hyperSOURCE-386/386SX

User Manual

September 1991

MICROTEK INTERNATIONAL, INC.
Development Systems Division
3300 N.W. 211th Terrace
Hillsboro, OR 97124-7136
Fax (503) 629-8460
(503) 645-7333

Part Number 11831-006

Trademarks

MICE-V and **hyperSOURCE-386** are trademarks of Microtek International, Inc.

IBM PC is a registered trademark of International Business Machines Corporation

XLINK is a registered trademark of Systems & Software, Inc.

CodeView is a registered trademark of Microsoft Corp.

80386 is a trademark of Intel Corporation

Customer Support

Before calling our Customer Support Department with problems, questions, or suggestions on hyperSOURCE-386, please read the Product Performance Report form found in the file "pprform.ini" in the hyperSOURCE-386 installation subdirectory. Please be prepared to provide the information requested on this form when you call. You can contact the Customer Support Department Monday through Friday from 8 a.m. to 5 p.m. Pacific Time at:

Phone: (503) 645-7333

FAX: (503) 629-8460

Preface

hyperSOURCE-386 Overview

HyperSOURCE-386 is a windowed interface providing both assembly-level and source-level debugging capability through the MICE-V 80386 emulator. HyperSOURCE-386 allows you to debug embedded applications. HyperSOURCE-386 provides a user interface with multi-window display areas as well as pop-up windows and pull-down menus. Commands can be input with a mouse or through the keyboard.

Organization of the Manual

This manual describes the operation of the hyperSOURCE-386 interface. The manual is divided into the following sections:

Chapter One, "Introduction," describes the hyperSOURCE-386 features, hardware and software requirements, the installation procedure, how to modify the DOS and Environment files, how to define key macros, what files are created and used in a debug session, how to invoke hyperSOURCE-386, the Help command, and how to exit hyperSOURCE-386.

Chapter Two, "Window Layout," describes the hyperSOURCE-386 display areas called windows. A sample screen display for each window is shown and a description of each window is described. Also described are how to program function and control keys, and how to use the mouse.

Chapter Three, "hyperSOURCE-386 Tutorials," describes how to execute hyperSOURCE-386 and then takes you from the basic steps to the more advanced steps of operating hyperSOURCE-386.

Chapter Four, "In-Circuit Considerations," describes how to execute hyperSOURCE-386, lists a few hyperSOURCE-386 initialization problems with their solutions, describes how to emulate ROM-based applications, RAM-based applications, and applications without target memory. Also, a few possible operation problems with hyperSOURCE-386 are described.

Chapter Five, "Debug Environment," describes how to use the high-level language debugging features supported by hyperSOURCE-386.

Chapter Six, "Command Reference," describes the hyperSOURCE-386 commands. The syntax, a brief description, and some examples are given for each command.

Chapter Seven, "Macros," describes how to use the hyperSOURCE-386 macro facility.

Table of Contents

Tables

Figures

Chapter One - Introduction

HyperSOURCE-386 is a windowed interface providing both assembly-level and source-level debugging capability through the MICE-V 80386 emulator. HyperSOURCE-386 allows you to debug embedded applications. HyperSOURCE-386 provides a user interface with multi-window display areas as well as pop-up windows and pull-down menus. Commands can be input with a mouse or through the keyboard.

Feature Summary

- HyperSOURCE-386 operates with the MICE-V 80386 or 80386SX system.
- HyperSOURCE-386 can load linked and located C source code and provide complete source-level support.
- HyperSOURCE-386 displays source code in high-level, assembly-level, or mixed-mode formats. It provides a full-stack trace, has the ability to monitor variables continuously, and executes breakpoints on source- or assembly-level information.
- HyperSOURCE-386 commands can be executed on the command line or with a mouse through pull-down menus.
- HyperSOURCE-386 has a transparent mode so you can access the low-level features of the standard command-line interface of the emulator.
- HyperSOURCE-386 provides in-depth on-line help.

Hardware Requirements

HyperSOURCE-386 requires an IBM PC AT, or compatible system with at least 2 MB of RAM, a 1.2 MB floppy drive, a hard disk drive with 1 MB of free space, and one serial port. A second port is required if you plan to use a serial mouse.

The following monitors are compatible with hyperSOURCE-386:

- Monochrome graphic display
- IBM CGA color graphic display or compatible
- IBM EGA color graphic display or compatible
- IBM VGA mono or color graphic display or compatible

Software Requirements

HyperSOURCE-386 requires MS-DOS or PC-DOS version 3.1 or higher, an assembler and/or a C compiler, and a linker capable of producing Intel OMF-386 object files.

Installation

To install hyperSOURCE-386, insert the hyperSOURCE-386 or the hyperSOURCE-386SX distribution diskette into drive A and type the following:

```
> a:install
```

This installation procedure copies all files from the distribution diskette to a hyperSOURCE-386 directory which was created during the installation procedure.

Note

The hyperSOURCE-386 product is used for both the 386 and 386SX emulators. Select and use only the appropriate installation diskette.

Modifying the DOS Files

In order for hyperSOURCE-386 to operate properly, you may need to modify your *config.sys* and *autoexec.bat* files.

DOS Configuration File - config.sys

The following commands should be included in the *config.sys* file. Refer to the file *config.new* created during installation.

1. FILES=n

   where *n* is 20 or greater. The FILES statement provides the number of simultaneously opened files that are required by hyperSOURCE-386.

2. DEVICE=MOUSE.SYS

   Include this statement if a mouse is used. The file *MOUSE.SYS* is the mouse device driver. The file name may vary but the device driver must be compatible to that of the Microsoft mouse. Consult the installation menu for the particular mouse that is installed in the system.

DOS Startup Batch File - autoexec.bat

The following commands should be included in the *autoexec.bat* file. Refer to the file *autoexec.new* created during installation.

1. PATH=drive:pathname;

   where *drive:pathname* is the drive and directory containing the hyperSOURCE-386 program file.

Instead of adding the following variables, SET HS386HLP and HS386ENV, to your autoexec.bat, you might choose to use the batch file *hs.bat* to invoke hyperSOURCE-386. If you do, move hs.bat to a subdirectory which is in your path.

2. SET HS386HLP=drive:pathname

   where *drive:pathname* is the drive and directory containing the help files for hyperSOURCE-386. HS386HLP is hyperSOURCE-386's environment variable for locating the directory of the help files. The help files are: *hs386hlp.txt*, *hs386hlp.bod*, and *hs386hlp.idx*. When hyperSOURCE-386 is started, it will search the current directory for the help files. If the help files cannot be found, it will then search the subdirectory specified in the HS386HLP environment variable.

3. SET HS386ENV=drive:pathname\filename

   where *drive:pathname\filename* is the drive, directory and file name of hyperSOURCE-386's environment file. You may specify environment variables in the environment file to configure the operation of hyperSOURCE-386. The default file name is *hs386.env* and the default directory is the current directory from which hyperSOURCE-386 is started.

Modifying the Environment File - hs386.env

HyperSOURCE-386's environment file enables you to set up internal environment variables that affect the operation of hyperSOURCE-386. The default file name of the environment file is *hs386.env*. If the environment file is not found when hyperSOURCE-386 is started, it will make use of internal defaults for configuration. You can specify environment variables in the environment file to override the default settings.

You can use the ENV command during a debug session to display the environment settings. The ENV command also allows you to save the settings to a file which can later be used as an environment file to configure hyperSOURCE-386.

Each line in the environment file may contain a command, a comment or both. The "#" character is used as the comment operator. Any text after the comment operator until the end of a line is treated as comment.

The following is an example of an environment file:

```
# IF YOU USE THE ENV COMMAND IN hS-386 TO SAVE THE ENVIRONMENT,
# THE DEFAULT IS TO OVERWRITE THIS FILE, LOSING ALL COMMENTS
#
#                                 MICE COMMUNICATION PARAMETERS
COM = 1                         # Communications port, 1 or 2
BAUD = 57600                    # Baud rate select, 300 to 57600
TMDELAY = 0x3000                # Transparent Mode data transfer rate delay
#
#                                 PATH VARIABLES
STARF = start.mac               # Macro file to execute on startup
SPATH = D:\hs386                # Path to source code files
#
#                                 DATA BUFFER SIZES
HISTORY = 50                    # Number of command lines in history buffer
DIALOG = 200                    # Size of dialog window buffer in lines
#
#                                 SOURCE WINDOW DISPLAY
CODE ON                         # Displays object code during disassembly
NUMBER ON                       # Display source file with line numbers
TAB = 4                         # Size of tab expansion in source window
EDITOR = ed                     # Program executed by EDIT command
#
#                                 AUDIO-VISUAL EFFECTS
BEEP OFF                        # Beep on error detection
EGA OFF                         # Use 43 lines on EGA; 50 lines on VGA
#
#                                 DEBUG CHARACTERISTICS
CRREPEAT OFF                    # [Enter] repeats previous command
HOME COMMAND                    # Cursor returns to COMMAND or SOURCE window
MAIN OFF                        # On load, run, then stop execution at main
MLIST OFF                       # Display macro bodies as they expand
PROLOG ON                       # Auto execute function prolog code
RADIX HEX                       # Input number base; HEX, DEC, OCT, BIN
SENSITIVE OFF                   # Case sensitivity in matching symbol names
EXTENSION = C                   # Change to LST if using Intel compilers
#
#                                 WINDOW SIZES AND POSITIONS
BARLINE = 18                    # Position of dialog/source bar line
REGWN =      1 60 23 19 CLOSE   # y x h w s
MEMWN =      4 21  7 50 CLOSE   #   where
BREAKWN =    2 45 10 18 CLOSE   # y=y_coordinate from upper left, range 0-24
RTRACEWN =   3 19 13 60 CLOSE   # x=x_coordinate from upper left, 0-79
CSTACKWN = 13 30  6 46 CLOSE    # h=height of window, 1-24
#                               # w=width, 1-80
#                               # s=status, OPEN or CLOSE
#
#                                 FUNCTION KEY DEFINITIONS
#                               # F2 to F9 (F1 & F10 reserved)
#                               # alt-F1 to alt-F10, shf-F1 to shf-F10
#                               # max. 50 keystrokes per definition
#                               # Do not put comments on F-key defin. lines
<F2> = map 0p to 1ffffp fast ram
```

The hyperSOURCE-386's environment variables in the environment file (hs386.env) above, are described in the following sections.

Specifying MICE Communication Parameters

Serial Port Number (COM)

The variable COM specifies the logical address of the serial port on the host computer for communicating with the remote target. The command syntax is as follows:

COM = n

where *n* is 1 or 2. If COM=1, COM1 is the selected port. If COM=2, COM2 is the selected port. The default is COM=1.

Baud Rate (BAUD)

The variable BAUD specifies the baud rate for the serial port on the host computer. The command syntax is as follows:

BAUD = n

where *n* is 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, or 57600. The baud rate in the hs386.env file is BAUD=57600. If your system cannot find the hs386.env file, the default is BAUD=57600.

Time Delay (TMDELAY)

The variable TMDELAY specifies the rate at which data is transferred between the MICE-V and hyperSOURCE when in Transparent Mode. You may need to alter this value if your PC is unable to display all characters correctly. The syntax is as follows:

TMDELAY = *n*

where n is a 16-bit value between 0 and 0xffff. The optimum rate varies depending on the speed of your PC. If the rate is too high, there will be a noticeable pause between display updates; if the rate is too low, characters will be dropped. The value of TMDELAY may also be changed by using the command while in Transparent Mode.

Specifying Path Variables

Startup Command File (STARF)

The variable STARF specifies the startup command file to be executed when hyperSOURCE-386 is invoked. The command syntax is as follows:

```
STARF = drive:pathname\filename
```

where *drive:pathname\filename* is the drive, pathname, and filename of the startup command file. The default is no startup command file.

Source Files (SPATH)

The variable SPATH specifies the directory path to be searched for the source files associated with the program modules, if the source files cannot be located in the current working directory. The command syntax is as follows:

```
SPATH [=] drive:pathname[,drive:pathname]...
```

where *drive:pathname* is the drive and pathname of the source files. HyperSOURCE-386 will search this path to find a C source file with the same name as the current module. It does not search SPATH for OMF files to load. These files must be in the current directory or called by pathname. The default is no source path.

You can also use the SPATH command during a debug session to display or change the search path.

Specifying Data Buffer Sizes

History Window (HISTORY)

The variable HISTORY specifies the number of command lines to be stored in the history window buffer. The command syntax is as follows:

```
HISTORY = n
```

where *n* is from 1 to 512. The default is HISTORY=40.

Dialog Window (DIALOG)

The variable DIALOG specifies the number of displayed lines to be stored in the dialog window buffer. The command syntax is as follows:

```
DIALOG = n
```

where *n* is from 43 to 512. The default is DIALOG=80.

Specifying Source Window Display Formats

Object Code (CODE)

The variable CODE specifies whether object code is displayed in the Source window. This variable applies to Assembly code only. The command syntax is as follows:

```
CODE boolean
```

where *boolean* is ON or OFF. Specifying CODE OFF allows more room on the screen for pop-up windows. Specifying CODE ON displays code in the source window. The default is CODE ON.

You can also use the CODE command during a debug session to display or change the setting.

Line Number (NUMBER)

The variable NUMBER specifies whether line numbers are displayed in the Source window. The command syntax is as follows:

```
NUMBER boolean
```

where *boolean* is ON or OFF. Specifying NUMBER ON, displays line numbers in the Source window. Specifying NUMBER OFF, turns off line numbers in the Source window. The default is NUMBER OFF.

You can also use the NUMBER command during a debug session to display or change the setting.

Tab Expansion (TAB)

The variable TAB specifies the number of spaces to be inserted when expanding a tab in the source window. The command syntax is as follows:

```
TAB = n
```

where *n* is from 1 to 8. The default is TAB=8.

Text Editor (EDITOR)

The variable EDITOR specifies the path name and program file name of the text editor. This editor will be invoked when the EDIT command is entered. The command syntax is as follows:

```
EDITOR = drive:pathname\filename
```

where *drive:pathname\filename* is the drive, pathname, and filename of the editor. If the drive:pathname portion has been specified in the DOS PATH variable, then you need only specify the program file name. The default is no editor selected.

Specifying Audio-Visual Effects

Error Beep (BEEP)

The variable BEEP specifies whether a beep tone is generated when an error is detected. The command syntax is as follows:

```
BEEP boolean
```

where *boolean* is ON or OFF. Specifying BEEP ON, generates a beep tone when an error occurs. Specifying BEEP OFF, does not generate a beep tone when an error occurs. The default is BEEP ON.

You can also use the BEEP command during a debug session to display or change the setting.

Display Mode (EGA)

The variable EGA specifies the line density of the display monitor. The command syntax is as follows:

EGA boolean

where *boolean* is ON or OFF. Specifying EGA ON, displays 43 lines on a EGA monitor and 50 lines on a VGA monitor. Specifying EGA OFF, displays 25 lines. The default is EGA OFF.

You can also use the EGA command during a debug session to display or change the setting.

