS prompt, and then run SST and tell SST that a mouse is available by typing the command **mouse** in COMMAND MODE. The mouse also drives the cursor in the ordinary DISPLAY MODE.

Alternatively, you can move the cell cursor with the "cursor diamond" given by numbers on the numeric keypad (NumLock the IBM PC keyboard) as follows:

0	toggle INSERT/OVERTYPE mode
1	move down and left
2	move down
3	move down and right
4	move left
5	toggle dot at cell cursor
6	move right
7	move up and left
8	move up
9	move up and right

A control cursor diamond is also available: Ctrl-Y (up), Ctrl-G (left), Ctrl-H (right), and Ctrl-B (down). Ctrl-A moves the cell cursor to the upper left corner of the font cell.

To store a dot at the cell cursor, type Ctrl-S (for Set) or type a period. To store a blank, type Ctrl-D (for Delete) or the space bar. The space bar moves right one dot after storing a blank. On the font display, you see the corresponding changes in the small. The Set/Delete/Blank meanings are changed slightly in FONT INSERT MODE. To toggle between the default FONT OVERTYPE MODE and the FONT INSERT MODE, type the Ins key. In the FONT INSERT MODE, the Ctrl-S sets a dot at the cell cursor after shifting the remainder of the character row one dot to the right, that is, it inserts a dot. Similarly, the space bar inserts a blank. The Ctrl-D key deletes the dot, shifting the remainder of the character row left. These functions are similar to the OVERTYPE and INSERT MODE's in text editors. The left and right keys of the Microsoft mouse are aliases for the Ctrl-S and Ctrl-D keys, respectively.

In addition to these single dot commands, there are two row and two column commands, and five cell commands. To insert a blank row at the cell cursor, type `h` or `H` (for Horizontal) or Ctrl-L (for Line). To delete the row at the cell cursor, type `x` or `X` or Ctrl-K (for Kill). To insert a blank column at the cell cursor, type `v` or `V` (for Vertical). To delete the column at the cell cursor, type `y` or `Y`.

The cell commands are **b** or **B**, which Blanks the character cell, **f** or **F**, which Flips the character upside down, **m** or **M**, which gives the Mirror image, **r** or **R**, which Reverse videos the character, and **u** or **U**, which Undoes the changes made so far on this character.

The Function Key 1 help screen in the FONT OVERTYPE MODE and FONT INSERT MODE summarizes these functions.

Streamlining Font Loading and Changes

SST's keyboard redirection feature executes files of commands that can load fonts as desired. For example, the file

```
o3BF 3
n\herc\standard.fnt
lt7
lt3
lt5
n\herc\sanserif.fnt
lt0
lt2
lt4
n\herc\small.fnt
lt1
lt6
ramfont 5
blink
mouse
```

turns on both graphics pages (like Hercules HGC.COM full option at the DOS prompt), loads a number of fonts from a subdirectory \HERC into the RamFont, turns on the 48K RamFont mode, turns the blink attribute into a double width attribute, and enables the mouse (if it was enabled at the DOS prompt as well).

You can define the function keys to perform repetitive changes. For example, in COMMAND MODE, typing

```
▸ F9="^k^k^6"
```

sets up Function Key 9 to delete two rows in the current character and then moves to the next character. By holding the Function Key 9 key down, you autorepeat this function, processing a set of characters rapidly.

RamFont Command

To control getting into and out of the Hercules RamFont modes, the RamFont command has beed added. This has the syntax

▸ `ramfont` [*n* [*m*]]

where if *m* is missing, *n*=0 or missing restores the ROM generator with 9-bit wide characters, *n*=1 turns on 4K RamFont, *n*=5 turns on 48K RamFont (*n* is the value sent to the xModeReg described under RamFont registers in the Hercules manual). If only jibberish appears on the screen in 48K RamFont mode, type ↵ `ramfont` ↵. This should get you back to TEXT MODE, where you can see what you're doing while you set up your fonts. If you're running a program under SST (or have a resident SST), you can get into SST by typing Ctrl-Enter, and then type the above. Be sure to enable Hercules half or full page operation before using RamFont mode.

If *m* is present, *m*=15 writes *n* to the underscore register (line 0 to F on which the underscore should appear), and *m*=16 writes *n* to the overscore register. These options imply 48K RamFont operation, although they do not switch into this mode (use **ramfont 5** or **ramfont 7** to switch).

The command

```
‣blink Ø
```

or blink without an argument turns the blink enable bit off. The command

```
‣blink 1
```

turns it on. We turn it off. Whew!

Modifying SST Screen Attributes

Use the q commands (see Chap. 7) to modify the screen attributes and hence character fonts in Hercules 48K RamFont mode. In particular, we recommend loading the standard font into **t7**, since then most everything works immediately without q commands. Load an italics font into **t1** since then IBM underlining displays in italics. Be sure to tell SST to run in 48K RamFont mode if the board has already been so configured, since otherwise IBM bold and reverse video attributes will be used.

Resident SST

Note that a resident SST (include SST/R in your AUTOEXEC.BAT file) can be used at any time to examine fonts, change RamFont modes, etc. It cannot be used to load in new fonts if DOS is busy (unfortunately, DOS is not reentrant). A non-resident SST (usual DEBUG MODE) can be used to load and save fonts at anytime while debugging another program. To get into SST while running another program, type Ctrl-Enter or push an NMI button. This button is a normally open push button switch connected to the top and bottom pins closest to the rear of the computer on any I/O channel connector. If you make modifications in your characters that you want to keep, you can load a nonresident SST to save the desired updated fonts from the font RAM.

Chapter 11
SOURCE LEVEL DEBUGGING

SST has the ability to display and single-step source code for programs provided source code line numbers are available from the map file or from the .EXE file compiled and linked with the Microsoft CodeView options. Single stepping and breakpointing directly in source code may allow you to home in on an error much more rapidly than working with the assembly mnemonics. At any time you can switch between pure source code, mixed assembly/source, and pure assembly code. Most of the SST TRACE MODE hot keys like Don't call subroutine, Fast, and Continuous trace work in source mode debugging as well. This chapter explains how to prepare .EXE and .MAP files for source level debugging and how the trace and unassemble commands are enhanced to deal with source code.

The **type** (browse) command is also very handy for searching through source files, although it doesn't support setting breakpoints directly in the source code as the trace and unassemble commands do.

Preparing for Source Code Debugging

You can set up an .EXE file for source-level debugging either by loading an appropriate .MAP file, or by loading an .EXE file prepared for use with Microsoft's CodeView debugger. Load the .MAP file with the `ll` command as described under Labels in Chap. 4. What distinguishes these .MAP files from others is that line numbers for various source files appear at the end. To obtain these line numbers, you have to instruct the compiler to generate them in the first place and then instruct the linker to include them in the .MAP file. For example, in using Microsoft C, include the **/Zi** switch for programs you wish to debug at the source level. Then for the linker, use the **/MAP** and **/LI** switches to create a .MAP file with LIne numbers.

Alternatively use the **/Zi** switch for the compiler and the **/CO** switch for the linker. This generates an .EXE file for use with Microsoft's CodeView debugger. Such .EXE files can be used directly, simply by loading them as you would any other .EXE files. The CodeView compile and link options automatically tack the appropriate source line number and symbol-table information onto the back of these special .EXE files. SST detects the presence of this information and translates it for internal use.

If SST detects source code for the C-language main() routine, SST automatically switches to source-level debugging and displays the beginning of the source file for the main program. Typing the space bar then breaks at main() rather than single stepping through the C-system set up code. Hopefully that setup code is thoroughly debugged by now!

Source/Assembly Modes

Unassembled code can always be displayed using 8086/8087/80386 assembly language mnemonics. If the appropriate source code files and corresponding line numbers are available, code can also be displayed in mixed SOURCE/ASSEMBLY MODE, and in pure SOURCE MODE. Mixed mode displays the source line first followed by the corresponding assembly mnemonics.

To switch between these formats in COMMAND MODE, type

```
s+
s&
s-
```

for source, mixed, and assembly formats, respectively. To switch between formats in TRACE MODE, type +, &, or -, respectively.

Source-Level Tracing

As for assembly-level tracing, a number of hot keys are available for source-level tracing. These correspond closely to their assembly-level counterparts described in Chaps. 7 and 14, and are described in this section.

The Don't call hot key single-steps source code without going down calls. The " " hot key single-steps source code going down calls (but not ints). Extra space bars encountered while source-level single stepping are ignored. The Fast hot key sets temporary breakpoint at next source line. The Nop hot key skips current line. The Continue hot key works at the source level. EXE's/MAP's with a linenumber for _main come up in TRACE MODE with source file display. The first " " or D breaks on _main.

If you stop on nonsource code due to a console interrupt or other reason, you can don't trace to the next source line by typing l or L (for Line). If a source single step initiated by " " runs too slow, type d or D, to switch to the Don't call subroutine single step. Other keys interrupt on nonsource code, giving an assembly language display.

The Undo hot key works up to the number of assembly language instructions saved. Source code is displayed whenever the instruction pointer (cs:ip) points at a source line.

The PgUp, PgDn, up/down arrows, and Function Key 3 Search options work with source-level tracing and unassembling.

In both UNASSEMBLE and TRACE MODE's, breakPoints are toggled by **P** at cursor, and the **H**ere hot key sets a temporary breakpoint at the address at the cursor and goes. The **K**ick hot key moves the Instruction Pointer to the address at the cursor.