Specifying Debug Characteristics

Command Repeat (CRREPEAT)

The variable CRREPEAT enables or disables the status of command repeating. The command syntax is as follows:

CRREPEAT boolean

where *boolean* is ON or OFF. Specifying CRREPEAT ON, repeats the last execution, disassembly, or memory display command when the <Enter> key is pressed. Specifying CRREPEAT OFF, does nothing when the <Enter> key is pressed. The default is CRREPEAT OFF.

Home Window (HOME)

The variable HOME specifies the home window. The command syntax is as follows:

HOME name

where *name* is COMMAND or SOURCE. Specifying HOME COMMAND, returns the cursor to the command line once a command has been executed. Specifying HOME SOURCE, returns the cursor to the source window once a command has been executed. The default is HOME COMMAND.

You can also use the HOME command during a debug session to display or change the setting.

Main Function (MAIN)

The variable MAIN specifies whether or not the Source window will switch to the main() function after a program file is loaded. The command syntax is as follows:

```
MAIN boolean
```

where *boolean* is ON or OFF. The default is MAIN OFF. With MAIN OFF, when hyperSOURCE loads an OMF file, the emulator program counter is set to the program's starting address, and hyperSOURCE displays code in the source file corresponding to this starting address in the SOURCE window. With MAIN ON, hyperSOURCE displays code in the source file containing main() regardless of the initial program counter. Note that the emulator has not actually executed any code at this time; to cause the emulator to execute from the program starting address to the entry point of main(), use the STEp or LINe command. Using these commands in this situation is equivalent to "go til main."

Display Macro Bodies (MLIST)

The variable MLIST specifies whether the macro bodies are to be expanded. The command syntax is as follows:

```
MLIST boolean
```

where *boolean* is ON or OFF. Specifying MLIST ON, displays the macro bodies on the console as macros are expanded. Specifying MLIST OFF, will not display the macro bodies. The default is MLIST OFF.

Prolog Execution (PROLOG)

The PROLOG variable enables or disables the automatic prolog execution. The prolog of a C function is the instructions at the beginning of the function that set up the local stack frame for the C function when it is entered. The command syntax is as follows:

```
PROLOG boolean
```

where *boolean* is ON or OFF. Specifying PROLOG ON, executes function prolog code automatically when the function is entered via GO or STEP. Specifying PROLOG OFF does not execute function prolog code. The default is PROLOG ON.

Number Base (RADIX)

The variable RADIX specifies the input number base (hexadecimal, decimal, octal, or binary). The command syntax is as follows:

```
RADIX base
```

where *base* is HEX, DEC, OCT, or BIN. The default is RADIX DEC.

You can also use the RADIX command during a debug session to display or change the setting. The base for output (display) is selected automatically, based on variable type.

Case Sensitivity (SENSITIVE)

The variable SENSITIVE specifies case sensitivity in matching symbol names. The command syntax is as follows:

```
SENSITIVE boolean
```

where *boolean* is ON or OFF. Specifying SENSITIVE ON, makes symbolic references case sensitive. Specifying SENSITIVE OFF, makes symbolic references case insensitive. The default is SENSITIVE OFF.

You can also use the SENSITIVE command during a debug session to display or change the setting.

Source File Extension (EXTENSION)

The variable EXTENSION specifies the default extension of your source files. The command syntax is as follows:

```
EXTENSION = extension
```

where *extension* is C or LST. The default is EXTENSION=C.

Note

Intel compilers generate include file line numbers differently from other compilers. Therefore, if you are using Intel tools, you must set EXTENSION to LST.

Specifying Window Size and Position

Dialog/Source Separator Bar (BARLINE)

The variable BARLINE allows you to specify the line number of the horizontal, 2-line dialog/source bar from the top of the window. The command syntax is as follows:

```
BARLINE n
```

where *n* is from 2 and 20. The default is BARLINE=18.

REGWN, MEMWN, BREAKWN, RTRACEWN, CSTACKWN

You can specify the size, position, and status of the register, memory, breakpoint, run trace, and call stack windows in the environment file. The command syntax is as follows:

```
variable = win
```

where *variable* is one of the window variables listed below:

REGWN	register window
MEMWN	memory window
BREAKWN	breakpoint window
RTRACEWN	run trace window
CSTACKWN	call stack window

win is an argument list that describes the size and position of the window. The arguments in the list are separated by spaces which are described as follows:

y	is the y-coordinate (vertical-axis) with respect to the upper left hand corner of the screen. Its range is from 0 to 23.
x	is the x-coordinate (horizontal-axis) with respect to the upper left corner of the screen. Its range is from 0 to 79.
h	is the height of the window. Its range is from 1 to 24.
w	is the width of the window. Its range is from 1 to 80.
s	is the status of the window. It is either OPEN or CLOSE.

The following list the window defaults:

```
REGWN = 1 60 23 19 CLOSE
MEMWN = 2 10 10 50 CLOSE
BREAKWN = 2 41 10 26 CLOSE
```

```
RTRACEWN = 2 30 8 48 CLOSE
CSTACKWN = 10 30 6 46 CLOSE
```

You can also change the size and position of the window during a debug session by pressing <alt>l in an active window.

Defining Key Macros

The key macro facility lets you associate a command line with a function key or a function key combination. So you can press a function key to enter a command instead of typing in the command.

The function keys and function key combinations that are programmable are F2 to F9, <Alt>F1 to <Alt>F10 and <Shf>F1 to <Shf>F10. The command that is associated with a function key may contain up to 50 keystrokes. These function key bindings are called key macros. Function keys F1 and F10 are reserved for hyperSOURCE-386. F1 is used to invoke the menu help. F10 is used to select the menu bar.

You can define the key macros in the environment file or in the Key menu selection of the mAcro menu. The command syntax for defining key macros in the environment file is as follows:

```
<Fx> = text string
<Alt-Fn> = text string
<Shf-Fn> = text string
```

where *x* is from 2 to 9; *n* is from 1 to 10. For example:

```
<F5> = pline
<Alt-F1> = time
<Shf-F2> = map 0p 0ffffp fast ram
```

Note

Function Key definition lines should not contain comments, as the # character is not recognized because it may be part of the macro definition.

Once you have defined a key macro, you can save the definition in the hs386.env file using the ENV command.

Files Used in a Debug Session

When hyperSOURCE-386 is started, it will make use of other files during the debug session. These files are described in the following sections.

Environment File

HyperSOURCE-386 scans the environment file, hs386.env, for internal defaults before starting its operations. Refer to the file hs386.env for how to configure your environment (source path, baud rate, etc.).

Program File

The OMF-386 demo program file was produced by Link&Locate-386 by Systems & Software, Inc. This OMF file is opened for reading upon execution of the LOAD command. This file will be closed after the loading is completed. Refer to *Linker/Locator* in Chapter Five.

Source Files

A program file may contain multiple object modules. An object module contains program code that covers a range of addresses. If the program counter is within the range of addresses of a particular object module, that object module will be called the active module. If the source file for the current module is available in the current directory, it will be opened for read access. However, if the source file resides in a directory not specified in SPATH or has a file name that is different from the module name, the SET command may be used to associate the object module with the source file. Moreover, the SPATH environment variable and the SPATH command can also be used to specify the directory that contains the source files.

Help Files

The help files contain on-line help information. The files *hs386hlp.txt*, *hs386hlp.bod* and *hs386hlp.idx* are used for this purpose. The default directory of the help files is determined by the value of the DOS environment variable HS386HLP. This variable can be defined by the DOS command SET.

Command Files

The command inputs for controlling the operations of hyperSOURCE-386 may be temporarily redirected from a command file. The command file is specified in the @ or INCLUDE command.

Files Created During a Debug Session

When hyperSOURCE-386 is started, it may create other files during the debug session. These files are described in the following sections.

Temporary Files

Up to six temporary files will be opened for hyperSOURCE-386 to use during the lifetime of the debug session. The temporary files will be created in the current directory. They all have "tmp" as file extension. These temporary files will be closed and deleted when the debug session is terminated. However, if the debug session is terminated abnormally, these temporary files will not be deleted from the current directory. Then you will have to delete them yourself.

List File

The list file is used to capture commands and their results that are displayed in the dialog window. The LIST command opens a list file. If a list file already exists, the contents will be erased and new data will be added to the beginning of the file. If the APPEND argument is specified in the LIST command and the list file already exists, new data will be appended to the end of the file. The NO LIST command closes a list file. The list file is also closed when the debug session is terminated.

Journal File

The journal file is used to capture the input commands only. The JOURNAL command opens a journal file. If a journal file already exists, the contents will be erased and new data will be added to the beginning of the file. If the APPEND argument is specified in the JOURNAL command and the journal file already exists, new data will be appended to the end of the file. If the KEYBOARD argument is specified, the journal file will store all entered keystrokes, including those entered in windows, rather than only the commands entered on the command line. The NO JOURNAL command closes a journal file. The journal file is also closed when the debug session is terminated.

Data Files

Up to six data files can be opened or created by using the hyperSOURCE-386 OPEN command. These files are general purpose and can be read and written using the READ and WRITE commands, respectively. These files will be closed with the CLOSE command or when the debug session is terminated.

Invoking hyperSOURCE-386

Before hyperSOURCE-386 can be started, the configuration must be set up as described in the previous sections. Furthermore, the MICE-V 80386 must be started first and waiting for the establishment of communication with hyperSOURCE-386.

The command to start hyperSOURCE-386 takes the following format:

```
-> hs386 [@command_file [list]] [n]
```

hs386 is the name of the hyperSOURCE-386 program with the file name hs386.exe. The program file must be accessible through the DOS path or reside in the current directory.

Up to three arguments can be supplied on the command line. The '[' and ']' used in the above command line indicate that all three arguments, *command_file*, *list*, and *n* are optional.

If the *command_file* argument is supplied on the command line, hyperSOURCE-386 will take command line input from the specified file instead of the user keyboard entries. HyperSOURCE-386 will execute one line at a time from the specified file until it reaches the end of the file. At that time, hyperSOURCE-386 will switch to command mode and start to execute commands entered from the user's keyboard.

The *n* argument is the communication port number. If *n* is not specified, the communication port is specified by the value of the COM variable in the environment file. If *n* is specified, it overrides the value of the COM variable in the environment file. n = 1 specifies the COM1 port; n = 2 specifies the COM2 port. The default baud rate is 57600 unless specified by the BAUD variable in the environment file.

Normally, hyperSOURCE-386 executes the command lines from the specified command file in quiet mode. In other words, hyperSOURCE-386 does not echo the command line read from the command file. However, if the list argument is also supplied, hyperSOURCE-386 will echo the command line obtained from the command file before executing the command line.

Help

To get help on an individual command, type HELP. To get help on the menu bar, press F10, use the cursor keys to move to the desired field, then press F1.

Exiting hyperSOURCE-386

The EXIT or QUIT command or <Alt>x may be used to end the hyperSOURCE-386 session. Before hyperSOURCE-386 terminates its operations and returns control to DOS, it closes all the files and erases all the temporary files. Thus, if hyperSOURCE-386 is terminated abnormally, it is your responsibility to remove the temporary files left behind by hyperSOURCE-386.

Chapter Two - Window Layout

HyperSOURCE-386 information is presented in selected display areas called windows. A graphic representation of the hyperSOURCE-386 window layout is shown in Figure 2.1.

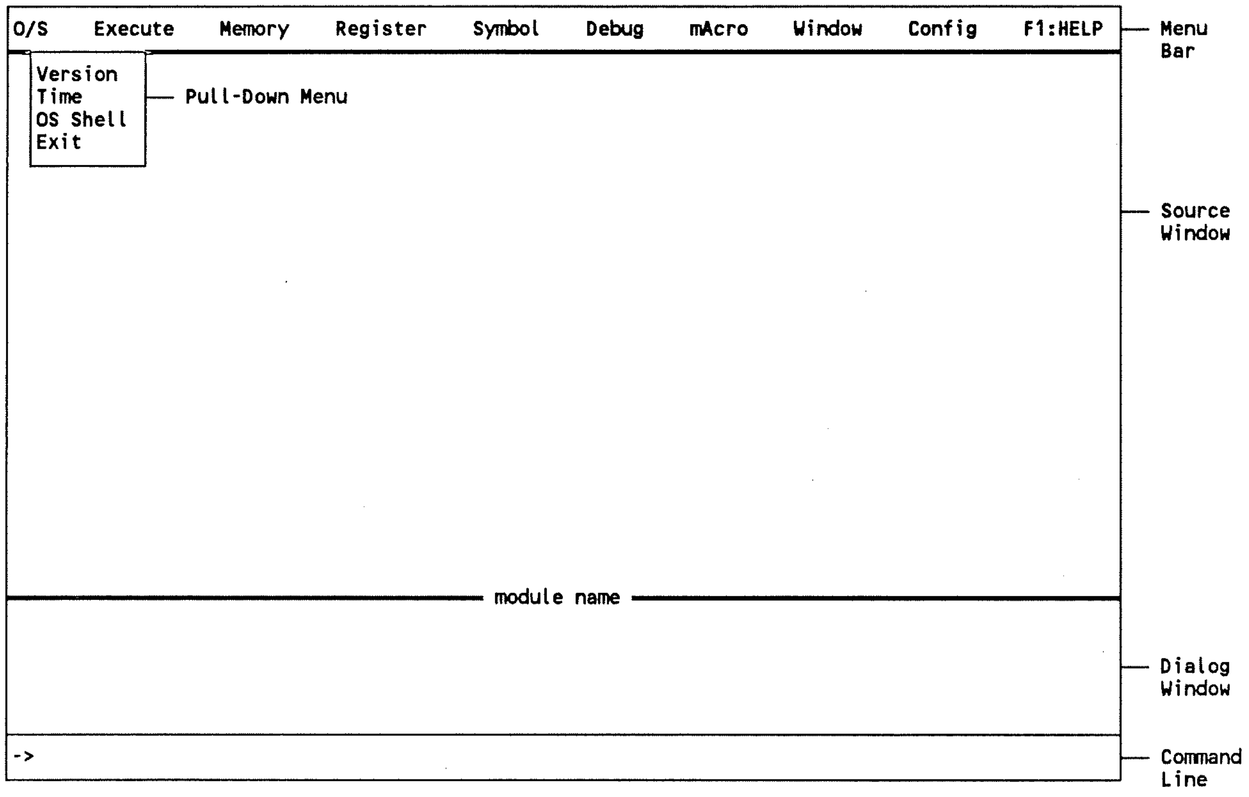


Figure 2.1 - hyperSOURCE-386 Window Layout

Menu Bar

The Menu bar occupies the top-most line of the display screen. To access the pull-down menus in the Menu bar, press the F10 function key or move the mouse to the Menu bar and click.

You can access the individual fields in the menu bar in one of two ways; either press F10, then use the left/right cursor keys, or press <Alt> and the appropriate letter key (the one shown as a capital letter) simultaneously. For example, to access the Execute menu, press <Alt>e; to access the Debug menu, press <Alt>d.

To exit from the menu bar, press the <Esc> key.

Pull-Down Menus

Pull-down menus are used to execute hyperSOURCE-386 commands. The following table lists the available hyperSOURCE-386 pull-down menus.