User Defined Symbols

To facilitate both source-level and assembly language debugging, SST includes commands to define typical data types. Specifically in COMMAND MODE, you can execute the **char**, **int**, **long**, **float**, and **real** commands. Other C types will be added in the future. The syntax is:

type [[[far]] *]] *variable name address*

For example,

```
int alpha 305
```

adds the symbol `alpha` of type **int** (signed 16-bit integer) to the segment specified by the **ds** segment register at the offset 305. You can specify any other segment. The optional * generates near **ptr** to variable of type *type*. The optional **far** generates a far pointer to the variable of type *type*.

The C-language types **char**, **int**, **long**, **float**, and **double** are supported as well as the hexadecimal types **byte**, **word** (unsigned 16-bits), and **dword** (**dd**).

The space for symbol tables is allocated dynamically with up to 64K bytes of tables. Each symbol has a 6-byte header as well as the symbol name. The header contains a 16-bit offset value, a 16-bit type, and a 16-bit length. Line numbers require the header alone unless a label or comment is included. The symbol entries are grouped into symbol segment lists. If the average symbol is 6 bytes in length, then up to about 5000 symbols can be referenced by name. The */n* RAM option on the DOS command line is not needed for symbols.

Chapter 12
SPECIAL 80386 SUPPORT

SST supports 80386 mnemonics for the assembling, unassembling, and tracing. In addition a number of special features have been added to take advantage of the 80386's special hardware. This chapter describes these features.

Hardware Breakpoints

The 80386 has four hardware breakpoints capable of breaking when a location (**byte**, **word**, or **dword**) is read, written, or executed. These breakpoints occur while the 80386 runs full speed. In particular, you can break execution as soon as a variable is written to or from, and you can set breakpoints in ROM. For a full discussion of this powerful feature of the 80386, please consult the chapter on debugging in the Intel *80386 Programmer's Reference Manual.*

SST uses sticky breakpoints 30 to 33 are for the 80386 debug registers. The syntax is a slight extension of the standard sticky breakpoint syntax (see the breakpoint command in Chap. 7). It is

▸ bs*n address* [*m*] [/*st*]

which sets 80386 hardware breakpoint *n* = 30 to 33 at the address *address* and skips *m* passes by this address. The special 80386 optional switch field /n*st* can be used to specify the Scope and Type of hardware breakpoint desired. The scope letter *s* can be omitted or be "g" for Global, or "l" for Local. Global is the default. If you specify Local, the breakpoints are automatically disabled as soon as a task switch occurs.

The breakpoint type letter *t* can be "r" for break on data reads or writes, but not on instruction fetches; "w" for break on data writes only (default); or "x" for break on instruction execution only.

These 80386 hardware sticky breakpoints can be enabled, disabled, cleared, and listed just as regular sticky breakpoints can.

Special displays

The **dr** displays the 80386 special debug registers (**dr0, dr1, dr2, dr3, dr6, dr7**), the control registers (**cr0, cr2, cr3**), the test registers (**tr6** and **tr7**), and the extended flags register. Type **dr** in COMMAND MODE.

Memory can be displayed with a linear address in real and protected 386 modes. Type the command

```
▸ dx offset
```

where on the 80386 the offset can be 32-bit.

The Infamous 80286 LoadAll Command

The 80286 microprocessor has an undocumented instruction called the *loadall* instruction with the op code 0F05. This instruction is very useful because it loads up the 80286's many registers rapidly in comparison to loading them up with documented instructions.

The 80386 has no such instruction and therefore generates an undefined op code exception (**int 6**) when it encounters the 0F05. The BIOS in the Compaq DeskPro 80386 emulates the loadall instruction. SST's **int-6** handler passes the **loadall** instruction on to previous **int-6** handler. The **loadall** instruction is also unassembled, but it is not supported by the assembler.

Protected Mode Debugging

On 80386 machines with 2 or more megabytes of memory, you can run SST in a special protected mode environment with special powers hitherto limited to the most advanced hardware debuggers. To do this, run the SST DOS Extender program V86MON.EXE, and switch SST into protected mode with the COMMAND MODE command "vm". It seems as if SST is running as usual, except for the display of a reverse-video lower-case "v" in the lower right corner of the register window. This character signifies the special "V86" debug mode.

In this mode, SST runs in protected paged mode outside of the usual DOS space, which is the first megabyte of RAM. You can run the `dr` command to display the 80386 special registers, and you can display memory in the linear address mode (`dx`*n*). In addition, you can protect I/O ports and pages of memory with the `pp` *n* and `page` *n* [*m*] commands. These commands are defined as follows:

To prevent any I/O port from being read or written, use the Port Protect command

`pp` *n*

where *n* is the desired port to protect. If any port so protected is subjected to an I/O reference, SST interrupts program execution displaying the message "PORT PROTECTION". Going into TRACE MODE, you see the instruction that tried to reference a port.

The 80386 has a paged memory facility used to implement multitasking operating systems. In addition, SST uses it to implement a powerful memory protection scheme. SST's `page` command gives substantial access to the 80386 paging mechanism (see Chap. 5 of the Intel

Programmer's Reference Manual). The syntax of the page command is as follows:

```
page n [m]
```

where *n* is the absolute (flat) address of the start of a page + a low byte value specifying the desired 80386 page-table protection/attribute characteristics (see section 5.2.4 of the Intel 80386 *Programmer's Reference Manual*). The optional *m* value specifies the number of page table entries to receive the protection/attribute byte. This byte with the bit meanings

bit 0 = 1	means the page is present; = 0 means it is not present (80386 page Present bit)
bit 1 = 1	means the page is writeable; = 0 means it is read only (for users) (80386 page Read/Write bit)
bit 2 = 1	means the page is can only be accessed by the supervisor (SST); = 0 means both the user and the supervisor can access the page (80386 page User/Supervisor bit)
bit 3 = 1	illegal page table entry causing SST to display a full screen of the page table with the cursor pointing at the entry specified.
bit 4	Intel reserved, must be 0
bit 5 = 1	means page has be read or written since last page table entry setting (80386 Access bit)
bit 6 = 1	means page has be written since last page table entry setting (80386 page Dirty bit)
bit 7	Intel reserved, must be 0

The SST protected-mode V86 debugging facility is compatible with RAM disk and disk caches that use extended RAM, but not yet with Expanded Memory Managers that use the 80386 protected mode, or with other protected mode programs. SST in V86 debug mode is new and probably still has problems, but in principle it's incredibly powerful and exciting.

Chapter 13
MICROSOFT WINDOWS DEBUGGING

Microsoft Windows provides an extensive multitasking graphics interface for programs running under MSDOS. Because it can swap user applications in and out of RAM, the location of the application segments and of application breakpoints can change at any time. To aid in the debugging in such an environment, Microsoft Windows checks to see if a Microsoft Windows cognizant debugger resides in RAM. Specifically it checks to see if the word SEGDEBUG is at offset 100h of the segment given by interrupt vector 3, the breakpoint interrupt. If so, Microsoft Windows far calls offset 0FBh of that segment with a variety of information about where and if application segments are loaded in memory. Furthermore, the Sys Req key causes Microsoft Windows to issue an **int** 2, which the debugger can trap.

SST has an appropriate Microsoft Windows interface that let's you set virtual breakpoints in user applications even before Microsoft Windows itself is loaded, let alone the applications themselves. By using the **/LI** option for the linker, you can have line number information in the application .MAP file, which allows multitasked source level debugging. This chapter describes how to set up SST to debug Microsoft Windows applications.

Necessary Equipment

To run Microsoft Windows with SST debug support, you need a monochrome card for SST and a color card for Microsoft Windows. This is also a good combination for debugging programs in general. Other combinations may work, but haven't been checked out at Scroll Systems.

Debugging Microsoft Windows Applications

First load SST on the monochrome monitor and load in the desired symbol tables from LINK4.EXE .MAP files using the `ll` command. Then type `qs1` and `qs` to change DOS to the EGA card and return SST to the monochrome monitor. Set any desired sticky breakpoints. Then type `q/ry` to quit and leave SST resident. These commands can be included in a script file. Back at the DOS prompt, type `win` to bring up Microsoft Windows.

As Microsoft Windows loads applications, assigns and reassigns segments, the monochrome screen reports what's going on. The COMMAND MODE command

```
▸msw 0C0
```

turns on a debug screen update mode that overwrites old segment values with new ones. The screen so written are much easier to read than the standard ones.

If a sticky breakpoint is encountered, control is returned to SST. Typing `g` or `G ↵` or **bye** 12 returns to Microsoft Windows.

A sample Microsoft Windows script file called W is listed below and appears on the diskette to read in .MAP files for the SHAPES.EXE and CLOCK.EXE programs and leave SST resident. Run this script file with the `<w` command in SST's COMMAND MODE. Then type `win` at the EGA DOS prompt.

The Sys Req key interrupts Microsoft Windows and returns control to SST. The go or **bye** commands return control to Microsoft Windows.

Handling of Symbols

The LINK4.EXE .MAP files are loaded by the SST `ll` command. Since this occurs before Microsoft Windows itself if loaded, the segments are assigned handles (pseudosegments) starting at 0FF81h. These handles are changed to real segment paragraphs when Microsoft Windows loads the corresponding application and assigns the application segments. Subsequent instances, i.e., two or more instances of the same application executing simultaneously, automatically link to the symbol tables for the first instance.

Sticky breakpoints can be set using the symbols from the LINK4.EXE .MAP files and are assigned the appropriate 0FF80h handles. When the corresponding applications are active, the breakpoints are assigned to the real segment paragraphs. When the applications are disabled, the breakpoint segments revert to their handle values, ready to be reassigned if the applications are reloaded.

The application name can be used to identify which symbol to use. For example, HELLO!WINMAIN refers to HELLO.EXE's WinMain rather than to some other application's WinMain whose symbol table is also loaded.

When Microsoft Windows symbol tables are loaded, SST becomes case sensitive. All DOS-like commands like **dir** are recognized only in lower case (DIR and BYE do not work). Assemble mnemonics and registers must be typed in lower case.