Table 2.1 - Pull-Down Menus

Pull-down Menu	**Description**
O/S	Host related operations
Execute	Target system functions
Memory	Memory display
Register	Register window
Symbol	Symbol access
Debug	Target execution control
mAcro	Macro processing
Window	Window display/control
Config	hyperSOURCE configuration

Source Window

The Source window occupies the middle section of the screen display area. This window displays the program source. The SPATH variable must be set to point to the subdirectory containing the source files.

Source Window Size

The size of the Source window can be changed during the debugging session at the command prompt. You can increase the Source window one display line with <Ctrl>g ("Grow") or decrease it one display line with <Ctrl>t ("Tiny"). However, increasing or decreasing the size of the Source window inversely affects the Dialog window since the combined size of Source window and Dialog window is fixed.

The width of the Source window is 80 columns. However, the width of the display buffer for the Source window is 132 columns. Therefore, when browsing in the Source window, you can use the left/right cursor keys to move beyond the 80th column to achieve the horizontal scroll effect.

Window Browse Mode

You can enter Window browse mode in one of two ways. The first way is to select Window/Select from the menu bar and move the cursor to the desired window. The second way is to use <Alt>#, where # is the window number, located in the top left hand corner, of the desired window. The Source window is always number 1 (e.g., 1:Source); the Dialog window is always number 2 (e.g., 2:Dialog). The other open window numbers vary depending on the sequence in which you opened the windows. Once in Window browse mode, use the cursor keys described in the following table to move around within the display buffer of the selected window.

Table 2.2 - Window Browse Cursor Keys

Cursor Key	**Function**
<Home>	Move the cursor to the beginning of the window
<End>	Move the cursor to the end of the window
<PgUp>	Move the display window up one page
<PgDn>	Move the display window down one page
↑	Move the cursor up one display line
↓	Move the cursor down one display line
→	Move the cursor left one character
←	Move the cursor right one character
<Ctrl>g	Increase Source window; decrease Dialog window
<Ctrl>t	Increase Dialog window; decrease Source window

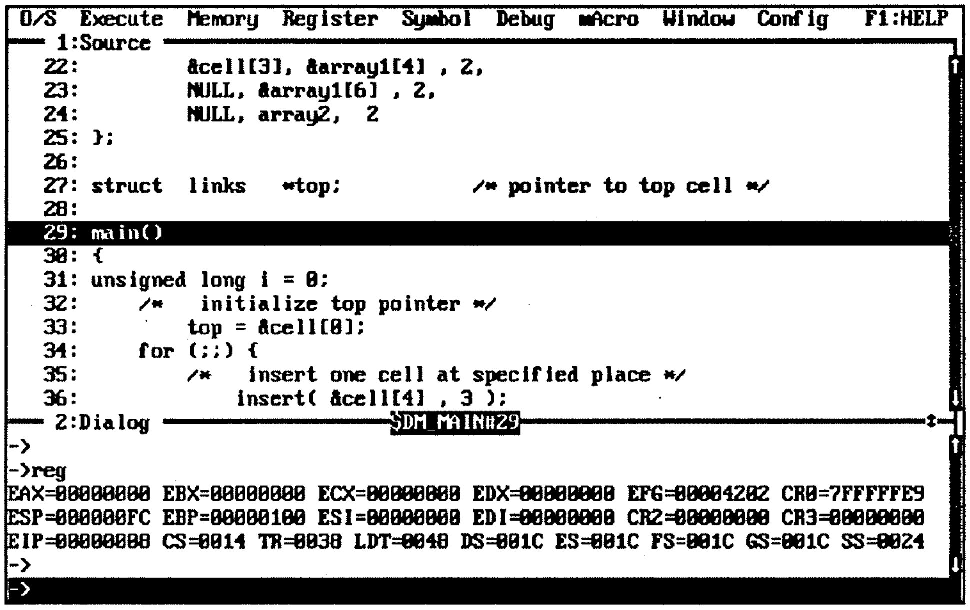
Foreground and Background Colors

The default background color of the Source window is either blue for the color display or black for the monochrome display. The foreground color is white for both displays.

The Source window is surrounded with a frame in either green background and white foreground for the color display or white background and black foreground for the monochrome display system. If the Source window displays source in high-level language statements, the module name of the source is shown on the bottom of the Source window frame.

Note

The source line in reverse video is the source line to be executed next, when not in Window browse mode. This is illustrated in Figure 2.2.



```
O/S  Execute  Memory  Register  Symbol  Debug  mAcro  Window  Config   F1:HELP
── 1:Source ─────────────────────────────────────────────────────────────
   22:          &cell[3], &array1[4] , 2,
   23:          NULL, &array1[6] , 2,
   24:          NULL, array2,  2
   25: };
   26:
   27: struct  links   *top;           /* pointer to top cell */
   28:
   29: main()
   30: {
   31: unsigned long i = 0;
   32:      /*   initialize top pointer */
   33:          top = &cell[0];
   34:      for (;;) {
   35:          /*   insert one cell at specified place */
   36:              insert( &cell[4] , 3 );
── 2:Dialog ──────────────$DM_MAIN#29───────────────────────────────────
->
->reg
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000 EFG=00004202 CR0=7FFFFFE9
ESP=000000FC EBP=00000100 ESI=00000000 EDI=00000000 CR2=00000000 CR3=00000000
EIP=00000000 CS=0014 TR=0038 LDT=0040 DS=001C ES=001C FS=001C GS=001C SS=0024
->
->
```

Figure 2.2 - Sample Display of Source Window

Display Formats

The following information can be displayed in the Source window:

1. High-level language statements
2. Disassembled instructions
3. Mixed Language - High-level language statements mixed with disassembled instructions

The Source window always displays the source (whether it is in high-level language or in assembly language) of the active program module. The active program module is either the module which contains the current program counter or is the module you selected. The default active module contains the current program counter.

The Source window will display the source in high-level language statements, if:

1. the active module has line number debugging records in the object file, and
2. the active module has a corresponding source file associated with it, and
3. the high-level language display type (VIEW command) is set for the Source window (this is the default condition).

The Source window will display the source in mixed mode if conditions 1 and 2 above are met and the mixed mode display for the Source window is set. Otherwise, the Source window will display the source in disassembled instructions.

Since the default active module contains the current program counter, hyperSOURCE-386 displays the program containing the next instructions to be executed after a program breakpoint. The Source window highlights the source of the active module with the next instruction or program statement to be executed.

Dialog Window

The Dialog window occupies the bottom of the screen between the Source window and the Command line. This window displays the results of the commands that were executed on the command line.

Dialog Window Size

Since the size of the Dialog window is limited, the lines of information to be displayed will easily fill up the entire Dialog window. For example, if the Dialog window contains only 10 display lines and the symbol table to be displayed contains 50 lines, then the first 40 lines of the symbol table will scroll up the Dialog window and only the last 10 lines of the symbol table will be visible. To resolve this problem, hyperSOURCE-386 stops the display process once the Dialog window is filled and continues the display process when you press any key. This feature can be disabled with the PAUSE command. Refer to the PAUse command in Chapter Six for more details.

Since the display area may still be too small and you may want to review the lines that have been scrolled up, hyperSOURCE-386 provides a display buffer for the Dialog window. Any display lines that are scrolled out of the Dialog window during command execution are stored in this display buffer. Consequently, the Dialog window actually holds more lines of display than its window size. To scroll through the Dialog window, select the Dialog window with <alt>2, then use the up/down and <PgUp>/<PgDn> cursor keys.

The default window size for the Dialog window is nine lines and the default size of the display buffer in the Dialog window is 80 lines. Default sizes can be changed with the DIALOG environment parameter in the environment file, hs386.env (e.g., DIALOG=100).

The size of the Dialog window can also be changed during the debugging session. The Dialog window can be increased one display line with <Ctrl>g at the command prompt or decreased one display line with <Ctrl>t. However, increasing or decreasing the size of the Dialog window inversely affects the Source window since the combined size of Source window and Dialog window is fixed.

Command Line

The Command line is used to enter hyperSOURCE-386 commands. You can execute hyperSOURCE-386 commands either by entering commands on the command line or through pull-down menus. Each menu selection has one or more submenus to guide you in the selection of appropriate commands.

When hyperSOURCE-386 is ready to accept user commands, it displays a -> prompt and a cursor on the command line. HyperSOURCE-386 is said to be in the command state. There are several options which control the hyperSOURCE-386 operations:

1. You can enter commands on the command line as follows:

   ```
   -> command [param1, [param2]...]
   ```

   Where *command* is either a hyperSOURCE-386 command or the name of a user defined macro. *param1* and *param2* are the parameters or options which complete the command.

2. To access the hyperSOURCE-386 pull-down menus, use the F10 function key or <alt>*, where * is the capital letter of the desired menu. HyperSOURCE-386 enters the menu state after you press the F10 key.

Recall a Previous Command

To recall a previous command into the command line for editing and/or execution, press the up cursor key at the command prompt. A list of the previous commands will be displayed in the Dialog window. Use the cursor keys to select the desired command and press <Enter>. The selected command will be recalled to the command line where it can be edited. Once edited, press <cr> to execute the command.

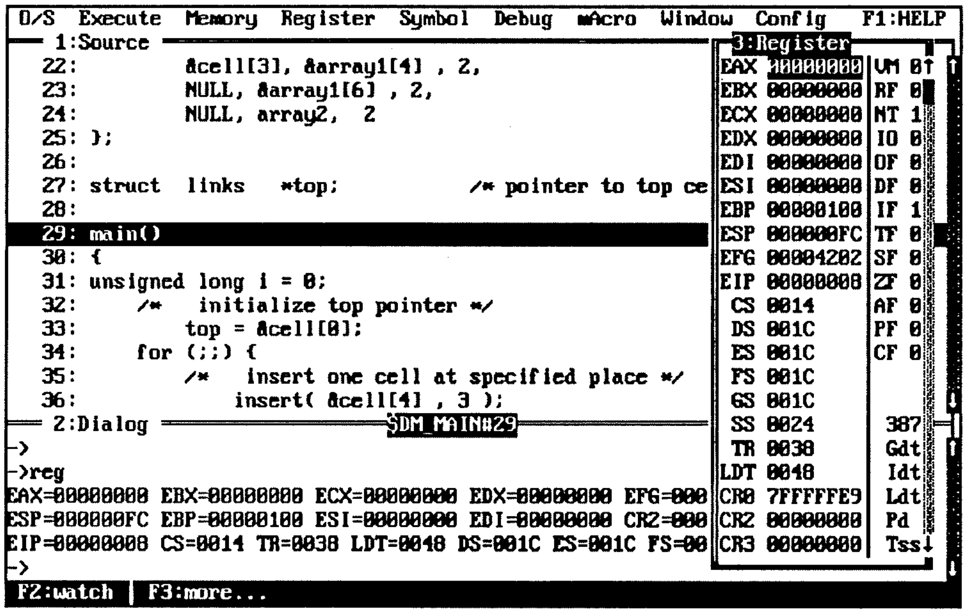
Pop-Up Windows

HyperSOURCE-386 has several pop-up windows. They are the Register window, the Memory window, the Debug windows, and the Symbol window.

HyperSOURCE-386 uses two types of pop-up windows; "sticky" and "transient." Sticky windows are used to view and/or change data structures. They remain on-screen when <Esc> is pressed, or can be closed by pressing the F2 function key. Transient windows are used only for viewing, and are closed when <Esc> is pressed.

Register Window

The Register window displays the contents of all of the CPU registers. The Register pop-up window can be accessed by <Alt>r. Registers can be changed in the window or on the command line. The registers can be monitored during the debugging session by the use of the register pop-up window. Once the register window has been selected, the F2 function key will select the register window to remain on the screen and monitor the registers. A sample Register window is shown in Figure 2.3. The Register window is "sticky."


O/S Execute Memory Register Symbol Debug mAcro Window Config F1:HELP
1:Source
3:Register
2:Dialog
$DM_MAIN#29
F2:watch
F3:more...

```
              &cell[3], &array1[4] , 2,
              NULL, &array1[6] , 2,
              NULL, array2,  2
    };

    struct  links   *top;             /* pointer to top ce

    main()
    {
    unsigned long i = 0;
        /*  initialize top pointer */
            top = &cell[0];
        for (;;) {
            /*  insert one cell at specified place */
                insert( &cell[4] , 3 );
```

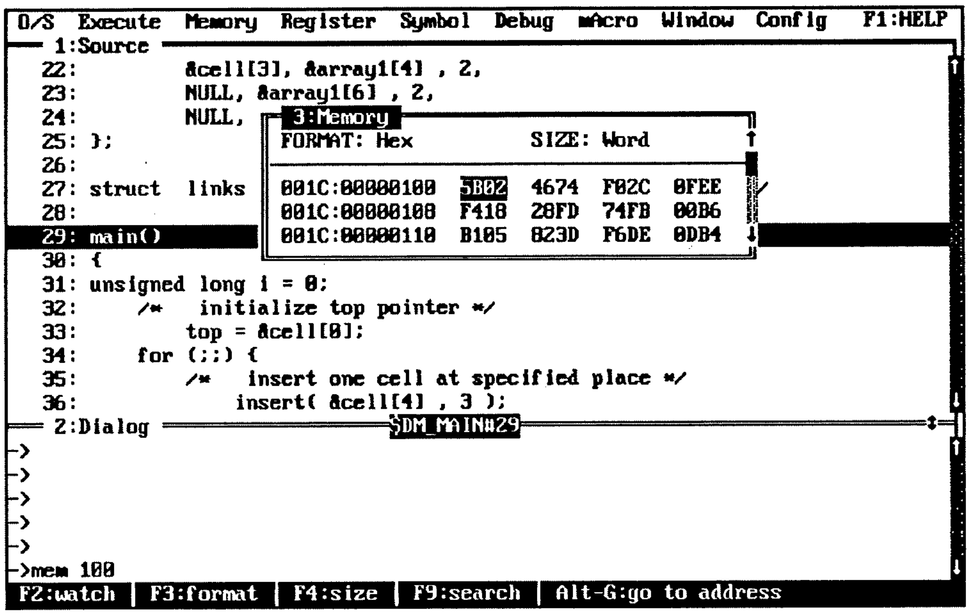
```
->
->reg
EAX=00000000 EBX=00000000 ECX=00000000 EDX=00000000 EFG=000
ESP=000000FC EBP=00000100 ESI=00000000 EDI=00000000 CR2=000
EIP=00000008 CS=0014 TR=0038 LDT=0048 DS=001C ES=001C FS=00
->
```

```
EAX 00000000 VM 0
EBX 00000000 RF 0
ECX 00000000 NT 1
EDX 00000000 IO 0
EDI 00000000 OF 0
ESI 00000000 DF 0
EBP 00000100 IF 1
ESP 000000FC TF 0
EFG 00004202 SF 0
EIP 00000008 ZF 0
 CS 0014     AF 0
 DS 001C     PF 0
 ES 001C     CF 0
 FS 001C
 GS 001C
 SS 0024      387
 TR 0038      Gdt
LDT 0048      Idt
CR0 7FFFFFE9  Ldt
CR2 00000000  Pd
CR3 00000000  Tss
```