SST also accepts Microsoft .SYM files. You can load WIN.COM under SST and run, but for some reason, SST can't then return to DOS correctly.

Sample Microsoft Windows Script File

Sarting on the monochrome monitor, the following SST script file loads in the .MAP files for CLOCK.EXE and SHAPES.EXE, initializes the EGA monitor for Microsoft Windows, returns the SST to the monochrome monitor, and terminates and stays resident. Execute this file from the SST COMMAND MODE by typing <w, where W is the filename of the script file.

```
nclock.map
11
nshapes.map
11
qs3
qs1
qs
msw c0
q/ry
```

DEBUG Tutorial

THUMBNAIL TUTORIAL ON HOW YOU CAN USE DEBUG
by
Gratz Roberts

This is a simple demonstration to show how one might make use of the DEBUG commands. It is not intended to be all inclusive, but something to stimulate your interest and help you better understand that you really can't hurt anything by using DEBUG if you follow the rules -- ALWAYS WORK ON A COPY and CHECK YOUR WORK BEFORE YOU OVERWRITE THE ORIGINAL PROGRAM.

A short four-line text data file (DEBUGTNG.FIL) is used in demonstrating DEBUG to provide a controlled environment. The contents of the text file is as follows:

```
Now is the time for him to come to her aid because the kids are unruly.
Now is the time for her to come to his aid because the kids are unruly!
Now is the time for all good  men to come to the aid of their country.
Now is the time for all good women to come to the aid of their country.
```

For sake of simplicity assume DEBUG.COM and DEBUGTNG.FIL are both on the default drive.

Note that debug command parameters can be separated by either a " " (space) or a "," (comma).

ALWAYS WORK WITH A COPY of the file you want to use DEGUG on. If the file you are copying from contains the .EXE extension, rename it in the copying process so that it no longer ends with .EXE. You could, for example, leave the extension completely off. Make it easy on yourself. Just don't try to debug a file that has the .EXE extension. (Two identical files, one with a .EXE and one without the .EXE will not load all the instructions in the same places in memory. Furthermore, the system will not let you save any changes you might make if the file has the .EXE extension.)

The DEBUG display layout is as follows:

```
     *10                Memory Content at this
Mem  HEX      ------------------- unit Byte -----------------
Seg :Addr     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  ASCII Translation

xxxx:0200   43 6F 70 79 72 69 67 68-74 20 31 39 38 35 20 62  Copyright 1985 b
```

The above sample shows a typical line from a debug display. If no addresses were issued with the Dump command, 8 such lines would be displayed. In the example, address 0200 contains the letter "C", 0201 contains the letter "o", etc. up through 020F which contains the letter "b". If the hex value to be translated is a non-printable character, it will be translated as a ".". Thus the period translation is ambiguous.

Issue the command DEBUG without a file and you will debug current memory. If the DEBUG command is followed by a valid filename, that file will be loaded into memory. In either case, the starting address is xxxx:0100, where xxxx represents memory segments and generally you need not worry about them unless you want to patch a program that is larger than 64K. The value of xxxx may differ from machine to machine, depending on its configuration and available memory. Note that the 4-digit address following the segment will always end in zero as it is incremented in multiples of 10 hex.

Assume that we type DEBUG DEBUGTNG.FIL & press RETURN. DEBUG will display either the ">" or the "-" as a prompt in column 1 of the display. The > will be displayed for Versions 1.x, whereas the - will be displayed for Version 2.x.

REGISTER COMMAND:
    Use the Register command to display the contents of the various registers, e.g.:

```
-r                  (Register)
AX=0000  BX=0000  CX=0125  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=05D9  ES=05D9  SS=05D9  CS=05D9  IP=0100   NV UP DI PL NZ NA PO NC
05D9:0100 4E            DEC     SI
```

At this stage of the game, you need only be concerned with the CX Register, because it contains the size of the file that was loaded. Since DEBUG loads the beginning of the file at address 0100, this offset must be used in computing the end of the file. Fortunately, DEBUG does this computation for you. You only have to ask!

HEX COMMAND:
    Use the Hex command to generate the sum and difference of two hex numbers.

```
-h 100 125          (Hex address address)
0225  FFDB          (Yields the sum and difference of the two numbers, pick
                    the one you are interested in, in this case, the sum.)

-h 125 100          (Note that is may be easier to understand when the
0225  0025           larger number is listed first.)
```

We now know that the end of the file should be at address 225 because we just added the number of bytes in register CX to the starting address to get the final address. To demonstrate, we'll now dump from 100 through 23F hex (end of the next line of memory).

DUMP COMMAND:
    Use the Dump command to display (dump) the contents of memory.

```
-d 100 23f          (Dump 100 through 23F)
05D9:0100  4E 6F 77 20 69 73 20 74-68 65 20 74 69 6D 65 20   Now is the time
05D9:0110  66 6F 72 20 68 69 6D 20-74 6F 20 63 6F 6D 65 20   for him to come
05D9:0120  74 6F 20 68 65 72 20 61-69 64 20 62 65 63 61 75   to her aid becau
05D9:0130  73 65 20 74 68 65 20 6B-69 64 73 20 61 72 65 20   se the kids are
```

```
05D9:0140  75 6E 72 75 6C 79 2E 0D-0A 4E 6F 77 20 69 73 20   unruly...Now is
05D9:0150  74 68 65 20 74 69 6D 65-20 66 6F 72 20 68 65 72   the time for her
05D9:0160  20 74 6F 20 63 6F 6D 65-20 74 6F 20 68 69 73 20    to come to his
05D9:0170  61 69 64 20 62 65 63 61-75 73 65 20 74 68 65 20   aid because the
05D9:0180  6B 69 64 73 20 61 72 65-20 75 6E 72 75 6C 79 21   kids are unruly!
05D9:0190  0D 0A 4E 6F 77 20 69 73-20 74 68 65 20 74 69 6D   ..Now is the tim
05D9:01A0  65 20 66 6F 72 20 61 6C-6C 20 67 6F 6F 64 20 20   e for all good
05D9:01B0  20 6D 65 6E 20 74 6F 20-63 6F 6D 65 20 74 6F 20    men to come to
05D9:01C0  74 68 65 20 61 69 64 20-6F 66 20 74 68 65 69 72   the aid of their
05D9:01D0  20 63 6F 75 6E 74 72 79-2E 0D 0A 4E 6F 77 20 69    country...Now i
05D9:01E0  73 20 74 68 65 20 74 69-6D 65 20 66 6F 72 20 61   s the time for a
05D9:01F0  6C 6C 20 67 6F 6F 64 20-77 6F 6D 65 6E 20 74 6F   ll good women to
05D9:0200  20 63 6F 6D 65 20 74 6F-20 74 68 65 20 61 69 64    come to the aid
05D9:0210  20 6F 66 20 74 68 65 69-72 20 63 6F 75 6E 74 72    of their countr
05D9:0220  79 2E 0D 0A 1A 21 64 40-64 23 64 24 64 25 64 A5   y....!d@d#d$d%d%
05D9:0230  64 26 64 2A 64 28 64 29-64 5F 64 2B 64 00 20 00   d&d*d(d)d_d+d. .
```

Examine the dump at 05D9:0220. You will find the first character of that line of the dump is the letter "y" followed by a period. Then there is a carriage return (0D) followed by a line feed (0A) followed by an EOF (end-of-file flag 1A). The rest of the line contains data that was resident in memory when DEBUG loaded the current file. But before going on to that, by now you have noticed that the last byte of the file is not in location 225 as we previously computed! What did we do wrong? Note that when you give the starting address and the length, the computed sum will be one too large. Conversely, the computed difference will be one less than the correct value.

When you load a file, you'd like to be sure you can tell what is being loaded from what might have been there before. A simple way to do that in this case would be to issue the Fill command (F 225 23F 00); however, why not clear all the area we are going to load into and pick up another command in the process.

FILL COMMAND:
    Use the Fill command to preload or to flood the contents of memory with a given value.

```
-f 100 23f 00        (Fill 100 through 23F with 00. You could also have filled
                     it with 77 or AA or FF, but 00 usually means empty.)

-d 100 23f           (Dump 100 through 23F just to see what it looks like)

05D9:0100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
05D9:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
05D9:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
           -----     ETC    ----
05D9:0210  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
05D9:0220  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
05D9:0230  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

Now load the data back into memory. You could accomplish this be exiting DEBUG and then entering it again as we did initially, or we could simply tell DEBUG that we want to load the file. But before we can load a file at this point, we must tell DEBUG the name of the file to be loaded.

NAME COMMAND:

Use the Name command to tell DEBUG the name of the file to be subsequently read (when Load command is issued) or written (when the Write command is issued).

```
-n debugtng.fil     (Name DEBUGTNG.FIL is the name of the file to be loaded)
```

LOAD COMMAND:

Use the Load command to load a file into memory so that DEBUG can process it.

```
-l 100              (Load the file beginning at address 100. Actually you
                     could have left the address off and it would still have
                     loaded at address 100. The address is being used here
                     to show that it could have been loaded at some other
                     location, at the end of another file, for example.)

-d 100 23f          (Dump 100 through 23F again)
05D9:0100  4E 6F 77 20 69 73 20 74-68 65 20 74 69 6D 65 20   Now is the time
05D9:0110  66 6F 72 20 68 69 6D 20-74 6F 20 63 6F 6D 65 20   for him to come
05D9:0120  74 6F 20 68 65 72 20 61-69 64 20 62 65 63 61 75   to her aid becau
05D9:0130  73 65 20 74 68 65 20 6B-69 64 73 20 61 72 65 20   se the kids are
05D9:0140  75 6E 72 75 6C 79 2E 0D-0A 4E 6F 77 20 69 73 20   unruly...Now is
05D9:0150  74 68 65 20 74 69 6D 65-20 66 6F 72 20 68 65 72   the time for her
05D9:0160  20 74 6F 20 63 6F 6D 65-20 74 6F 20 68 69 73 20    to come to his
05D9:0170  61 69 64 20 62 65 63 61-75 73 65 20 74 68 65 20   aid because the
05D9:0180  6B 69 64 73 20 61 72 65-20 75 6E 72 75 6C 79 21   kids are unruly!
05D9:0190  0D 0A 4E 6F 77 20 69 73-20 74 68 65 20 74 69 6D   ..Now is the tim
05D9:01A0  65 20 66 6F 72 20 61 6C-6C 20 67 6F 6F 64 20 20   e for all good
05D9:01B0  20 6D 65 6E 20 74 6F 20-63 6F 6D 65 20 74 6F 20    men to come to
05D9:01C0  74 68 65 20 61 69 64 20-6F 66 20 74 68 65 69 72   the aid of their
05D9:01D0  20 63 6F 75 6E 74 72 79-2E 0D 0A 4E 6F 77 20 69    country...Now i
05D9:01E0  73 20 74 68 65 20 74 69-6D 65 20 66 6F 72 20 61   s the time for a
05D9:01F0  6C 6C 20 67 6F 6F 64 20-77 6F 6D 65 6E 20 74 6F   ll good women to
05D9:0200  20 63 6F 6D 65 20 74 6F-20 74 68 65 20 61 69 64    come to the aid
05D9:0210  20 6F 66 20 74 68 65 69-72 20 63 6F 75 6E 74 72    of their countr
05D9:0220  79 2E 0D 0A 1A 00 00 00-00 00 00 00 00 00 00 00   y...............
05D9:0230  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

Notice that the memory locations between the EOF flag and 23F have been loaded with 00 just as we told it.

```
-d 100 L125         (Dump starting at 100 a total of 123 hex bytes)
05D9:0100  4E 6F 77 20 69 73 20 74-68 65 20 74 69 6D 65 20   Now is the time
05D9:0110  66 6F 72 20 68 69 6D 20-74 6F 20 63 6F 6D 65 20   for him to come
05D9:0120  74 6F 20 68 65 72 20 61-69 64 20 62 65 63 61 75   to her aid becau
05D9:0130  73 65 20 74 68 65 20 6B-69 64 73 20 61 72 65 20   se the kids are
05D9:0140  75 6E 72 75 6C 79 2E 0D-0A 4E 6F 77 20 69 73 20   unruly...Now is
05D9:0150  74 68 65 20 74 69 6D 65-20 66 6F 72 20 68 65 72   the time for her
```

```
05D9:0160  20 74 6F 20 63 6F 6D 65-20 74 6F 20 68 69 73 20    to come to his
05D9:0170  61 69 64 20 62 65 63 61-75 73 65 20 74 68 65 20   aid because the
05D9:0180  6B 69 64 73 20 61 72 65-20 75 6E 72 75 6C 79 21   kids are unruly!
05D9:0190  0D 0A 4E 6F 77 20 69 73-20 74 68 65 20 74 69 6D   ..Now is the tim
05D9:01A0  65 20 66 6F 72 20 61 6C-6C 20 67 6F 6F 64 20 20   e for all good
05D9:01B0  20 6D 65 6E 20 74 6F 20-63 6F 6D 65 20 74 6F 20    men to come to
05D9:01C0  74 68 65 20 61 69 64 20-6F 66 20 74 68 65 69 72   the aid of their
05D9:01D0  20 63 6F 75 6E 74 72 79-2E 0D 0A 4E 6F 77 20 69    country...Now i
05D9:01E0  73 20 74 68 65 20 74 69-6D 65 20 66 6F 72 20 61   s the time for a
05D9:01F0  6C 6C 20 67 6F 6F 64 20-77 6F 6D 65 6E 20 74 6F   ll good women to
05D9:0200  20 63 6F 6D 65 20 74 6F-20 74 68 65 20 61 69 64    come to the aid
05D9:0210  20 6F 66 20 74 68 65 69-72 20 63 6F 75 6E 74 72    of their countr
05D9:0220  79 2E 0D 0A 1A                                    y....
```

Note that the dump ended with the EOF flag.

```
-d 10a 148          (Dump 10A through 148)
05D9:0100                                20 74 69 6D 65 20         time
05D9:0110  66 6F 72 20 68 69 6D 20-74 6F 20 63 6F 6D 65 20   for him to come
05D9:0120  74 6F 20 68 65 72 20 61-69 64 20 62 65 63 61 75   to her aid becau
05D9:0130  73 65 20 74 68 65 20 6B-69 64 73 20 61 72 65 20   se the kids are
05D9:0140  75 6E 72 75 6C 79 2E 0D-0A                        unruly...
```

Notice what happens above when the dump doesn't either start or terminate with full lines. Note that the hex address always increments in steps of 10.

```
-d 10a L3f          (Dump beginning at 10A a total of 3F bytes -- the L means
                    length)
05D9:0100                                20 74 69 6D 65 20         time
05D9:0110  66 6F 72 20 68 69 6D 20-74 6F 20 63 6F 6D 65 20   for him to come
05D9:0120  74 6F 20 68 65 72 20 61-69 64 20 62 65 63 61 75   to her aid becau
05D9:0130  73 65 20 74 68 65 20 6B-69 64 73 20 61 72 65 20   se the kids are
05D9:0140  75 6E 72 75 6C 79 2E 0D-0A                        unruly...
```

Suppose you need to compare two streams of memory to find out what, if anything, differs between the two streams. To demonstrate, compare the first sentence of the file with the second sentence of the file.

COMPARE COMMAND:
Use the Compare command to locate variations between two streams (ranges) of memory.

```
-c 100 148 149      (Compare 100 through 148 with the same number of bytes that
                    begin at address 149)

First Stream    Second Stream
05D9:0115  69  65  05D9:015E     (i is not equal to e)
05D9:0116  6D  72  05D9:015F     (m is not equal to r)
05D9:0124  65  69  05D9:016D     (e is not equal to i)
05D9:0125  72  73  05D9:016E     (r is not equal to s)
05D9:0146  2E  21  05D9:018F     (. is not equal to !)
```

So you can see that each character that did not match was printed out for your examination. Thus the Compare command essentially says that both streams are identical, except for the addresses and data listed.

```
-h 148 10a          (Hex 148 and 10A, Get sum and difference, remember?)
0252  003E
```

MOVE COMMAND:

Use the Move command to copy the contents of one stream in memory to another location. Note that the Move does not destroy the source stream.

```
-m 17c L16 225      (Move starting at location 17C, 16 hex bytes to a new
                     location starting at address 225. Note that the move
                     command is a misnomer inasmuch as it copies into a new
                     location rather than moving the original contents.)

-d 170 23F          (Dump 170 through 23F)
05D9:0170  61 69 64 20 62 65 63 61-75 73 65 20 74 68 65 20   aid because the
05D9:0180  6B 69 64 73 20 61 72 65-20 75 6E 72 75 6C 79 21   kids are unruly!
05D9:0190  0D 0A 4E 6F 77 20 69 73-20 74 68 65 20 74 69 6D   ..Now is the tim
05D9:01A0  65 20 66 6F 72 20 61 6C-6C 20 67 6F 6F 64 20 20   e for all good
05D9:01B0  20 6D 65 6E 20 74 6F 20-63 6F 6D 65 20 74 6F 20    men to come to
05D9:01C0  74 68 65 20 61 69 64 20-6F 66 20 74 68 65 69 72   the aid of their
05D9:01D0  20 63 6F 75 6E 74 72 79-2E 0D 0A 4E 6F 77 20 69    country...Now i
05D9:01E0  73 20 74 68 65 20 74 69-6D 65 20 66 6F 72 20 61   s the time for a
05D9:01F0  6C 6C 20 67 6F 6F 64 20-77 6F 6D 65 6E 20 74 6F   ll good women to
05D9:0200  20 63 6F 6D 65 20 74 6F-20 74 68 65 20 61 69 64    come to the aid
05D9:0210  20 6F 66 20 74 68 65 69-72 20 63 6F 75 6E 74 72    of their countr
05D9:0220  79 2E 0D 0A 1A 74 68 65-20 6B 69 64 73 20 61 72   y....the kids ar
05D9:0230  65 20 75 6E 72 75 6C 79-21 0D 0A 2B 64 00 20 00   e unruly!..+d. .
```

Note that the data we moved (copied) is not at the end of the file but has not overwritten the EOF flag. If we were to tell DEBUG to save our data now, it would only save the original data. The data we just appended to the end of the file would not be written out. We take care of that by erasing the EOF flag and telling DEBUG how long the file is via the RCX (Register CX command).

But first, why not enter new data for the EOF flag and the "the" that follows? We'll enter the word "Your" in place of those 4 bytes.