Figure 2.3 - Register Window

Memory Window

The Memory pop-up window will allow the user to display and/or change memory contents inside the window. The Memory window can be accessed either by <Alt>m or by using the MEMory command on the command line. The F2 function key will allow the memory window to remain on the screen so the memory locations may be monitored during the debug session. A sample Memory window is shown in Figure 2.4. The Memory window is "sticky."



```
O/S  Execute  Memory  Register  Symbol  Debug  mAcro  Window  Config   F1:HELP
 1:Source
              &cell[3], &array1[4] , 2,
              NULL, &array1[6] , 2,
              NULL,   3:Memory
 };                   FORMAT: Hex            SIZE: Word

 struct  links        001C:00000100  5B02  4674  F02C  0FEE
                      001C:00000108  F418  28FD  74FB  00B6
 main()               001C:00000110  B105  823D  F6DE  0DB4
 {
 unsigned long i = 0;
        /*   initialize top pointer */
              top = &cell[0];
        for (;;) {
              /*   insert one cell at specified place */
                   insert( &cell[4] , 3 );
 2:Dialog                    $DM_MAIN#29
->
->
->
->
->
->mem 100
F2:watch | F3:format | F4:size | F9:search | Alt-G:go to address
```

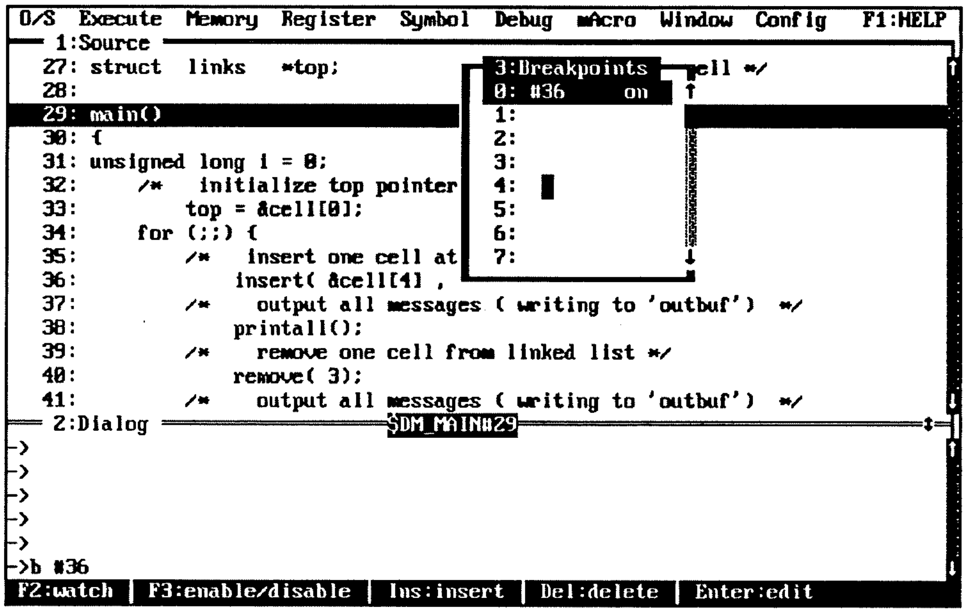
Figure 2.4 - Memory Window

Debug Windows

Two of the windows in the Debug menu can be left open to monitor debug activity. These windows are the Breakpoints window and the Callstack window. The Runtrace, BusEvent, Trigger, timebaSe, Qualify, Map, and Icemode windows are removed from the screen when their use is finished. The Debug menu is invoked with <Alt>d. Fields in the menu are selected with the up/down cursor keys and the corresponding window selected by pressing <Enter>. The F2 function key will allow the Breakpoints and Callstack windows to be monitored throughout the debugging session.

Breakpoints Window

The Breakpoints window is used to set and monitor the status of execution breakpoints used in debugging. An example of the Breakpoints pop-up window is shown in Figure 2.5.

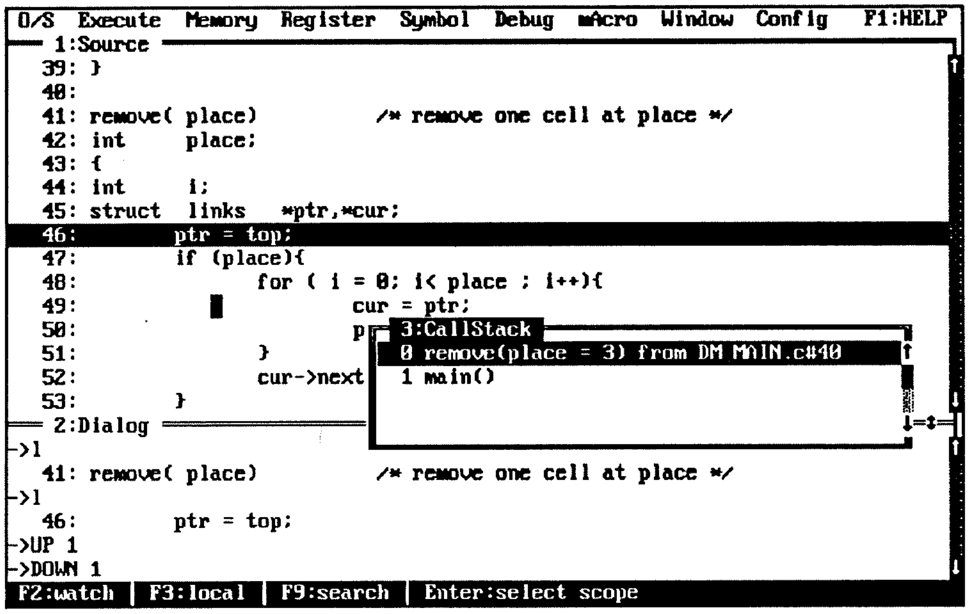


```
O/S  Execute  Memory  Register  Symbol  Debug  mAcro  Window  Config   F1:HELP
── 1:Source ─────────────────────────────────────────────────────────────────
  27: struct  links   *top;             3:Breakpoints ──ell */
  28:                                  0: #36      on  ↑
  29: main()                           1:
  30: {                                2:
  31: unsigned long i = 0;             3:
  32:     /*   initialize top pointer  4:  █
  33:          top = &cell[0];         5:
  34:     for (;;) {                   6:
  35:          /*   insert one cell at  7:             ↓
  36:               insert( &cell[4] ,
  37:          /*    output all messages ( writing to 'outbuf') */
  38:               printall();
  39:          /*    remove one cell from linked list */
  40:               remove( 3);
  41:          /*    output all messages ( writing to 'outbuf') */
══ 2:Dialog ══════════════════$DM_MAIN#29═══════════════════════════════════
->
->
->
->
->
->b #36
F2:watch | F3:enable/disable | Ins:insert | Del:delete | Enter:edit
```

Figure 2.5 - Breakpoints Window

CallStack Window

The CallStack window is used to monitor the current chain of procedure calls in the program being executed. An example of the CallStack window is shown in Figure 2.6.



```
O/S  Execute  Memory  Register  Symbol  Debug  mAcro  Window  Config   F1:HELP
 1:Source
 39: }
 40:
 41: remove( place)          /* remove one cell at place */
 42: int      place;
 43: {
 44: int      i;
 45: struct   links    *ptr,*cur;
 46:          ptr = top;
 47:          if (place){
 48:               for ( i = 0; i< place ; i++){
 49:                        cur = ptr;
 50:                        p   3:CallStack
 51:               }           0 remove(place = 3) from DM MAIN.c#40
 52:               cur->next    1 main()
 53:          }
 2:Dialog
->1
 41: remove( place)          /* remove one cell at place */
->1
 46:          ptr = top;
->UP 1
->DOWN 1
F2:watch | F3:local | F9:search | Enter:select scope
```

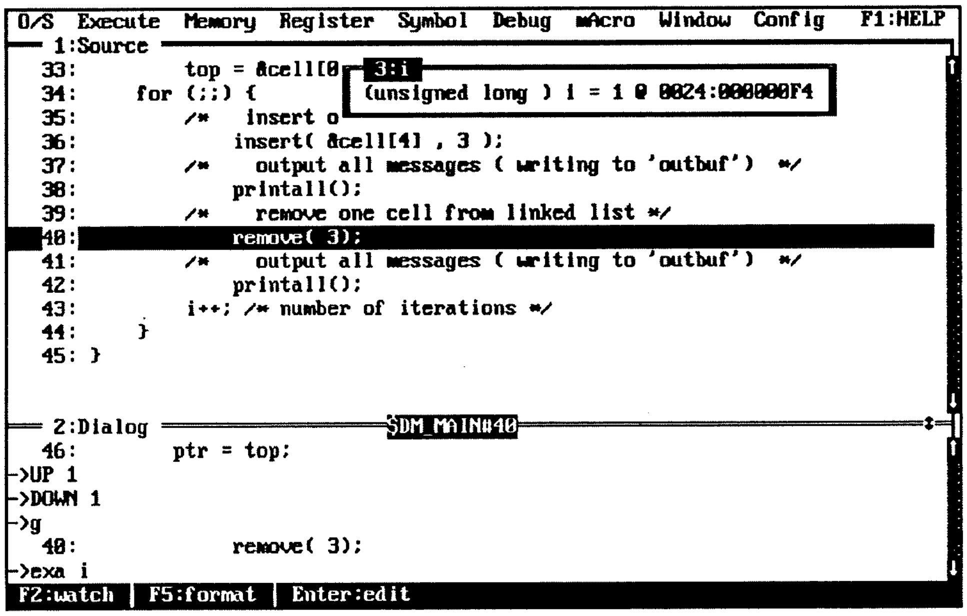
Figure 2.6 - CallStack Window

Symbol Windows

The Symbol windows are used to view and monitor program symbols, modules, structures, and variables. The only Symbol window that can be left active on the screen during a debugging session is the Examine window. The rest of the Symbol windows are closed when work in the window is finished. The Symbol menu is accessed by <Alt>s. Fields in the Symbol menu are selected with the up/down cursor keys and the associated windows are selected with the <Enter> key. The F2 function key is used to open the Examine window to monitor variables throughout the debugging session.

Examine Window

The Examine window may be used to monitor and modify program variables. An example of the Examine window is shown in Figure 2.7.



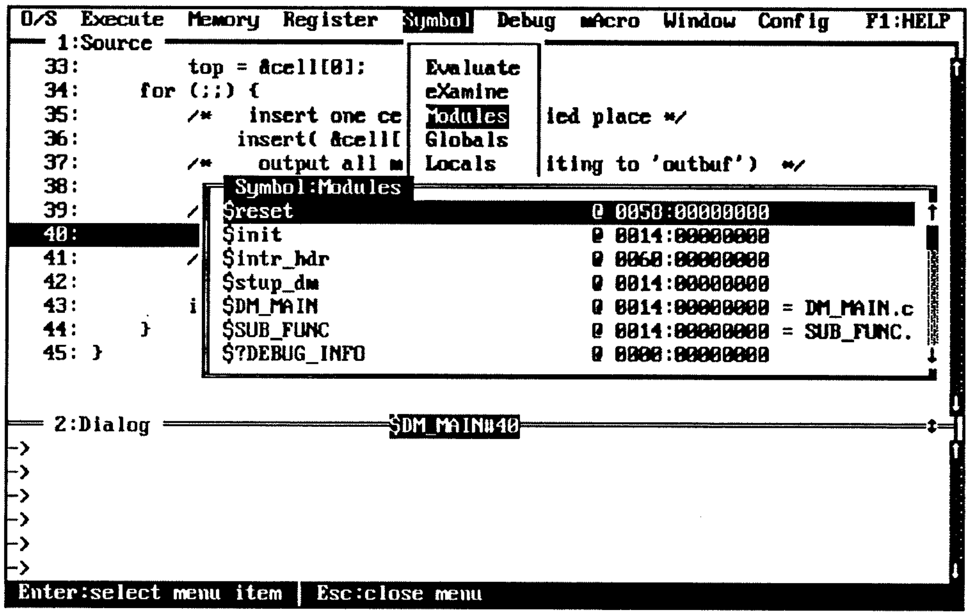
```
O/S  Execute  Memory  Register  Symbol  Debug  mAcro  Window  Config   F1:HELP
== 1:Source ==================================================================
  33:        top = &cell[0 ┌3:i────────────────────────────────────┐
  34:    for (;;) {        │(unsigned long ) i = 1 @ 0024:000000F4│
  35:        /*   insert o └──────────────────────────────────────┘
  36:            insert( &cell[4] , 3 );
  37:        /*   output all messages ( writing to 'outbuf')  */
  38:            printall();
  39:        /*   remove one cell from linked list */
  40:            remove( 3);
  41:        /*   output all messages ( writing to 'outbuf')  */
  42:            printall();
  43:        i++; /* number of iterations */
  44:    }
  45: }

== 2:Dialog ====================$DM_MAIN#40==================================
  46:        ptr = top;
->UP 1
->DOWN 1
->g
  40:            remove( 3);
->exa i
 F2:watch | F5:format | Enter:edit
```

Figure 2.7 - Examine Window

Module Window

The Module window displays all modules associated with the current program. An example of the module window is shown in Figure 2.8.



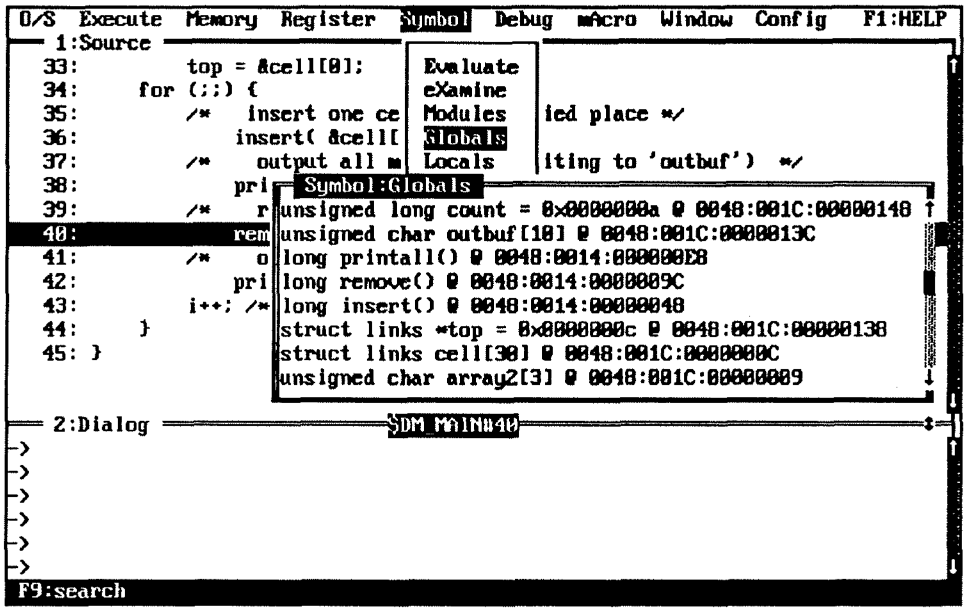
```
O/S  Execute  Memory  Register  Symbol  Debug  mAcro  Window  Config    F1:HELP
 1:Source
  33:           top = &cell[0];         Evaluate
  34:       for (;;) {                  eXamine
  35:           /*   insert one ce      Modules   ied place */
  36:                insert( &cell[     Globals
  37:           /*     output all m     Locals    iting to 'outbuf')  */
  38:            Symbol:Modules
  39:          / $reset                               @ 0050:00000000
  40:            $init                                @ 0014:00000000
  41:          / $intr_hdr                            @ 0060:00000000
  42:            $stup_dm                             @ 0014:00000000
  43:          i $DM_MAIN                             @ 0014:00000000 = DM_MAIN.c
  44:       }    $SUB_FUNC                            @ 0014:00000000 = SUB_FUNC.
  45: }          $?DEBUG_INFO                         @ 0000:00000000

 2:Dialog                          $DM_MAIN#40
->
->
->
->
->
->
Enter:select menu item | Esc:close menu
```

Figure 2.8 - Module Window

Global Window

The Global window displays all global symbols associated with the current program. An example of the Global window is shown in Figure 2.9.



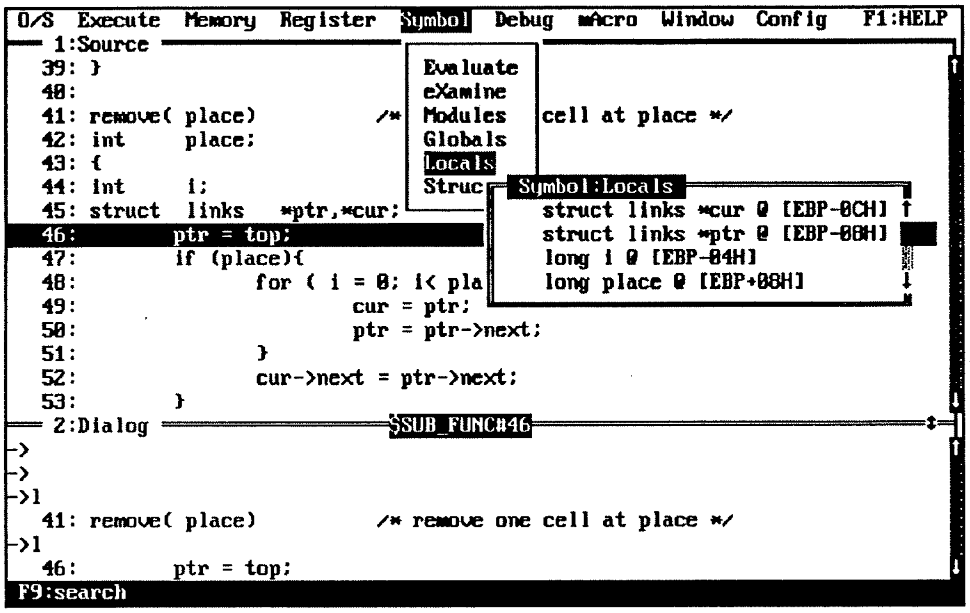
```
O/S  Execute  Memory  Register  Symbol  Debug  mAcro  Window  Config    F1:HELP
 1:Source
  33:          top = &cell[0];      Evaluate
  34:      for (;;) {               eXamine
  35:          /*   insert one ce   Modules     ied place */
  36:               insert( &cell[  Globals
  37:          /*     output all m  Locals      iting to 'outbuf')  */
  38:               pri  Symbol:Globals
  39:          /*     r unsigned long count = 0x0000000a @ 0048:001C:00000140
  40:               rem unsigned char outbuf[10] @ 0048:001C:0000013C
  41:          /*     o long printall() @ 0048:0014:000000E8
  42:               pri long remove() @ 0048:0014:0000009C
  43:          i++; /*  long insert() @ 0048:0014:00000048
  44:      }            struct links *top = 0x0000000c @ 0048:001C:00000138
  45: }                 struct links cell[30] @ 0048:001C:0000000C
                        unsigned char array2[3] @ 0048:001C:00000009

 2:Dialog                        $DM_MAIN#40
->
->
->
->
->
->
F9:search
```

Figure 2.9 - Global Window

Local Window

The Local window displays all active local symbols. An example of the Local window is shown in Figure 2.10.

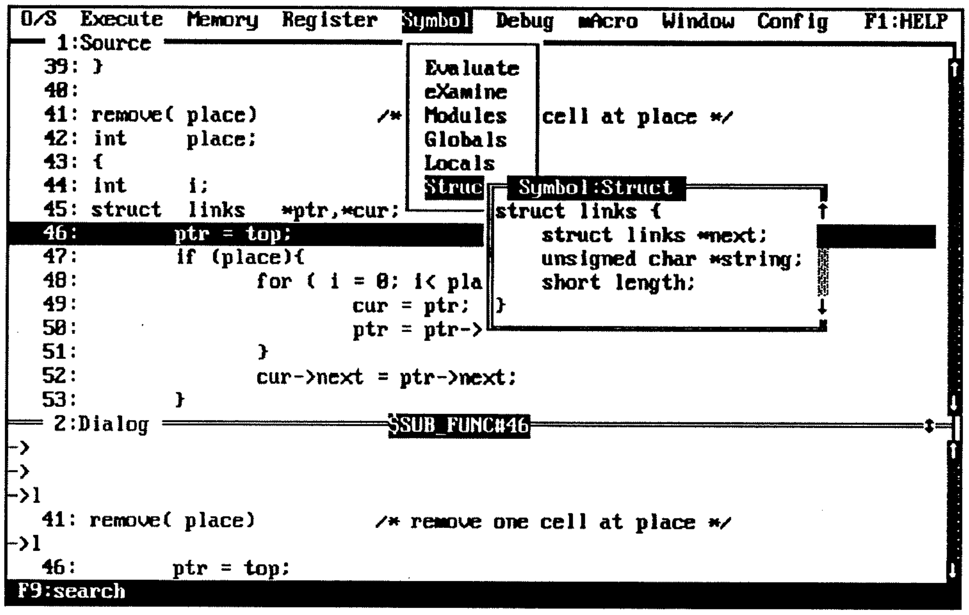


```
O/S  Execute  Memory  Register  Symbol  Debug  mAcro  Window  Config   F1:HELP
 1:Source                        Evaluate
 39: }                           eXamine
 40:                             Modules
 41: remove( place)          /*  Globals   cell at place */
 42: int     place;              Locals
 43: {                           Struc  Symbol:Locals
 44: int     i;                            struct links *cur @ [EBP-0CH]
 45: struct  links   *ptr,*cur;            struct links *ptr @ [EBP-08H]
 46:         ptr = top;                    long i @ [EBP-04H]
 47:         if (place){                   long place @ [EBP+08H]
 48:                for ( i = 0; i< pla
 49:                       cur = ptr;
 50:                       ptr = ptr->next;
 51:                }
 52:                cur->next = ptr->next;
 53:         }
 2:Dialog                     $SUB_FUNC#46
->
->
->1
 41: remove( place)          /* remove one cell at place */
->1
 46:         ptr = top;
F9:search
```

Figure 2.10 - Local Window

Structure Window

The Structure window shows all structures in the current program. An example of the Structure window is shown in Figure 2.11.



```
O/S  Execute  Memory  Register  Symbol  Debug  mAcro  Window  Config   F1:HELP
 1:Source
  39: }                                Evaluate
  40:                                  eXamine
  41: remove( place)            /*     Modules   cell at place */
  42: int      place;                  Globals
  43: {                                Locals
  44: int      i;                      Struc  Symbol:Struct
  45: struct   links    *ptr,*cur;          struct links {
  46:              ptr = top;                    struct links *next;
  47:              if (place){                   unsigned char *string;
  48:                    for ( i = 0; i< pla     short length;
  49:                              cur = ptr;   }
  50:                              ptr = ptr->
  51:                    }
  52:                    cur->next = ptr->next;
  53:              }
 2:Dialog                          $SUB_FUNC#46
->
->
->1
  41: remove( place)            /* remove one cell at place */
->1
  46:             ptr = top;
F9:search
```

Figure 2.11 - Structure Window

Function Keys

The F1 and F10 function keys are defined by hyperSOURCE-386. You can program the F2-F9, <Alt>F1-F10, and <Shf>F1-10 function keys to contain key input sequences up to 50 characters long. Function keys can replace commonly used command line inputs with a single key stroke. To create or modify these function key definitions, either modify the environment file (hs386.env) or enter the definitions via the mAcro/Key submenu.

Table 2.3 - Function Keys

Key	Function	Description
F1	Help	Invoke the on-line help facility
F2-F9		User definable
F10	MENU	Activate the pull-down menus
<Alt>F1-<Alt>F10		User definable
<Shf>F1-<Shf>F10		User definable

Control Keys

The following table lists the available hyperSOURCE-386 control keys.

Table 2.4 - Control Keys

Control Key	Description
<Esc>	Abort operation
<Ctrl>g	Increase size of Dialog window; decrease size of Source window
<Ctrl>t	Increase size of Source window; decrease size of Dialog window

Note

Both <Ctrl>g and <Ctrl>t are only recognized on the command line.

Using the Mouse

This section describes the mouse functions of hyperSOURCE-386. HyperSOURCE-386 requires a Microsoft-compatible mouse. The two buttons on the mouse have these basic functions:

- Left button: to select or accept
- Right button: to cancel

Selecting Help

You can select the help option on the menu bar with the mouse by left-clicking the **F1:Help** option on the menu bar.

For general help about hyperSOURCE-386, make sure that no other menu on the menu bar is selected before you select **F1:Help**.

If you need help for a specific menu option, select the option from the menu and select **F1:Help**. To do so, follow these steps:

1. Left-click the option you want on the menu bar.
   You see a pull-down menu or a dialog box if you select the Register or Memory option. For dialog boxes, go to step 3.

2. Double-click the left mouse button on the option you want from the pull-down menu.
   You see a dialog box or a choice-list box.

3. Left-click **F1:Help** on the menu bar and the on-line help appears for the option you selected.

Selecting Menu Options

To select an option from the menu bar, left-click the option. You see a pull-down menu for all options except for the Memory and Register options.

To select a pull-down menu option, double-click the left mouse button on the option.

To select the Memory or Register option from the menu bar, double-click the left mouse button on the option. These options do not have pull-down menus. When you choose the Memory option, a dialog box pops up asking for the memory starting address. When you select Register, the register window appears.

REMEMBER
To exit a menu, right-click!

Resizing Dialog and Source Window

To enlarge or shrink the Dialog and Source windows, left-click on the separator bar between the windows and hold the button down. Now you can drag the bar up or down to change the size of the windows.

Selecting Windows

To select a window, left-click on the window title. This highlights the window title to indicate the window is active.

Note

You cannot select the History window with the mouse. Select the Dialog window and press [↑].

Resizing and Repositioning Windows

If the window you want to resize or reposition is a display window, double-click the left mouse button on the window title. This marks the window's borders.

If you want to resize or reposition the active window, left-click the window title. This marks the window's borders.

To move a border, left-click on it and hold the button down. Now you can drag the border to another position.

The following table shows the functions of the borders in resizing or repositioning a marked window.

Table 2.5 - Window Border Functions

USE	**TO**
upper border	move window up or down (works only with decreased window height)
left border	move window left or right
lower border	resize window height
right border	resize window width (works only if you moved window to the left)

To accept the size and position of the marked window, click on **Enter/F10:accept**.

To cancel your changes, click the right mouse button.

Scrolling in Windows

To scroll up and down in windows, you use the scroll bar on the right-hand side of the window.

To move up one line, left-click ↑; to move down one line, left-click ↓.

To scroll continuously, left-click and hold the button on the arrow. The screen starts scrolling after a short delay.

You can also use the scroll indicator to scroll continuously. If you left-click the scroll indicator on the scroll bar, hold the button down, and drag the mouse, the window scrolls in the direction of the drag.

To skip from one position to another, left-click anywhere on the scroll bar. This repositions the cursor at the distance between the scroll indicator and where you clicked the scroll bar.

Selecting Function Keys

When you are in a window, you see a bar with function key options at the bottom screen. To select these keys with a mouse, left-click the option on the function-key bar.

The Source Window

The following section describes how you can use the mouse in the Source window. These mouse functions only work if you are on the command line. Do not select the Source window.

Setting a Breakpoint

Left-click on the left-hand side of the screen, on the column with line numbers or addresses. This leaves a highlight at the position where you set the breakpoint.

Deleting a Breakpoint

Left-click on the highlight indicating the breakpoint that you want to delete. The highlight disappears from the screen.

Copying Command to Command Line

Double-clicking the left button on a command or variable name copies that command or variable name to the command line. If you copy a word that is not a command, you see the error message "illegal command keyword or undefined symbol."

Viewing Source Code of Symbols

Double-clicking the left mouse button on a symbol copies that symbol to the command line with the **SOUrce** command. The Source window shows the source code where you defined the symbol.

Scrolling in the Dialog Window

To scroll up and down in the Dialog window, left-click the arrows on the scroll bar.

Note

You cannot scroll up in a dialog window without a mouse. The ↑ on your keyboard has a different function from the upward arrow on the scroll bar. If you press [↑] on your keyboard, you select the history window. You cannot select the history window with a mouse.

Copying from the History Window

To scroll up and down in the History window, left-click the arrows on the scroll bar.

To copy a line from the History window to the command line, double-click the left mouse button on the line in the history window. You see the line appear in the command line. If you want to execute the command, press <Enter>. If you want to cancel the procedure, press <Esc>.

Editing in the Memory and Register Windows

To edit a memory location in a Memory window, left-click to select the location. You can now edit the location.

To edit a field in the Register window, left-click to select the field. You can now edit the field.

Editing and Scrolling in the Symbol Menu

To edit the value of simple variables (single-line displays), left-click the variable. A dialog box prompts you to enter the new value.

To edit structs and arrays, double-click the left mouse button on the struct or array.

To display the symbols for a module, double-click the left mouse button on the module.

To display a struct definition, double-click the struct.

Defining Breakpoints

To define a breakpoint, double-click the left mouse button on the breakpoint in the breakpoint window. A choice list pops up for you to define the breakpoint.

Displaying and Defining Macros in the Macro Menu

To display a macro body, double-click the left mouse button on the macro.

To define or edit macro keys, left-click to select the function key you want to edit.

Chapter Three - hyperSOURCE-386 Tutorial

This chapter contains a tutorial for running hyperSOURCE-386. It is highly recommended that you work through this tutorial, which will introduce you to the most frequently used commands and allow you to experiment with them in a controlled environment.

Once you have completed the tutorial, you should be prepared to use the hyperSOURCE-386 and MICE-V combination in your target system.

Preparing to Run hyperSOURCE-386

To execute hyperSOURCE-386, do the following steps:

1. Connect the RS-232 cable from the COM1 or COM2 port on your PC to the TERMINAL port on the back of the MICE-V chassis.

2. Power up your PC host.

3. If you have not already installed hyperSOURCE-386, do so at this time. Insert the hyperSOURCE-386 distribution diskette into drive A and type a:install. If your RS-232 cable is connected to the COM2 port on your PC, then you must edit the environment (*.env) file. Change COM=1 to COM=2. Make sure you do not use an editor that embeds control characters.