ENTER COMMAND:

Use the Enter command to replace one or more bytes in memory with new values.

```
-e 224             (Enter bytes beginning at 224)

05D9:0224  1A.      (The machine responds with address then data)
                   (type 59 after the period, press the SPACE bar and the
                    contents of the next cell will be displayed followed by
```

```
                      the period, etc.  To terminate the entry press RETURN)

05D9:0224 1A.59 74.6F 68.75 65.72 (This is how the line would look before
                      the RETURN key is pressed)
```

Note that it would have been much easier to have used the Fill instruction (e.g. F 224 L4 "Your") instead of the Enter, because the Fill does not require dialog in HEX.

```
-d170 23F             (Dump the data again & look at result)
05D9:0170  61 69 64 20 62 65 63 61-75 73 65 20 74 68 65 20   aid because the
05D9:0180  6B 69 64 73 20 61 72 65-20 75 6E 72 75 6C 79 21   kids are unruly!
05D9:0190  0D 0A 4E 6F 77 20 69 73-20 74 68 65 20 74 69 6D   ..Now is the tim
05D9:01A0  65 20 66 6F 72 20 61 6C-6C 20 67 6F 6F 64 20 20   e for all good
05D9:01B0  20 6D 65 6E 20 74 6F 20-63 6F 6D 65 20 74 6F 20    men to come to
05D9:01C0  74 68 65 20 61 69 64 20-6F 66 20 74 68 65 69 72   the aid of their
05D9:01D0  20 63 6F 75 6E 74 72 79-2E 0D 0A 4E 6F 77 20 69    country...Now i
05D9:01E0  73 20 74 68 65 20 74 69-6D 65 20 66 6F 72 20 61   s the time for a
05D9:01F0  6C 6C 20 67 6F 6F 64 20-77 6F 6D 65 6E 20 74 6F   ll good women to
05D9:0200  20 63 6F 6D 65 20 74 6F-20 74 68 65 20 61 69 64    come to the aid
05D9:0210  20 6F 66 20 74 68 65 69-72 20 63 6F 75 6E 74 72    of their countr
05D9:0220  79 2E 0D 0A 59 6F 75 72-20 6B 69 64 73 20 61 72   y...Your kids ar
05D9:0230  65 20 75 6E 72 75 6C 79-21 0D 0A 1A 20 21 22 00   e unruly!... !".
```

Now before we save our work, we must load the CX register with the correct number of bytes to be written. How many?

```
-h 23B 100             (Hex command -- find the difference)
033B 013B             (Difference = 013b; therefore we must write a total of
                      013B + 1 or 013C bytes out to the file.)
```

REGISTER COMMAND (SPECIFIC):

Use the Register command followed by the register name to display the contents of and/or modify the designated register.

```
-rcx                   (Display contents of the CX register)
CX 0125               (CX register has 0125 hex)
:                     (Enter new value after the colon, i.e. 013C)

-rcx                   (Display CX again to be sure we did it right.  Press
CX 013C                the RETURN key at the colon and contents will remain
:                     unchanged)
```

WRITE COMMAND:

Use the Write command to write the modified program back out to a file.

```
-w                    (Write the data out to the file)
Writing 013C bytes (DEBUG tells you how many bytes it is writing)
```

Now for a more practical example.

Use the Search command to locate the "ESCape" string in VICMINI.EXE. If the file did not have the .EXE extension, we could use the Name and Load commands and then proceed; however, in this case we must exit DEBUG and change the name of the VICMINI.EXE file before we can modify it properly.

QUIT COMMAND:

Use the Quit command to exit the DEBUG program. Note that changes made but not written out to a file may be lost when you Quit.

```
-q                 (Quit DEBUG & return to DOS)
copy b:vicmini.exe a.b
debug a.b
```

Display the registers (Remember we need to know how big the file is).

```
-r
AX=0000  BX=0000  CX=B100  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=05D9  ES=05D9  SS=05D9  CS=05D9  IP=0100   NV UP DI PL NZ NA PO NC
05D9:0100 4D            DEC     BP
```

The file is B100 bytes long. Thus, we can now search from beginning to end as follows:

SEARCH COMMAND:

Use the Search Command to locate text or other strings of data in memory.

```
-s 100 LB100 "ESC" (Search beginning at 100 a total length of B100 bytes
                   for the string "ESC")

05D9:9535           (Found string "ESC" at location 9535)
05D9:9851           (Found string "ESC" at location 9851)
05D9:AF3F           (Found string "ESC" at location AF3F)
```

(I've started at the last one, working back toward the beginning, because I already know the the first one is the correct one.)

```
-d AF00            (Dump AF00. Note that I picked a few bytes in from of the
                   target string to get a better appreciation of what is
                   present in this area)
05D9:AF00  16 1B 70 73 79 6E 1B 71-00 00 00 00 17 1B 70 65   ..psyn.q......pe
05D9:AF10  74 62 1B 71 00 00 00 00-18 1B 70 63 61 6E 1B 71   tb.q......pcan.q
05D9:AF20  00 00 00 00 19 1B 70 65-6D 1B 71 00 00 00 00 00   ......pem.q.....
05D9:AF30  1A 1B 70 73 75 62 1B 71-00 00 00 00 1B 1B 70 45   ..psub.q......pE
05D9:AF40  53 43 1B 71 00 00 00 00-1C 1B 70 66 73 1B 71 00   SC.q......pfs.q.
05D9:AF50  00 00 00 00 1D 1B 70 63-73 1B 71 00 00 00 00 00   ......pcs.q.....
05D9:AF60  1E 1B 70 72 73 1B 71 00-00 00 00 00 1F 1B 70 75   ..prs.q.......pu
05D9:AF70  73 1B 71 00 00 00 00 00-F1 1B 70 20 31 20 1B 71   s.q.....q.p 1 .q
```

The ESC above, was not the one we were trying to find. Try the next one.

```
-d 9800            (Dump 9800)
05D9:9800  49 4C 45 53 20 20 20 20-20 20 20 20 20 20 20 20   ILES
05D9:9810  20 20 20 20 20 20 20 20-20 20 20 20 20 53 50 45                SPE
05D9:9820  43 49 41 4C 00 20 59 20-2E 2E 2E 20 4C 69 73 74   CIAL. Y ... List
05D9:9830  20 64 69 73 6B 20 66 69-6C 65 73 20 20 20 20 20    disk files
05D9:9840  20 20 20 20 5A 20 2E 2E-2E 20 43 68 61 6E 67 65       Z ... Change
05D9:9850  20 45 53 43 61 70 65 20-6B 65 79 00 20 57 20 2E    ESCape key. W .
05D9:9860  2E 2E 20 54 79 70 65 20-28 76 69 65 77 29 20 61   .. Type (view) a
05D9:9870  20 66 69 6C 65 20 20 20-20 20 20 21 20 2E 2E 2E    file      ! ...
```

The ESC here is not the one either as it is from the full screen menu.

```
-d 9500            (Dump 9500)
05D9:9500  2A 20 4D 69 6E 69 74 65-6C 20 63 61 6E 6E 6F 74   * Minitel cannot
05D9:9510  20 62 65 20 72 75 6E 20-2A 2A 2A 2A 07 0A 0D 00    be run ****....
05D9:9520  25 73 20 20 76 25 73 20-20 20 20 2D 20 20 20 20   %s  v%s    -
05D9:9530  54 79 70 65 20 45 53 43-61 70 65 20 66 6F 72 20   Type ESCape for
05D9:9540  4D 65 6E 75 00 00 44 69-73 6B 20 46 75 6C 6C 21   Menu..Disk Full!
05D9:9550  20 28 54 65 78 74 20 43-6F 6C 6C 65 63 74 69 6F    (Text Collectio
05D9:9560  6E 29 07 20 20 00 43 6F-6D 6D 61 6E 64 3A 20 00   n).  .Command: .
05D9:9570  20 20 20 20 20 20 20 20-20 00 4D 49 4E 49 54 45             .MINITE
```

This is the one we're looking for. Now change the "ESCape" to "FKEY 1".

```
-f 9535 L6 "FKEY 1" (Fill the six cells beginning at 9535 with the text
                     string enclosed by the quotation marks.)

-d9500             (Dump 9500 -- check our work!)
05D9:9500  2A 20 4D 69 6E 69 74 65-6C 20 63 61 6E 6E 6F 74   * Minitel cannot
05D9:9510  20 62 65 20 72 75 6E 20-2A 2A 2A 2A 07 0A 0D 00    be run ****....
05D9:9520  25 73 20 20 76 25 73 20-20 20 20 2D 20 20 20 20   %s  v%s    -
05D9:9530  54 79 70 65 20 46 4B 45-59 20 31 20 66 6F 72 20   Type FKEY 1 for
05D9:9540  4D 65 6E 75 00 00 44 69-73 6B 20 46 75 6C 6C 21   Menu..Disk Full!
05D9:9550  20 28 54 65 78 74 20 43-6F 6C 6C 65 63 74 69 6F    (Text Collectio
05D9:9560  6E 29 07 20 20 00 43 6F-6D 6D 61 6E 64 3A 20 00   n).  .Command: .
05D9:9570  20 20 20 20 20 20 20 20-20 00 4D 49 4E 49 54 45             .MINITE

-w                 (Write the corrections back out to the file)

Writing 8100 bytes (Wrote the same number of bytes it started with)

-q                 (Quit -- return to DOS)
```

Now correct the name, rename a.b vicmini.exe, and then check the program out to see if it works OK. If so, replace the original version with the patched copy.

This covers most of DEBUG except for Input, Output, Trace, and Unassemble, which are probably a little too complex for beginning DEBUG users. Hope you have found it to be a little useful.

SUMMARY OF DEBUG COMMANDS & FUNCTIONS

FUNCTION	COMMAND & PARAMETER	RESULT
Compare	C <range> <address>	Compares Memory Ranges
Dump	D [<address> [L <value>]] D <range>	Displays Specified Range of Memory
Enter	E <address> [<list>]	Change contents of Memory
Fill	F <range> <list>	Flood Memory with Value
Go	G [= <address> [<address> ..]]	Execute Memory
Hex	H <address> <address>	Sum & Difference Calc.
Input	I <value>	Get Data from Port
Load	L [<address> [<drive> <record> <record>]]	Load Memory from File
Move	M <range> <address>	Move Memory to new Loc.
Name	N <file name>	Specify File to be Read
Output	O <value> <byte>	Send Data to Port
Quit	Q (Exit DEBUG)	Exit the DEBUG Program
Register	R [<register name>]	Disp/Change Registers
Search	S <range> <list>	Find String/Value Loc.
Trace	T [<address>] [<value>]	For use with ASM Prog.
Unassemble	U [<address> [L <value>]] U [<range>]	Generates ASM like code
Write	W [<address> [<drive> <record> <record>]]	Write NNNN Bytes of Memory to File, where NNNN=Contents of CX Register.