4. Power up the MICE and invoke kermit. Verify that the emulator passes all power-up diagnostics properly.

5. Exit kermit and move to the HS386 directory.

6. Invoke hyperSOURCE-386 as shown:

```
> hs386
or
> hs386sx
or
> hs          (use the supplied batch file)
```

Note

When you first power up the MICE-V, it executes self diagnostics which take up to one minute to complete. HyperSOURCE-386 will connect when these diagnostics complete. Subsequent connections will take only a few seconds to complete.

HyperSOURCE-386 Problems and Solutions

Note

Skip ahead to "HyperSOURCE-386 Tutorial" unless you've encountered problems loading hyperSOURCE-386.

The following describes a few problems you might encounter with hyperSOURCE-386. The solutions to those problems are also described.

1. **File Problems** - HyperSOURCE-386 is unable to open the necessary files.

   a) HyperSOURCE-386 will quit and return control to the host operating system. This error can occur because:

      - The disk is full.
      - The maximum number of files that can be opened has been reached. You must increase the number by adding **FILES=20** to your *config.sys* file.
      - You do not have write privilege in the current directory.

   b) HyperSOURCE-386 is unable to load your OMF file properly. If you are using OMF files created with Intel development tools, modify the source file extension parameter in the **.env* file. Refer to "Specifying Debug Characteristics" in Chapter One.

2. **Host Memory Problems** - The host system has insufficient or unusable memory. Use a system information utility to verify the host memory configuration.

   a) HyperSOURCE-386 requires extended RAM, not expanded.

   b) HyperSOURCE-386 conflicts with *himem.sys*. HyperSOURCE-386 will initialize and appear to work, but OMF loads and other communication-intensive operations will fail.

c) Four megabytes of extended RAM should be sufficient for nearly all configurations; however, very large OMF files containing large numbers of symbols may require additional memory.

d) You may need to allocate more existing memory for hyperSOURCE-386 (e.g., log off network, remove resident (TSR) programs). Use the DOS "chkdsk" command to display the amount of free memory.

3. **Communications Link Problems** - HyperSOURCE-386 cannot establish communication with the MICE-V.

a) Make sure power is applied to the emulator. If not, turn the emulator on and re-execute hyperSOURCE-386.

b) Make sure the RS-232 cable is connected to the serial port of the emulator.

c) Verify that COM = 1 or COM = 2 in the **.env* file matches the serial communications port on your PC.

d) Make sure that the MICE-V emulator is initializing properly. Use the terminal port to establish communications with the emulator and verify power-up diagnostics.

e) Make sure that target hardware is not introducing any problems. Bring hyperSOURCE-386 up and verify operation prior to connecting to the target.

f) Try using a faster PC host. Some slow PC ATs can cause communication problems with the MICE-V.

g) Try eliminating unnecessary memory resident programs and device drivers from your *autoexec.bat* and *config.sys* files. For example, if you are not using a mouse with hyperSOURCE-386, you should remove the device driver for the mouse from your *config.sys* file.

HyperSOURCE-386 Tutorial

Once hyperSOURCE-386 has initialized with the MICE-V, you may continue with the following tutorial. This tutorial requires that a "normal" installation has been completed; i.e., the hyperSOURCE-386 executable, help files, *.env, and demo programs all reside in the appropriate subdirectories.

If this setup has been changed, the tutorial may produce unpredictable results!

This tutorial contains a C module which performs a continuous re-ordering of a linked list. Essentially, it takes the fourth element of a linked list and substitutes another element into the linked list. Then the program loops, where it once again deletes the fourth element of the linked list and substitutes once again. In each one of the link list elements is an area containing a number. These numbers begin in the sequence 1, 2, 3, 5, 5. Then after the first substitution the sequence is 1, 2, 3, 4, 5. Then it changes back and forth between the two sequences forever. It calls other procedures which are contained in the module sub_func.

Let's begin.

-> hs	Invoke hyperSOURCE-386 if you have not already done so.

Loading and Executing Code

This section of the tutorial demonstrates how to set up the emulator, load code for debugging, and execute code in various ways.

-> reset <Enter>	Type reset unless you've just powered up the emulator.
-> pause off <Enter>	Turn off screen pause.
-> map clear <Enter>	This command maps all memory accesses to the target.
-> map 0p to 0ffffhp fast ram <Enter>	Map 64K of internal overlay RAM to low memory.
-> map 0f0000 0fffffhp fast ram <Enter>	Map 64K of internal overlay RAM to high memory.
-> map 0fe000hp 0ffffffhp fast ram <Enter>	For 386SX.
-> load demo.omf <Enter>	Load the OMF86 realmode file.
-> go main	Execute from the starting address to the main C function

<Enter>	main(). You'll return and take a look at some assembly-level features later.
-> <Alt> e	Pull down the Execution menu.
<F1> <Esc> <Esc>	Press F1 to display the help screen for the execution menu. The 'S' or Step command is your main tool, used to perform high-level steps, one statement at a time, through your code. 'S IN' or Step INto, will step into statements instead of through them. Later you will use the equivalent instruction-level commands 'IS' (Instruction Step) and 'IS IN' (Instruction Step INto).
-> s <Enter>	Single-step through one statement.
-> s <Enter>	Again.
-> s <Enter>	Again, executing the entire procedure insert(). The highlight in the Source window should now be on line #38, at the procedure printall. The highlight indicates that the statement about to be executed.
-> s in <Enter>	Step INto printall. Note the source window automatically updates to show the source for printall, part of the file sub_func. The file name appears on the barline separating Source and Dialog windows.
-> s 3 <Enter>	Step three times. Line #67 in printall should be highlighted.
-> go til #72 <Enter>	Execute from the current PC to line #72 in the current module. The 'til' is optional.
-> go til ret <Enter>	Execute from the current PC until a Return from subroutine occurs, i.e., a stack pop returns you to the previous scope, main().
-> b #43 <Enter>	You'll do more with breakpoints in a later section, but for now you need to use one to show the function of 'Go Forever.'

-> go <Enter>	This causes the program to execute, stopping at the breakpoint on line 43.
-> g for <Enter> <Esc>	Go Forever starts the emulator without installing any defined breakpoints.
-> running <Enter>	Check status.
-> mem 0 <Enter>	While the emulator is running, some operations are not permitted.
-> b 0 #44 <Enter>	Ditto.
-> pri <Enter>	However, you can dump the trace buffer.
-> htrc <Enter>	The trace buffer stops acquisition when you display it. The HTRC command restarts the acquisition.
-> macro snap MD>pri 8188t MD>htrc MD>emac ->:snap	You may skip this step, but this macro definition is presented here to give you an idea of one useful way to use the 'pri' and 'htrc' commands.

Next you'll take a look at the IS, 'Instruction Step' command. This can be used in conjunction with 'View Mix' to step through code at the instruction level, but is more useful when debugging assembly level code. You'll reset the emulator and step through the assembly-level startup code.

-> reset <Enter>	You should see the code at the reset vector.
-> is <Enter>	The emulator executes one instruction, the JMP to start_.
-> is 12t <Enter>	The emulator executes 12 instructions.
-> u	Disassemble, starting at current CS:IP. Note the <Enter>

<Enter>	REP instruction about to be executed.
-> is <Enter>	The IS command recognizes this as a "high-level" statement and stops on the other side, rather than stepping into it.
	This completes the tutorial section on LOADING AND EXECUTING CODE.

Window Management

This portion of the tutorial demonstrates the various hyperSOURCE-386 data windows and pull-down menus, as well as their operation.

Choose the appropriate include file/command to set up your emulator for next portion of the tutorial.

-> inc setupr.inc list <Enter>	For 386.
-> inc setuprx.inc list <Enter>	For 386SX.
-> <F1> <Esc>	The F1 key is used to summon help. If a menu is being displayed, F1 will display help text describing that menu.
-> <alt>s e	Open the Symbols menu, and select Evaluate.
<F1>	The help window describes the type of expressions which can be entered for evaluation.
<Esc> <Esc> <Esc>	To close the help windows.
-> help <Enter>	Open the main help menu. Scroll down and select a help item by pressing Enter. You may also go directly to the help item by typing the command as an argument to help, i.e., 'help tm'. The on-line help screens are virtually identical to the command summary section of the user manual.
<Esc>	Return to the command line.

-> <alt>r (Select 386) <Enter>	Open the Register menu.
	hyperSOURCE-386 offers two types of windows: Data Windows are provided to monitor and/or modify data or debug variables. Examples are the Register window, the Memory window, the Breakpoints window, etc. These Data windows are 'sticky'; that is, they remain on screen when the <Esc> key is pressed. They can be identified by the number assigned to them (i.e., 3:Register) and their contents can be edited.
<Esc>	The Register window stays on screen.
s 4 <Enter>	The open Register window is automatically updated.
<alt>3	This shortcut can be used to select any open window, i.e., <alt># will take you to the numbered window you select, in this case the Register window, #3.
<F2>	Close the Register window.
-> <alt>s g <Esc> <Esc>	Transient Windows are provided as temporary viewports to debug information. They pop up when selected, but close immediately with the <Esc> key. They are not assigned numbers and cannot be edited.

Viewing Source Files

This section demonstrates the various ways you can use hyperSOURCE-386 to view source files.

Choose the appropriate include file/command to set up your emulator for next portion of the tutorial.

-> inc setupr.inc list <Enter>	For 386.
-> inc setuprx.inc list <Enter>	For 386SX.
-> reset <Enter>	Reset the emulator, positioning the CS:IP at the restart vector.
-> s <Enter>	Single step the processor, jumping to the start of the assembly level initialization code. Even though the current module, $startup, is known by hyperSOURCE-386, the source file is not displayed because the source file extension is .ASM, not .C. To see the assembly source file, you will manually associate the current module with it.
-> <alt>c m startup <Enter> startup.asm <Enter>	Using the hyperSOURCE-386 pull-down menus, SET the current module $startup to point to the source file startup.asm. As long as startup.asm is located in a subdirectory identified in SPATH, it will now be displayed in the Source window.
-> s <Enter>	If necessary, use the Step command to update the Source window.
-> go main <Enter>	Execute through the startup assembly code, to main().

-> <alt>1	Move the hyperSOURCE-386 cursor into the Source window. The Source window is always #1, the Dialog window is always #2. The HOME environment variable determines whether the cursor will return to the Source window or to the command line by default.
-> <pgup><pgup>	Use <PgUp>, <PgDn>, <Uparrow>, and <Downarrow> keys to scroll in the source window. Notice the current module name dm_main displayed on the barline at midscreen.
<alt>g <Enter>	<alt>g is a shortcut key to Goto a new location. Pressing Enter accepts the offered default of current CS:IP, which is currently line #31 of main().
<alt>g #43 <Enter>	Use delete to erase the default, then enter #43 and Enter to go to line number 43.
	Now, using the cursor keys, position the cursor up one line, on line #42, under the p in printall.
<F2>	Press F2 to Follow, which switches the source window into the procedure printall(). The CS:IP of the emulator has not changed, only the source window view. Notice the module name on the barline is now $sub_func. This procedure works when: 1. The procedure to be followed is part of a module with a corresponding filename with .c extension. (Assuming EXT is set to .c in the .env file.) 2. The source file is in a subdirectory which is named in the SPATH variable.
<pgup><pgdn>	You can use the cursor keys to view sub_func.c.
<alt>g <Enter>	Go (back) to CS:IP. <alt>g, "main", would also work. The cursor should still be in the Source window; if not, use <alt>1 to switch it there.

<F3>	Mixed mode shows both high-level statements and the (select Mixed) corresponding assembly-level instructions.
<Enter>	
<F3> (select Assembly) <Enter>	This display is disassembled machine code.
<F3> (select Code display) <Enter>	Code On/Off controls the display of machine code in the source window when in assembly-level display.
(select OFF) <Enter>	Turn off the code display in the source window.
<F3> (select High Level) <Enter>	Resume high-level display.
<F8>	When the cursor is in the Source window, F8 is equivalent to the S command.
<F7>	F7 is equivalent to the S IN command. (But there is no subroutine to step into at this point, so it acts like the S command.)
	F6, 'go here' causes the emulator to begin execution at its current CS:IP, stopping on the line where the cursor is located.
<F9> <Enter>	The Search key, F9, can be useful for finding an iteration specific location, symbol, etc. in the Source window.
<Esc>	Return the cursor to the command line.
-> sou printall <Enter>	The source command can be used from the command line to view any source file in SPATH.
-> sou insert <Enter>	
-> sou startup <Enter>	

-> <alt>g <Enter>	Return source window to CS:IP
<Esc>	Return the cursor to the command line.

Examining and Modifying Data

This portion of the tutorial will demonstrate several ways in which you can view and modify data, in low- to high-level constructs.

Choose the appropriate include file/command to set up your emulator for next portion of the tutorial.

-> inc sample.inc list <Enter>	For 386.
-> inc samplex.inc list <Enter>	For 386SX. Use this include file to reset the emulator to a known state, load the protected-mode demo program, and execute into main().
-> pause on <Enter>	Turn on the pause feature in the Dialog window.
-> symb <Enter>	The Symbol command shows you all the symbols currently understood by hyperSOURCE-386. These were loaded from the OMF file.
-> global <Enter>	Show only the Global symbols.
-> local <Enter>	Show Local symbols in the current module.
-> <alt>s m	Display the modules currently loaded.
<Enter> <Esc> <Esc> <Esc>	Display symbols in a module.
-> <alt>s s <Enter> <Esc>	Display structure definitions.

<Esc> <Esc>	
-> b #43 <Enter>	Set a breakpoint,
-> g <Enter>	initialize variables.
-> ?i <Enter>	Query the value of a variable.
-> i=8 <Enter>	Assign i the value 8.
-> i++ <Enter>	Auto-increment i.
-> ?i <Enter>	Verify the new value. (Should be 9!)
-> eva i <Enter>	Evaluate the current value of the variable i.
-> eva i+4 <Enter>	Various C expressions can be evaluated.
-> eva i*i <Enter>	
-> char outbuf len sizeof (outbuf) <Enter>	Evaluate a C expression.
-> local <Enter>	i is the only local variable in this module.
-> exa top <Enter>	Open an examine window for the variable 'top'.
<F4>	Follow the linked list.

<downarrow>	Select "struct links far *next".
	Walk the linked list by continuing to press <F4>:Follow and <downarrow> as desired.
<F2> <F2> <F2> <F2>	Close all open variable windows.
-> *top->next->next-> string <Enter>	Display the value in the third element of the linked list.
-> mem 100 <Enter>	View a specific address in memory.
<F2>	Close the window.
-> mem outbuf <Enter>	View the output buffer used by the procedure printall().
2038 <Enter> 2038	Enter the bytes shown into the memory assigned to outbuf.
<F4> <Enter>	Change the memory display Size to byte.
<Esc>	Return to the command line, leaving the memory window open.
-> g <Enter>	Execute through the program loop one time. Notice that the memory window now contains different data (outbuf has been written to) and that it is automatically updated at the breakpoint.
-> exa i <Enter>	Open a variable examine window to monitor the variable i, the program loop counter.
<F5>d <Esc>	Select Decimal format, then press Esc to keep it on-screen.
-> g <Enter>	Notice i is incremented on each program loop.

-> <alt>4 <F2> <alt>3 <F2>	Switch to window #4 and close it. Switch to window #3 and close it.
-> go til #36 <Enter>	The procedure insert() is about to be executed.
-> s in <Enter>	Step into insert().
-> s <Enter>	Step again to validate current stack.
-> <alt>d c	Open the Callstack window. Note that the (current) scope of insert() is highlighted.
<F3>	View local symbols in the scope of insert(). Note the value of i.
<Esc> <downarrow> <Enter>	Close the Callstack:Locals window, a transient window. Select the scope of main().
<F3>	View local symbols in main(). Notice the variable i in this scope has a different value (this i is the main program loop counter).
<Esc> <F2>	Close both windows.
-> g <Enter>	hyperSOURCE-386 is aware that you've changed the program's scope. The go starts from the current CS:IP, after restoring the proper scope. Program breaks at the breakpoint set above on line #43 of main().
-> <alt>r <Enter> <Esc>	Selects the Register window, and leave it open.
-> cx=5555 <Enter>	Change a register value from the command line. You can also do this directly in the Register window. Notice cx in the Register window is updated.

-> <alt>3 <down> <down> <Enter>	Place cursor in the Register window, select the CX register field, 1234, and enter a new value for CX.
<F2>	Close the Register window.

Breakpoints

This section will demonstrate how to access the various types of breakpoints in hyperSOURCE-386.

Choose the appropriate include file/command to set up your emulator for next portion of the tutorial.

-> inc sample.inc list <Enter>	For 386.
-> inc samplex.inc list <Enter>	For 386SX. Use this include file to reset the emulator to a known state, load the protected-mode demo program and execute into main().
-> b #33 <Enter>	Set a breakpoint on the statement on line #33. Notice the line number is highlighted, indicating the presence of a breakpoint.
-> b <Enter>	By default, hyperSOURCE-386 uses a software breakpoint.
-> help b <Enter> <Esc> <Esc>	The help page on breakpoints will explain when and how hardware execution breakpoints are used.
-> b exe #36 <Enter>	Force a hardware execution breakpoint on line #36.
-> b <Enter>	Debug registers are used to implement hardware breakpoints. If hyperSOURCE-386 cannot set a software breakpoint because the address is in ROM, it will automatically set a hardware breakpoint.

-> go <Enter>	Load software breakpoints into emulator memory, and start emulation, breaking on the software breakpoint on line #33.
-> view mix <Enter>	Software breakpoints are implemented by inserting INT3 instructions in the code. They are inserted only when you type go, and are removed after a breakpoint before the prompt is displayed. Thus, you will never see INT3's unless they are in your user code. If that happens, hyperSOURCE-386 will report a spurious breakpoint, i.e., one it did not place.
-> view hl <Enter>	Resume high-level display.
-> <alt>d b	Open the breakpoint window. HyperSOURCE-386 supports 32 software breakpoints, each of which can have a conditional statement and/or count.
<Ins>	Insert a new breakpoint,
#38	at line #38.
<F10>	Accept the definition.
<downarrow>	Highlight breakpoint 1: on line #36.
<F3>	Disable it.
<downarrow>	Highlight breakpoint 2: on line #38.
<downarrow>	
<Enter>	Edit it.
<down> <down>	Move cursor to the CONDITION field.
i>5	Type in the condition i>5.
<F10>	Accept it.
<F2>	Close the breakpoint window.
-> i=2 <Enter>	Set i to 2 before running the test.
-> g <Enter>	Start emulator. The program will run, stopping at line #38 each time it is encountered long enough to evaluate the condition. If not true, emulation resumes. When the condition evaluates true, emulation halts.

-> ?i <Enter>	Query the value of i (should be 6).
-> i=2 <Enter>	Reset the value of i to 2. These conditional breakpoints are powerful, but they require stopping the emulator long enough to evaluate the condition.
	Simple breaks can be defined and executed in realtime by using the MICE TRIGx:WHEN command in transparent mode.

Trace Analysis

In this section you will use the MICE RTA board to capture and analyze data which was captured in the realtime trace buffer.

Choose the appropriate include file/command to set up your emulator for next portion of the tutorial.

-> inc setupr.inc list <Enter> or	For 386.
-> inc setuprx.inc list <Enter>	For 386SX.
	Use this include file to reset the emulator to a known state and load the real-mode demo program.
-> go main <Enter>	Execute to main().
-> go #43 <Enter>	Execute through the program one time to initialize the loop counter variable, i.
-> xlt &i <Enter>	Note the physical address in memory where the variable i is stored.
-> tm	Enter transparent mode.
> trig0: when addr 0nnnP then brk <Enter>	Replace the "nnn" in this command with the actual physical address of i obtained above. This command sets a MICE-V complex breakpoint to break emulation whenever the address

	of the variable i is accessed.
> htrc trig0 <Enter>	Make trig0 an active trigger.
^A	End transparent mode and return to hyperSOURCE-386.
-> go <Enter>	Run the program. The program will break. Even though hyperSOURCE-386 is not aware of any breakpoints, it will sync to the program location of the current instruction pointer.
	Note that i has been incremented.
-> pri <Enter>	Display the last 10t cycles of the trace buffer in the dialog window. The bus event which caused the break (access to address nnn) is seen several cycles before the end of the buffer. The additional cycles are due to emulator "skid" and the process of coming-out-of-emulation.
^G^G^G	As needed to expand the Dialog window.
-> go <Enter>	Go one time around the program loop.
-> <alt>d r	Load the trace buffer into a hyperSOURCE-386 Runtrace window. By default, only the last 100 frames are uploaded. You can use the F3:SetMax key to change this value, up to a maximum of 8192 frames.
<alt>l <down> <down> <Enter>	'L'ocate the Runtrace window down two lines to uncover the variable window, #3.
<Esc>	Leave the Runtrace window open.
-> go <Enter>	Run program, causing it to make one loop and break again. The open Runtrace window will be automatically updated.
-> tm <Enter>	Enter transparent mode.

> trig0:when addr 0nnnP then trc or when addr 0nnnP data 0xxxx1234 then brk <Enter>	Substitute the actual physical address of i for the "nnn" as above. This command sets up the trace to capture only bus cycles with this address, and then causes a break when when the data value xxxx1234 appears at that address. (Enter the don't care x's as shown.) Omit xxxx for 386SX.
> htrc trig0 <Enter>	Make trig0 active.
^A	Exit transparent mode.
-> go <Enter>	Run the program. When the value xxxx1234 appears at the address of i, the program will break and the Runtrace window will be automatically updated with the last 100t frames of the buffer. This time, the buffer contains only actions involving the address of the variable i.
-> reset	Reset the processor.
-> quit	Terminate the debug session.

This completes the hyperSOURCE-386 tutorial. Refer to the MICE-V manual for additional information about setting up complex triggers.

Chapter Four - In-Circuit Considerations

This section will cover how to:

- plug the MICE-V 80386 in-circuit probe into your 80386 target system
- get your 80386 target and MICE-V 80386 emulator up and running
- halt emulation

Preparing to Run hyperSOURCE-386

To execute hyperSOURCE-386, do the following steps:

1. Turn off the MICE-V 80386 emulator.
2. Turn off the target system.
3. Remove the 80386 microprocessor chip from the target.
4. Plug the 80386 in-circuit probe (ICP) module into your target board matching pin 1 of the probe (pin 1 has been removed from probe) to pin 1 of your target board socket.
5. Connect your PC to the MICE-V 80386 emulator using either a RS-232 cable (Channel A) or the parallel interface (Channel B). Refer to the MICE-V 80386 User's Manual for details. If you are connecting to the emulator using either the COM2 port or the parallel interface, you must modify the environment file. Refer to "Specifying MICE Communication Parameters" in Chapter One.
6. Power up the PC host.
7. Install hyperSOURCE-386 if not already done.
8. Power up the MICE-V 80386 emulator.
9. Power up your target.
10. For hyperSOURCE-386 to support source-level debugging, it needs access to your source files. If they are not in the current directory, then SPATH must be set in the hs386.env file. Refer to "SPATH" in Chapter Two for further details.

11. Invoke hyperSOURCE-386:

```
-> hs386
```

Note

When you first power up the MICE-V 80386, it executes self diagnostics which take up to one minute to complete. HyperSOURCE-386 will connect when these diagnostics complete. Subsequent connections will take only a few seconds to complete.

If hyperSOURCE fails to initialize properly, refer to "HyperSOURCE Problems and Solutions" in Chapter Three.

Emulating ROM-based Applications

A ROM-based application assumes that your target system can initialize itself from ROM-based code. For applications that are ROM-based, you can begin immediately with the following command sequence. If you need to load code to support your target, skip ahead to "Emulating RAM-based Applications."

```
-> sig e                      //Enable all target signals
-> reset                      //Initialize the emulator to run target code as if the target
->                            //was just powered up.
-> gr                         //Start emulation from reset.
```

Once you have determined that your target system has initialized correctly, continue with the following:

```
-> <Esc>                      //Return to prompt.
-> halt                       //Halt emulation and automatically display registers and the
                              //next instruction.
```

If the system does not seem to be operating correctly, skip ahead to "Possible Operation Problems with hyperSOURCE-386."

Once you have successfully halted, you can display registers, disassemble code, dump memory, single step, and use execution breakpoints. Refer to Chapter Six for an in-depth discussion on commands such as: DASM, GO, INPUT, OUTPUT, REGISTER, and STEP.

Emulating RAM-based Applications

In a RAM-based application you must load code through hyperSOURCE-386 in order for your target system to be operational. You can load code with the following command sequence.

```
-> sig e                    //Enable all target signals
-> reset                    //Cause the emulator to run target code as if the target was
->                          //just powered up.
-> load "<file>"            //Load either omf or boot format.
```

> If the load is unsuccessful, make sure that your target was initialized so that code can be loaded. Many targets have special initialization sequences before RAM is enabled. Often this involves writing a certain value to an I/O port or booting with a ROM-based program. Use the BYTE, WORD, or DWORD command to ensure that you can communicate with the target memory that you are trying to load.

```
-> go                       //Start emulation from the current cs:eip as initialized by the
->                          //loaded file, or use the [from <address>] option to
->                          //indicate the starting address.
```

-OR-

```
-> reset                    //Reset the emulator.
-> gr                       //Go from reset.
```

Once you have determined that your target system has initialized correctly, continue with the following command:

```
-> <Esc>
-> halt                     //Halt emulation and automatically display registers and the
->                          //next instruction.
```

If the system does not seem to be operating correctly, skip ahead to "Possible Operation Problems with hyperSOURCE-386."

Once you have successfully halted, you can display registers, disassemble code, dump memory, single step, and use execution breakpoints. Refer to Chapter Six for an in-depth discussion on commands such as: DASM, GO, INPUT, OUTPUT, REGISTER, and STEP.

Emulating without Target Memory

In applications where no target memory is available or you wish to map emulator memory over target memory, such as ROM, use the MAP command. To map memory and load code, do the following command sequence.

```
-> map <start addr>p          //Map a block of memory from
   <end addr>p fast ram
->                            //<start addr> (physical) to <end addr> internal
->                            //(emulator memory)
-> sig e                      //Enable all target signals.
-> reset                      //Cause emulator to run code as if the target was just
->                            //powered on.
-> load "<file>"              //Load either OMF or boot format
-> go                         //Start emulation from the current cs:eip as initialized by the
->                            //loaded file, or use the [from <address>] option to
->                            //indicate the starting address.
-OR-
-> reset                      //Reset the emulator.
-> gr                         //Go from reset.
```

Once you have determined that your target system has initialized correctly, continue with the following command:

```
-> <Esc>
-> halt                       //Halt emulation and automatically display registers and the
->                            //next instruction.
```

If the system does not seem to be operating correctly, skip ahead to "Possible Operation Problems with hyperSOURCE-386."

Once you have successfully halted, you can display registers, disassemble code, dump memory, single step, and use execution breakpoints. Refer to Chapter Six for an in-depth discussion on commands such as: DASM, GO, INPUT, OUTPUT, REGISTER, and STEP.

Possible Operation Problems with hyperSOURCE-386

If after attempting the previous emulation sessions, the hyperSOURCE-386 does not appear to be operating correctly, it could be due to any of the following:

1. The signals are not enabled. Use the SIG command to display a list of the current target signal settings.

2. The MICE-V 80386 clock is not synchronized with your target system clock. Type the following sequence to synchronize your target system clock and the hyperSOURCE-386 clock:

   ```
   -> reset                    //Reset the hyperSOURCE-386.
   -> gr                       //Start emulation from reset.
   ```

   If your target system still does not operate properly, turn your target system power off and back on, or press the reset button on the target if one is available.

3. It is possible that unused hardware signals are floating. If the state of a hardware signal is unknown, then turn off that signal not being used (e.g., if NMI is not used then set NMI=OFF). Use the SIG command to display a list of the current target signal settings.

4. Your target system may have a watchdog timer in use. If you have a watchdog timer and the 80386 does not respond in a certain amount of time, the watchdog timer may assert reset, hold, or both of these signals. Normally the watchdog timer will have timed out before the hyperSOURCE-386 initializes. Therefore, when you attempt to enable the signals, the emulator will hang. To accommodate this type of target, do one of the following steps:

   a) Disable the watchdog timer.

   b) Start emulation before the watchdog timer times out by doing the following:

   - Turn your target system's power off and back on again.

   - Next, type the following commands before the watchdog timer times out (i.e., type the commands at a rather quick pace):

     ```
     -> sig e             //Enable all target signals.
     -> reset             //Reset the hyperSOURCE-386.
     -> gr                //Start emulation from reset.
     ```

     After the system initializes, you can halt the emulator with no further interferences from the watchdog timer. **In a RAM-based application with a watchdog timer, you must proceed to this point before you can successfully load code.**

5. The target system boots up correctly, but when you enter the HALT command, hyperSOURCE-386 reports "Cannot Halt Target Processor." This error could be caused by one of the following:

a. If you target is in protected mode: Confirm that you have set the correct IDT and GDT values and BRkPidt and BRKgdt are enabled and BRkRidt is disabled.

b. If your target is in real mode: Confirm that you have BRkPidt and BRKgdt disabled and BRkRidt enabled and set to 0p.

Exiting hyperSOURCE-386

The EXIT or QUIT command or <Alt>x may be used to end the hyperSOURCE-386 session. Before hyperSOURCE-386 terminates its operations and returns the control to DOS, it closes all the files and erases all the temporary files. Thus, if hyperSOURCE-386 is terminated abnormally, it is your responsibility to remove the temporary files left behind by hyperSOURCE-386.

Chapter Five - Debug Environment

This chapter discusses the high-level language debugging features supported by hyperSOURCE-386. HyperSOURCE-386 supports files which were compiled using the Metaware High C compiler or the Intel C compiler. Detailed steps which are used to create an 80386 demonstration debug program are described in this chapter.

Symbolic Reference

In hyperSOURCE-386, all variables and program objects can be referenced using the same symbolic name and data type that are defined in the original source program. Furthermore, hyperSOURCE-386 supports the same high-level language expression evaluation and assignments. Thus, manipulations of program objects can be done in hyperSOURCE-386 to verify program execution or to change program logic.

Referencing Symbols

If a program is written in C language and contains register variables, these variables cannot be referenced in hyperSOURCE-386.