PARAMETER	DEFINITION
Address	A two-part designation consisting of a Segment Register designation followed by a 4-digit memory address (e.g. CS:0100) or a 4-digit segment address followed by an offset value (e.g. 0ABC:0100). If the address is omitted, debug will assume the first address after the last one it processed. On entering DEBUG, the default address is 0100 hex.
Byte	A two-digit hexadecimal value (e.g. 01 or 2A or FF, etc.) from 00 hex thru FF hex.
Drive	A one-digit hexadecimal value indicating the drive to which the file will be read and/or written. (Values are 0 = Drive A, 1 = Drive B, 2 = drive C, 3 = Drive D, etc.)
List	A series of byte values or strings. If List is used, it MUST be the last parameter on the command line.
Range	The range parameter consists of two addresses, or one address followed by the letter L and a value, where the value indicates the number of bytes that the command acts upon.
Record	A one-, two-, or three-digit hexadecimal value that indicates the logical record number of the disk and number of disk sectors to be written or loaded.
String	Any number of characters enclosed in quotation marks, where quotation marks in this case can be either the ' or the ", but must be consistent. Within one, the other can be used as a literal string.
Value	A hexadecimal value in the range of 0000 thru FFFF inclusive.

DEBUG

Second VICTOR printing November, 1983.

ISBN 0-88182-093-8 Printed in U.S.A.

DEBUG

Overview

DEBUG is a debugging program that provides a controlled testing environment for binary and executable object files. DEBUG works on binary files in the same way a text editor works on source files. It lets you alter the contents of a file or CPU register, and then immediately reexecute a program to check the validity of the changes. DEBUG eliminates the need to reassemble a program to see if a problem has been fixed by a minor change.

You can abort all DEBUG commands at any time by pressing Ctrl-C. Ctrl-S freezes the display, so that you can read it before the output scrolls away. Pressing any other key restarts the display.

D.1 Using DEBUG

You can start DEBUG using two methods. With method 1, you type all commands in response to the DEBUG prompt. With method 2, you type all commands at the same time.

Table D-1: Methods to Start DEBUG

METHOD	COMMAND
1	DEBUG
2	DEBUG [filespec [arglist]]

D.1.1 Method 1: Prompts

To start DEBUG using method 1, type

```
DEBUG↵
```

DEBUG responds with the hyphen (-) prompt, signaling that it is ready to accept your commands. Because you have not specified a filename, you can use other commands to work on current memory, disk sectors, or disk files.

WARNING: When DEBUG starts, it sets up a program header at offset 0 in the program work area. On previous versions of DEBUG you could overwrite this header. You can still overwrite the default header if you do not specify a filespec. If you are debugging a .COM or .EXE file, however, do not tamper with the program header below address 5CH, or DEBUG terminates.

Do not restart a program after the "Program terminated normally" message is displayed. You must reload the program with the N and L commands for it to run correctly.

D.1.2 Method 2: Complete Command Line

To start DEBUG using a command line, type:

```
DEBUG [filespec [arglist]]↵
```

filespec is the file to be debugged, and arglist is the rest of the command that DEBUG uses when filespec is loaded into memory. arglist is a list of filename parameters and switches that you want passed to the program filespec. You can specify an arglist if you gave a filespec. Thus, when filespec is loaded into memory, it is loaded as if it had been started with the command **filespec arglist**.

If, for example, you type

```
DEBUG FILE.EXE↵
```

DEBUG loads FILE.EXE into memory starting at 100 hexadecimal in the lowest available segment. The BX:CX registers load with the number of bytes placed into memory.

D.2 Commands

Each DEBUG command consists of a single letter followed by one or more parameters. You can use any combination of uppercase and lowercase letters in commands and parameters. **Note:** The control characters and the special editing functions described in this manual apply here.

If you make a syntax error in a DEBUG command, DEBUG reprints the command line and indicates the error with an up-arrow (^) and the word "error." For example:

```
dcs:100 cs:110
    ^ error
```

Table D-2 summarizes DEBUG commands. They are explained in detail, with examples, later in this section.

Table D-2: DEBUG Commands

COMMAND	FUNCTION
A[address]	Assemble
C range address	Compare
D[range]	Dump
E address [list]	Enter
F range list	Fill
G[= address [address...]]	Go
H value value	Hex
I value	Input
L[address [drive record record]]	Load
M range address	Move
N filename filename	Name
O value byte	Output
Q	Quit
R[register-name]	Register
S range list	Search
T[= address][value]	Trace
U[range]	Unassemble
W[address [drive record record]]	Write

All DEBUG commands except Quit accept parameters. You can separate parameters by delimiters (spaces or commas), but you must use a delimiter between two consecutive hexadecimal values. Thus, the following commands are equivalent:

```
dcs:100 110
d cs:100 110
d,cs:100,110
```

Table D-3 defines DEBUG command parameters.

Table D-3: Command Parameters

PARAMETER	DEFINITION
drive	A one-digit hexadecimal value that indicates which drive a file is loaded from or written to. These values designate drives as follows: 0 = A:, 1 = B:, 2 = C:, 3 = D:.
byte	A two-digit hexadecimal value placed in or read from an address or register.
record	A one- to three-digit hexadecimal value that indicates the logical record number on the disk and the number of disk sectors you want to write or load. Logical records correspond to sectors; however, their numbering differs because they represent the entire disk space.
value	A hexadecimal value of up to four digits that specifies a port number or the number of times a command should repeat its functions.
address	A two-part designation consisting of either an alphabetic segment register designation or a four-digit segment address and an offset value. The segment designation or segment address can be omitted; in these cases the default segment is used. DS is the default segment for all commands except G, L, T, U, and W. For these commands, the default segment is CS. All numeric values are hexadecimal. In these addresses, for example, you must put a colon between a segment designation (whether numeric or alphabetic) and an offset: CS:0100 04BA:0100
range	range consists of two addresses, an L, and a value, where value is the number of lines the command operates on, and L80 is assumed. You cannot use the last form if another hex value follows the range, since the hex value would be interpreted as the second address of the range. For example: CS:100 110 CS:100 L 10 This example is illegal: CS:100 CS:110 ^ error The limit for range is 10000 hex. To specify a value of 10000 hex within four digits, type 0000 (or 0).

PARAMETER	DEFINITION
list	A series of byte values or strings. list must be the last parameter on the command line. For example: fcs:100 42 45 52 54 41
string	Any number of characters enclosed in quotation marks. Quotation marks can be single (') or double ("). If the delimiter quotation marks appear within a string, double the quotation marks. For example, the following strings are legal: 'This is a "string" is okay.' However, this string is illegal: 'This is a 'string' is not.' Similarly, these strings are legal: "This is a 'string' is okay." "This is a ""string"" is okay." but this string is illegal: "This is a "string" is not." Double quotation marks are not needed in the following strings: "This is a "string" is not necessary." 'This is a ""string"" is not necessary.'. The ASCII values of the characters in the string are used as a list of byte values.

Assemble (A)

A[address]

Assembles 8086/8087/8088 mnemonics directly into memory.

All numeric values are hexadecimal and you must enter them as 1–4 characters. Specify prefix mnemonics in front of the opcode to which they refer. You can also enter them on a separate line.

The segment override mnemonics are CS:, DS:, ES:, and SS:. The mnemonic for the far return is RETF. String manipulation mnemonics must explicitly state the string size. For example, use MOVSW to move word strings, and MOVSB to move byte strings.

D

The assembler automatically assembles short, near or far jumps and calls, depending on byte displacement to the destination address. You can override the defaults with the NEAR or FAR prefix. For example:

```
0100:0500  JMP   502         ; a 2-byte short jump
0100:0502  JMP   NEAR 505    ; a 3-byte near jump
0100:505   JMP   FAR  50A    ; a 5-byte far jump
```

You can abbreviate the NEAR prefix to NE, but you cannot abbreviate the FAR prefix.

DEBUG cannot tell whether some operands refer to a word memory location or to a byte memory location. In these cases, you must explicitly state the data type with the prefix WORD PTR or BYTE PTR. You can also use WO and BY. For example:

```
NEG     BYTE PTR [128]
DEC     WO [SI]
```

DEBUG also cannot tell whether an operand refers to a memory location or to an immediate operand. DEBUG uses the common convention that operands enclosed in square brackets refer to memory. For example:

```
MOV      AX,21               ; Load AX with 21H
MOV      AX,[21]             ; Load AX with the contents
                             ; of memory location 21H
```

Assemble has two popular pseudo-instructions available. The DB opcode assembles byte values directly into memory. The DW opcode assembles word values directly into memory. For example:

```
DB       1,2,3,4,"THIS IS AN EXAMPLE"
DB       'THIS IS A QUOTE: "'
DB       "THIS IS A QUOTE: '"

DW       1000,2000,3000,"BACH"
```

Assemble supports all forms of register indirect commands. For example:

```
ADD      BX,34,[BP+2].[SI-1]
POP      [BP+DI]
PUSH     [SI]
```

Assemble also supports all opcode synonyms. For example:

```
LOOPZ    100
LOOPE    100

JA       200
JNBE     200
```

For 8087 opcodes, you must explicitly specify WAIT or FWAIT. For example:

```
FWAIT FADD ST,ST(3)        ; This line will assemble an
                           ; FWAIT prefix
LD TBYTE PTR [BX]          ; This line will not
```

Compare (C)

```
C range address
```

Compares the portion of memory specified by range to a portion of the same size beginning at address.

If the two areas of memory are identical, there is no display and DEBUG returns with the MS-DOS prompt. Differences are displayed in this format:

```
address1 byte1 byte2 address2
```

These two commands have the same effect:

```
C100,200 300
C100L100 300
```

Each command compares the block of memory from 100 to 1FFH with the block of memory from 300 to 3FFH.

Dump (D)

```
D[range]
```

Displays the contents of the specified region of memory.

If you specify a range of addresses, DEBUG displays the contents. If you enter the D command without parameters, 128 bytes display at the first address (DS:100) after the address displayed by the previous Dump command.

The dump displays in two portions: a hexadecimal dump (each byte is shown in hexadecimal value) and an ASCII dump (the bytes are shown in ASCII characters). Nonprinting characters are indicated by a period (.) in the ASCII portion of the display.

The display line shows 16 bytes with a hyphen between the eighth and ninth bytes. Each displayed line begins on a 16-byte boundary.

If you type the command:

```
dcs:100 110
```

DEBUG displays:

```
04BA:0100 42 45 52 54 41 ... 4E 44 TOM SAWYER
```

If you type the following command:

```
D
```

DEBUG displays 128 bytes. Each line of the display begins with an address, incremented by 16 from the address on the previous line. Each subsequent D (without parameters) displays the bytes immediately following those last displayed.

If you type the command:

```
DCS:100 L 20
```

then the display is formatted as described above, but 20H bytes are displayed.

If you then type the command:

```
DCS:100 115
```

the display is formatted as described above, but all the bytes in the range from 100H to 115H in the CS segment display.

Enter (E)

E address[list]

Enters byte values into memory at the specified address.

If you type the optional list of values, DEBUG automatically replaces the byte values. If an error occurs, no byte values are changed.

If you type the address without the list, DEBUG displays the address and its contents, then repeats the address on the next line and waits for your input. At this point the Enter command waits for you to do one of the following:

1. Replace a byte value with another value by typing the value after the current value. If the value you type is not a legal hexadecimal value, or if you enter more than two digits, DEBUG does not echo the illegal or extra character.
2. Press the Spacebar to advance to the next byte. To change the value, enter the new value after the current value. If you space beyond an 8-byte boundary, DEBUG starts a new display line with the address displayed at the beginning.
3. Type a hyphen (-) to return to the preceding byte. If you decide to change a byte behind the current position, type the hyphen to return the current position to the previous byte. When you type the hyphen, a new line is started with the address and its byte value displayed.
4. Press the Enter key to terminate the Enter command. You can press the Enter key at any byte position.

Assume you enter the following command:

```
ECS:100
```

DEBUG displays

```
04BA:0100   EB._
```

To change this value to 41, type 41 as shown:

```
04BA:0100   EB.41_
```

To step through the subsequent bytes, press the Spacebar to see

```
04BA:0100   EB.41     10.       00.       BC._
```

D

To change BC to 42, type

```
04BA:0100   EB.41     10.       00.       BC.42_
```

Now, to change 10 to 6F, type the hyphen as many times as needed to return to byte 0101 (value 10). Then replace 10 with 6F:

```
04BA:0100   EB.41     10.       00.       BC.42-
04BA:0102   00.-_
04BA:0101   10.6F_
```

Press Enter to end the Enter command and return to the DEBUG command level.

Fill (F)

```
F range list
```

Fills the addresses in the range with values in the list.

If the range contains more bytes than the number of values in the list, the list is used repeatedly until all bytes in the range are filled. If the list contains more values than the number of bytes in the range, the extra values in the list are ignored. If any of the memory in the range is not valid (bad or nonexistent), the error occurs in all succeeding locations.

Assume you type the following command:

```
F04BA:100 L 100 42 45 52 54 41
```

DEBUG fills memory locations 04BA:100 through 04BA:1FF with the bytes specified. The five values are repeated until all 100H bytes are filled.

Go (G)

```
G[=address[ address...]]
```

Executes the program currently in memory.

If you type only the Go command, the program runs as it would outside DEBUG.

If you set =address, execution begins at the address specified. The equals sign (=) is required, so that DEBUG can distinguish the start =address from the breakpoint addresses.

When the other optional addresses are set, execution stops at the first address encountered, regardless of that address's position in the list of addresses to halt execution or program branching. When program execution reaches a breakpoint, the registers, flags, and decoded instruction display for the last instruction executed. The result is the same as if you had entered the Register command for the breakpoint address.

You can set up to ten breakpoints. Breakpoints must be set, however, only at addresses containing the first byte of an 8086-88 opcode. If you set more than ten breakpoints, DEBUG returns the BP error message.

The user stack pointer must be valid and have 6 bytes available for this command. The Go command uses an IRET instruction to cause a jump to the program under test. The user stack pointer is set, and the user flags, Code Segment register, and Instruction Pointer are pushed on the user stack. Thus, if the user stack is not valid or is too small, MS-DOS can crash. DEBUG places an interrupt code (0CCH) at the specified breakpoint address(es).

When DEBUG encounters an instruction containing the breakpoint code, all breakpoint addresses are restored to their original instructions. If execution does not halt at one of the breakpoints, the interrupt codes are not replaced with the original instructions.

Assume you type the following command:

```
GCS:7550
```

The program currently in memory executes up to the address 7550 in the CS segment. DEBUG then displays registers and flags, and the Go command terminates.

After DEBUG encounters a breakpoint, you can type the Go command again and the program executes just as if you had typed the filename at the MS-DOS command level. The only difference is that program execution begins at the instruction after the breakpoint rather than at the usual start address.

Hex (H)

```
H value value
```

Performs hexadecimal arithmetic on the two parameters specified.

DEBUG adds the two parameters and subtracts the second parameter from the first. The results of the arithmetic display on a single line: first the sum, then the difference.

If you type the command:

```
H19F 10A
```

DEBUG performs the calculations and then displays the results:

```
02A9 0095
```

Input (I)

```
I value
```

Inputs and displays one byte from the port specified by value.

This command allows a 16-bit port address.

Assume you type the following command:

```
I2F8
```

Assume also that the byte at the port is 42H. DEBUG inputs the byte and displays the value 42.

Load (L)

```
L[address [drive record record]]
```

Loads a file into memory.

Set BX:CX to the number of bytes read. The file loaded must already be named. Name the file either when you start DEBUG or with the N command. Both the DEBUG invocation and the N command format a filename properly in the normal format of a file control block at CS:5C.

If you type the L command without any parameters, DEBUG loads the file into memory beginning at address CS:100 and sets BX:CX to the number of bytes loaded. If you type the L command with an address parameter, loading begins at the memory address specified.

If you type L with all the parameters, DEBUG loads absolute disk sectors instead of a file. The records are taken from the drive specified. The drive designation is numeric—0 = A:, 1 = B:, 2 = C:. DEBUG begins loading with the first record specified, and continues until the number of sectors specified in the second record are loaded.

Assume you type the following commands:

```
A>DEBUG
-NFILE.COM
```

To load FILE.COM, type L. DEBUG loads the file and displays the DEBUG prompt. To load portions of a file or certain records from a disk, type

```
L04BA:100 2 0F 6D
```

DEBUG then begins with logical record number 15 and loads 109 (6D hex) records into memory beginning at address 04BA:0100. When the records are loaded, DEBUG returns the hyphen prompt.

If the file has an .EXE extension, it is relocated to the load address specified in the header of the .EXE file: the address parameter is always ignored for .EXE files. The header itself is stripped off the .EXE file before it is loaded into memory. Thus, the size of an .EXE file on disk differs from its size in memory.

If the named file is a .HEX file, typing the L command with no parameters tells DEBUG to load the file beginning at the address specified in the .HEX file. If the L command includes the option address, DEBUG determines the start address by adding the address specified in the L command to the address found in the .HEX file.

Move (M)

M range address

Moves the block of memory specified by range to the location beginning at the address specified.

DEBUG always performs overlapping moves. Overlapping moves are where part of the block overlaps some of the current addresses without loss of data. Addresses that could be overwritten are moved first. When you want to move from higher addresses to lower addresses, this command moves the data beginning at the block's lowest address and then works toward the highest. When you want to move from lower addresses to higher addresses, DEBUG moves the data beginning at the block's highest address and works toward the lowest.

The M command actually copies data rather than moves it. If you do not plan to write new data to the addresses in the block you are moving, the existing data remains intact. Consequently, the sequence of the move is important.

Assume that you type

```
MCS:100 110 CS:500
```

DEBUG first moves data from address CS:110 to address CS:510, then CS:10F to CS:50F, and so on until CS:100 is moved to CS:500. You can review the results of the move by typing the D command, with the same address you typed for the M command.

Name (N)

```
N filename [filename]
```

Sets filenames.

The Name command performs two functions:

1. Name assigns a filename for a later Load or Write command. Thus, if you start DEBUG without naming a file to debug, you must type the N filename command before a file can be loaded.
2. Name assigns filename parameters to the file you are debugging. In this case, Name accepts a list of parameters that are used by the file being debugged.

These two functions overlap. Consider the following set of DEBUG commands:

```
-NFILE1.EXE
-L
-G
```

These commands result in four steps:

1. (N)ame assigns the filename FILE1.EXE to the filename used in any later Load or Write commands.
2. (N)ame also assigns the filename FILE1.EXE to the first filename parameter used by any program that is later debugged.
3. (L)oad loads FILE1.EXE into memory.
4. (G)o executes FILE1.EXE with FILE1.EXE as the single filename parameter, that is, FILE1.EXE is executed as if FILE1.EXE had been typed at the command level.

A more useful chain of commands might be:

```
-NFILE1.EXE
-L
-NFILE2.DAT FILE3.DAT
-G
```

Here, Name sets FILE1.EXE as the filename for the subsequent Load command. The Load command loads FILE1.EXE into memory, and then the Name command is used again, this time to specify the parameters used by FILE1.EXE. Finally, when the Go command is executed, FILE1.EXE executes as if you had typed FILE1 FILE2.DAT FILE3.DAT at the MS-DOS command level.

If DEBUG executes a Write command, then FILE1.EXE—the file being debugged—is saved with the name FILE2.DAT. To avoid these results, always execute a Name command before either a Load or a Write command.

The Name command can affect four regions of memory:

CS:5C	FCB for file 1
CS:6C	FCB for file 2
CS:80	Count of characters
CS:81	All characters typed

DEBUG sets up a File Control Block (FCB) for the first filename parameter you gave at CS:5C. If you type a second filename parameter, you set up an FCB beginning at CS:6C. The number of characters you type, exclusive of the first character, “N”, is given at location CS:80. The actual stream of characters given by the command (again, exclusive of the letter “N”) begins at CS:81.

This stream of characters might contain switches and delimiters that would be legal in any command typed at the MS-DOS command level.

A typical use of the Name command is

```
DEBUG PROG.COM
-NPARAM1 PARAM2/C
-G
-
```

In this example, the Go command executes the file in memory as if you had typed the following command line:

```
PROG PARAM1 PARAM2/C
```

Output (O)

O value byte

Send the byte specified to the output port specified by value.

This command allows a 16-bit port address.

If you type **O2F8 4F**, DEBUG outputs the byte value 4F to output port 2F8.

Quit (Q)

```
Q
```

Terminates the DEBUG utility.

The Q command takes no parameters and exits DEBUG without saving the file you are currently working on. You are returned to the MS-DOS command level.

To end the debugging session, type

```
Q
```

DEBUG terminates and control returns to the MS-DOS command level.

Register (R)

```
R[register-name]
```

Displays the contents of one or more CPU registers.

If you do not type register-name, the R command dumps the register save area and displays the contents of all registers and flags.

If you type a register name, the 16-bit value of that register displays in hexadecimal and a colon appears as a prompt. You can then either type a value to change the register, or, if you do not want any changes, press Enter.

The only valid register names are

AX	DI	PC
BP	DX	SI
BX	ES	SP
CS	F	SS
CX	IP	SX

(IP and PC both refer to the Instruction Pointer.)

Any other entry for register-name results in a BR error message. See the end of this appendix for a list of DEBUG error messages.

If you enter F as the register-name, DEBUG displays each flag with a two-character alphabetic code. To alter any flag, type the opposite two-letter code. The flags are either set or cleared.

Table D-4 lists the flags with their codes for SET and CLEAR.

Table D-4: Register Flag Codes

FLAG NAME	SET CODE	CLEAR CODE
Overflow	OV	NV
Direction	DN (Decrement)	UP (Increment)
Interrupt	EI (Enabled)	DI (Disabled)
Sign	NG (Negative)	PL (Plus)
Zero	ZR	NZ
Auxiliary Carry	AC	NA
Parity	PE (Even)	PO (Odd)
Carry	CY	NC

Whenever you type the RF command, the flags display in a row at the beginning of a line, in the order shown in Table D-4. At the end of the list of flags, DEBUG displays a hyphen (-). You can enter new flag values as alphabetic pairs, in any order. You do not have to leave spaces between the flag entries.

To exit the R command, press the Enter key. Any flags for which you did not enter new values remain unchanged.

If you enter more than one value for a flag, DEBUG returns a DF error message. If you enter a flag code other than those shown in Table D-4, DEBUG returns a BF error message. In both cases, flags up to the error in the list are changed; those flags at and after the error are not.

At startup, the segment registers are set to the bottom of free memory, the Instruction Pointer is set to 0100H, all flags are cleared, and the remaining registers are set to zero.

If you type

R

DEBUG displays all registers, flags, and the decoded instruction for the current location. If the location is CS:11A, the display looks similar to this:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A  NV UP DI NG NZ AC PE NC
04BA:011A   CD21          INT 21
```

If you type

RF

DEBUG displays the flags:

```
NV UP DI NG NZ AC PE NC - _
```

Now type any valid flag designation, in any order, with or without spaces. For example:

NV UP DI NG NZ AC PE NC - PLEICY

DEBUG responds only with the DEBUG prompt. To see the changes, type either the R or RF command:

```
RF
```

DEBUG displays

```
NV UP EI PL NZ AC PE CY - _
```

Press Enter to leave the flags this way, or to specify different flag values.

D

Search (S)

```
S range list
```

Searches the range specified for the list of bytes specified.

The list can contain one or more bytes, each separated by a space or comma. If the list contains more than one byte, only the first address of the byte string is returned. If the list contains only one byte, all addresses of the byte in the range are displayed.

If you type

```
SCS:100 110 41
```

DEBUG responds

```
04BA:0104
04BA:010D
-type:
```

Trace (T)

T[= address][value]

Executes one instruction and displays the contents of the decoded instruction and all registers and flags.

If you type the optional =address, DEBUG traces at the address specified. The optional value tells DEBUG to execute and trace the number of steps specified by value.

The T command uses the hardware trace mode of the 8086 or 8088 microprocessor. Consequently, you can also trace instructions stored in ROM, read-only memory.

If you type

T

DEBUG returns a display of the registers, flags, and decoded instruction for that one instruction. Assume that your current position is 04BA:011A; DEBUG might return the display:

```
AX=0E00 BX=00FF CX=0007 DX=01FF SP=039D BP=0000
SI=005C DI=0000 DS=04BA ES=04BA SS=04BA CS=04BA
IP=011A  NV UP DI NG NZ AC PE NC
04BA:011A  CD21          INT 21
```

If you type

T=011A 10

DEBUG executes sixteen (10 hex) instructions beginning at 011A in the current segment, and then displays all registers and flags for each instruction as it is executed. The display scrolls until the last instruction is executed. Then the display stops, and you can see the register and flag values for the last few instructions performed. Remember that Ctrl-S suspends the display at any time, so that you can study the registers and flags for any instruction.

Unassemble (U)

U[range]

Disassembles bytes and displays the source statements that correspond to them, with addresses and byte values.

The display of disassembled code looks like a listing for an assembled file. If you type the U command without parameters, DEBUG disassembles 20 hexadecimal bytes at the first address after that displayed by the previous Unassemble command. If you type the U command with the range parameter, then DEBUG disassembles all bytes in the range. If the range is given as an address only, then 20H bytes are disassembled instead of 80H that the Dump command would default to.

If you type

U04BA:100 L10

DEBUG disassembles 16 bytes beginning at address 04BA:0100:

```
04BA:0100   206472   AND   [SI+72],AH
04BA:0103   69       DB    69
04BA:0104   7665     JBE   016B
04BA:0106   207370   AND   [BP+DI+70],DH
04BA:0109   65       DB    65
04BA:010A   63       DB    63
04BA:010B   69       DB    69
04BA:010C   66       DB    66
04BA:010D   69       DB    69
04BA:010E   63       DB    63
04BA:010F   61       DB    61
```

If you enter

U04BA:0100 0108

The display shows

```
04BA:0100    206472   AND   [SI+72],AH
04BA:0103    69       DB    69
04BA:0104    7665     JBE   016B
04BA:0106    207370   AND   [BP+DI+70],DH
```

If you change bytes in some addresses, the disassembler alters the instruction statements. You can type the U command for the changed locations, the new instructions viewed, and the disassembled code used to edit the source file.

Write (W)

W[address[drive record record]]

Writes the file being debugged to a disk file.

If you type W with no parameters, BX:CX must already be set to the number of bytes to be written; the file is written beginning from CS:100. If you type the W command with just an address, DEBUG writes the file beginning at that address.

If you use a G or T command, BX:CX must be reset before you use the Write command without parameters. Note that if DEBUG loads and modifies a file, the name, length, and starting address are all set correctly to save the modified file, as long as the length has not changed.

The file must be named either with the DEBUG invocation command or with the N command. Both the DEBUG invocation and the N command format a filename properly in the normal format of a file control block at CS:5C.

If you use the W command with parameters, DEBUG writes the file beginning from the memory address specified. The file is written to the specified drive (the drive designation is numeric here—0 = A, 1 = B, 2 = C, and so on) beginning at the logical record number specified by the first record. DEBUG continues until the number of sectors specified in the second record are written.

Note: Writing to absolute sectors is extremely dangerous because the process bypasses the file handler.

If you type

```
W
```

DEBUG writes the file to disk and then displays the DEBUG prompt.

If you type

```
W
CS:100 1 37 2B
```

DEBUG writes out the contents of memory, beginning with the address CS:100, to the disk in drive **B**:. The data written out starts at logical record number 37H and consists of 2BH records. When the write is finished, DEBUG displays

```
WCS:100 1 37 2B
-_
```

D.3 DEBUG Error Messages

You might see any of the following error messages during a DEBUG session. Each error terminates DEBUG command under which it occurs, but it does not terminate DEBUG itself.

ERROR CODE	DEFINITION
BF	Bad flag: You attempted to alter a flag, but the characters typed were not one of the acceptable pairs of flag values. See the Register command for the list of acceptable flag entries.
BP	Too many breakpoints: You specified more than ten breakpoints as parameters to the Go command. Retype the Go command with ten or fewer breakpoints.
BR	Bad register: You typed the R command with an invalid register name. See the Register command for the list of valid register names.
DF	Double flag: You typed two values for one flag. You can specify a flag value only once per RF command.