Compiling, Linking, and Locating a 80386 Program

The following sections describe how to create an 80386 demonstration debug program for hyperSOURCE-386. There are six source files used to create the demonstration program:

init.asm	used to perform low-level
intr_hdr.asm	initialization of the environment
reset.asm	and put the 386 processor
stup_dm.asm	in protected mode
dm_main.c	the main portion of the program
sub_func.c	contains routines used by the main program

There are three steps you must follow to recreate this demonstration program. First you need to assemble the *.asm* source files using Microsoft's Assembler (MASM). Second, you need to create the object modules using Metaware High C compiler. Finally, you need to create the OMF-386 format file using Systems & Software's XLINK386. A batch file in the hs directory, *sample.bat*, performs all the required steps in creating the demo. Refer to the batch file and the following paragraphs for details.

Compiling with MASM

To create a hyperSOURCE-386 object file from an assembler source file, use the following Microsoft assembler (MASM) command syntax:

```
-> masm /Zi /Mx init.asm
```

/Zi tells the assembler to use the CodeView symbol format.

/Mx tells the assembler to make public and external names case sensitive.

The symbol record in the object file preserves the letter case of the symbol name in the source file. For example, if the symbol is defined in lowercase, the symbol record will also be in lowercase. An underscore is prefixed to all global (or extern) symbols; local (or auto) symbols are not affected. To access an assembler symbol during a debug session, an underscore must be prefixed to global symbols. Refer to the SENSITIVE command in Chapter Six if using uppercase symbols.

Compiling with Metaware High C

To create the hyperSOURCE-386 object modules from the C source files, use the following Metaware High C compiler (HC386) command syntax:

```
-> hc386 /g /c filename.obj filename.c
```

/g tells the compiler to include symbols.

/c specifies that the .obj file will be the output rather than an .exe file. This parameter is required since XLINK386 will be used, not the Microsoft linker.

Character Set

The character set is used to create the command language vocabulary. Valid characters include the alphabetic characters (A through Z, and a through z), three special characters (_, @, and ?), and the numeric characters (0 through 9).

Defining Symbols

A symbol is a user-defined name consisting of a string of alphanumeric characters. Symbols are normally used in hyperSOURCE-386 to reference memory locations and can be created in the original program or defined during a hyperSOURCE-386 session. If a symbol is defined in the original program, the corresponding symbolic information must be included (using appropriate switches during compilation, linking, and locating) in the absolute object file. When the object module in the absolute object file is loaded, the symbolic information is automatically stored into hyperSOURCE-386's internal symbol table. To display the symbol table, use the SYMBOL command.

To define a symbol during a hyperSOURCE-386 session, use the TYPE command. The rules for defining a symbol are the same as the C conventions. Each symbol name is unique up to the first 31 characters in length. The first character must be an alphabetic character or one of the following three special characters: _, @, or ?. Character case sensitivity is observed by default or when it is explicitly set with the SENsitive ON command.

Note

The ? character is treated as an operator instead of a character by the EVALUATE command.

Data Type

Each program variable in hyperSOURCE-386 is referenced by a symbol name and is associated with a data type specifying its intended usage and the operation(s) that may be performed on it.

Primitive data types supported by hyperSOURCE-386 are listed below.

BYTE	8-bit unsigned integer
CHAR	8-bit signed integer
WORD	16-bit unsigned integer
DWORD	32-bit unsigned integer
QWORD	64-bit unsigned integer
SHORT	16-bit signed integer
LONG	32-bit signed integer
FLOAT	32-bit single-precision floating point number
DOUBLE	64-bit double-precision floating point number
TREAL	80-bit floating point number
POINTER	16-bit address object that can point to BYTE, WORD, DWORD, FLOAT, DOUBLE, TREAL, CHAR, SHORT, LONG, and user-defined structures

Additional data types which are recognized by hyperSOURCE-386 but cannot be used explicitly to access their memory contents are PROCEDURE, LABEL, and high-level language statement numbers. A procedure is equivalent to a program function and is referenced the same as the function name.

User-defined data types, such as structures, may further contain fields of primitive data types or user-defined structures.

With the data type information, you can refer to the symbolic name and its qualifier to obtain its value using the current type. For example, if the following structure is declared in a C source program:

```
struct inpblk {
        char iocmd;
        char *iobuf;
        int iosiz;
} myblk;
```

then, the structure and each field can be referenced as follows:

```
-> myblk
-> myblk.iocmd
```

And, its value can be changed using direct assignment:

```
-> myblk.iocmd = 67
```

Specifying Symbols

If a symbol is encountered in the command line, hyperSOURCE-386 first determines whether it is a hyperSOURCE-386 command keyword. If it is not, it will be treated as a program variable. However, there are occasions when a program variable may have the same name as a hyperSOURCE-386 command keyword. Furthermore, a program variable can be defined in multiple C modules with the same name.

When this condition occurs, use one of the following symbols to resolve the symbol reference conflicts.

$	module name prefix
##	function name prefix
#	symbol name or line number prefix

For example:

```
$M##F#S
$M##F#24
```

If a prefix is used in the symbol reference, the method of symbol search is from outer block to inner block under the domain of module or function, if any.

If no prefix is used, that is, only the symbol name is used, the method of symbol search is from current active function to outer blocks. In other words, only active symbols are referred to. This convention is like the one adopted by the C language.

Note that if symbols are referenced without specifying any prefix, the symbols may be erroneously treated as hyperSOURCE-386 command keywords.

HyperSOURCE-386 supports block-structured programming languages. Symbols can be referenced in the same structure as the original program. To fully reference a symbol, the module name and the name of the embedding block function must be specified. The syntax is as follows:

```
        [$module name][##function name] [#] symbol name
```

For example, assume that the original program has the following block structure:

```
/* file M */
extern int G;
        int A,C,D;
void B1 (void) {
        int A,B,G;
        { /*unnamed block B2 */
                int A,C,F,
        }
}
void B3 (void) {
        int A,B,E;
}
```

Variables A, C, and D are declared in module M, which has two functions B1 and B3. Function B1 has an inner unnamed block B2 in which variables A, C, and F are declared. If you are currently in function B1 (but not within the unnamed block B2), variable A can be referenced by any one of the following forms:

```
        $M##B1#A
        ##B1#A
        #A
```

The only way to reference variables in unnamed blocks is to be in that block. If you are in the unnamed block B2 above, A can be referenced the same way as the last example, since the innermost A is currently active. There is no way to access A in block B1 while still in B2.

If incomplete module or function names are used, hyperSOURCE-386 will search from the current function outward and check the module that contains the global symbols last. For example, if you are in function B1, but not within the unnamed block B2:

- C will reference the symbol $M#C
- ##B2#F will reference the symbol $M##B1##B2#F
- B will reference the symbol $M##B1#B
- ##B1#G will reference $M##B1#G
- #G will reference $M##B1#G (if you are outside B1, in function B3 for example, #G will reference the global #G)

Source Line Number Reference

Line numbers for source statements are generated by the high-level language compiler and can be referenced by the following format:

```
        [$module name]#line number
```

For example:

```
-> go til $INIT#24          //Break on line number 24 of the INIT module.
```

Pointer Reference

Since a pointer contains the address of an element, it is possible to access the element indirectly with the unary operators "*" and "&". Thus, a variable of pointer type gives the address of an object, "*" gives the value of that object, and "&" gives the address of that object.

For example, BYTPTR is of type POINTER (located at address 40H:40H) to a variable with type of CHAR (located at address 20H:53H and its content is 67):

```
-> bytptr
      0040H:0040H POINTER TO CHAR (20H:53H)
-> &bytptr                 //Get the address of 40H:40H.
-> *bytptr
      20H:53: CHAR (67T)
```

HyperSOURCE-386 supports pointers with up to seven levels of indirection. For example:

```
-> *******seven_levels_pointer
-> ***aaa[4][3][2][2]
```

hyperSOURCE-386 supports multi-dimensional arrays with up to seven dimensions. For example:

```
-> one_dim_array[4]        //Access 5th element.
-> two_dim_array[2][3]     //Access 3rd column, 4th row.
-> seven_dim_array[1][3][4][2][3][4][5]
```

Data Structure Reference

Structures are user-defined data types which may contain fields of primitive data types as well as structures. Members of data structures can be referenced by the following format:

```
     structure_name.member_name
-OR-
     pointer_to_structure->member_name
```

For example:

```
-> parmblock.iobuf                   //iobuf is a structure member.
-> str_ptr->iobug                    //str_ptr is a pointer to structure.
-> str_ptr->i_link->i_link->i_link   //Nested reference.
```

Radixes

All numerical data entered as parameters, addresses, or data are assumed to be in decimal integer format unless a radix suffix is specified. Permissible radixes and their corresponding suffixes are as follows:

T - decimal (default suffix)
H - hexadecimal
Q - octal
Y - binary

Any hexadecimal number must be prefixed with a zero if the first digit is not a decimal digit. For example, 7EH is a legal hexadecimal number; A5H is not. It must be entered as 0A5H.

You can change the default radix with the RADIX command. For example:

```
-> radix h                //Change to hexadecimal radix.
-> radix t                //Change to decimal radix.
-> radix q                //Change to octal radix.
-> radix y                //Change to binary radix.
```

Memory Object Reference

The basic memory unit is a byte or an 8-bit unsigned number. Each memory object is referenced with a unique address using a segment and offset construct. A range of memory locations can be displayed or modified with the BYTE command. Memory objects are normally referenced using their address in conjunction with a data type. The CHAR, WORD, DWORD, DOUBLE, FLOAT, TREAL, and POINTER commands are used to display or modify memory locations in terms of the corresponding data types. For example:

```
-> byte &my_data                 //A byte at address &my_data.
-> word 2040h to 2080h           //Display 16-bit memory objects from address 2040H
->                               //to 2080H based on DS.
-> double &start=1.0, 2.0        //Enter two double values starting at location
->                               //&START.
-> dword &start len 8=2          //Fill 8 dwords from &START with 2.
-> byte &start = "THIS IS A TEST"
-> poi my_point                  //Display a pointer value.
-> poi iopb.i_buff = &my_buffer  //Load address of my_buffer.
-> copy &locl=arrayl len 4       //Copy memory.
```

I/O Port Reference

The INPUT command reads from the input ports and the OUTPUT command writes to the output ports. If the argument "D" is specified in the command, the size of the I/O port is 32-bit; if "W" is specified, the size is 16-bit; otherwise, it is 8-bit. For example:

```
-> input 20               //Read from 8-bit port #20.
-> input 20 w             //Read from 16-bit port #20.
-> input 20 d             //Read from 32-bit port #20.
-> input 20h:0=20         //Read byte value from port #20 and place it at location 20H:0.
-> output 2Ah=11h         //Write to 8-bit port #2AH.
-> output 2Ah=1011h w     //Write to 16-bit port.
```

Register Reference

The REGISTER command is used to examine or change the values of the CPU registers. The CPU registers can also be referenced with the following keywords:

Keyword	Description
EAX	Accumulator registers
EBX	Base registers
ECX	Count registers
EDX	Data registers
ESP	Stack pointer
EBP	Base pointer
ESI	Source index
EDI	Destination index
EIP	Instruction pointer
ES, FS, GS	Extra segment registers
CS	Code segment register
SS	Stack segment register
DS	Data segment register
EFG	Flags register
TR	Task register
LDT	Local descriptor table register
CR0	Contains system control flags, which control modes or states of the processor.
CR2	Page fault linear address
CR3	Page directory base register.

The following registers can be accessed from the Register window using the F3 function key:

387	387 co-processor registers
GDT	Global descriptor table
IDT	Interrupt descriptor table
LDT	Local descriptor table
PD	Page directory
TSS	Task state segment

For example:

```
-> reg              //Display all register values.
-> reg eax=2        //Set EAX to 2.
-> eax              //Display value of EAX.
-> eax=3            //Set EAX to 3.
-> cs:eip           //Display program counter.
```

Status Flag Reference

The FLAG command is used to display or change the values of the CPU status flags. The status flags can also be referenced with the following keywords:

Keyword	Description
AF	Auxiliary carry flag
CF	Carry flag
DF	Direction flag
IF	Interrupt enable flag
IOPL	I/O privilege level
NT	Nested task flag
OF	Overflow flag
PF	Parity flag
RF	Resume flag
SF	Sign flag
TF	Trap flag
VM	Virtual 8086 mode
ZF	Zero flag

For example:

```
-> flag          //Display all status flags.
-> flag if=1     //Set IF flag.
-> if            //Display value of IF.
-> df=1          //Set DF.
```

Operands

Operands can be numerical constants, variable references, location references, or CPU registers. Their values in expressions are as follows:

- Numerical constant - depends on the data type
- Variable reference - depends on the type of the variable
- Location reference - a 32-bit value
- CPU register - depends on the register. For example, CF has a binary value of 0 or 1.

Operators

There are two types of operators: unary operators and binary operators. All expressions can contain any combination of unary and binary operators and the evaluation of the expression is based on predefined precedence rules (refer to Table 5.1).

Arithmetic Operators

+	Unary plus
—	Unary minus (2's complement)
+	Addition
—	Subtraction
*	Multiplication
/	Division
%	Modulus

Relational Operators

==	Is equal to
>	Is greater than
<	Is less than
>=	Is greater than or equal to
<=	Is less than or equal to
!=	Is not equal to

Logical Operators

! or NOT	Logical NOT
&& or AND	Logical AND
|| or OR	Logical OR
^^ or XOR	Logical exclusive OR

Bitwise Logical Operators

&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	Left shift
>>	Right shift
~	1's complement (unary)

Assignment Operators

a = b	Assigns the value of b to a
a += b	Addition - assigns the value of a+b to a (a=a+b)
a -= b	Subtraction - assigns the value of a-b to a (a=a-b)
a *= b	Multiplication - assigns the value of a * b to a (a=a*b)
a /= b	Division - assigns the value of a / b to a (a=a/b)
a %= b	Modulus - assigns the value of a % b to a (a=a%b)
a &= b	Bitwise AND - assigns the value of a & b to a (a=a&b)
a |= b	Bitwise OR - assigns the value of a ¦ b to a (a=a¦b)
a ^= b	Bitwise exclusive OR - assigns the value of a ^ b to a (a=a^b)
a <<= b	Left shift - assigns the value of a shifted b places left to a (a=a<<b)
a >>= b	Right shift - assigns the value of a shifted b places right to a (a=a>>b)

Miscellaneous Operators

++	increment
--	decrement
*	indirect reference of pointer
&	address
.	struct field reference
->	pointer to struct
sizeof	size of type or variable
:	separator for specifying address base and offset

Type Operators

A type operator is used in an expression either to cast the value of a variable of a certain data type to another data type or to override the value of a variable with no type to the specified data type. The type operators are as follows:

BYTE, CHAR, WORD, SHORT, DWORD, QWORD, LONG, FLOAT, DOUBLE, TREAL, and STRUCT

For example:

```
-> a = (word)b + (word)c << 4
-> real = (treal)fa / (treal)fb
-> value = *(byte *)ptrword + *(word *)ptrbyte
```

Note that type operations may override or cast a variable. If the variable is of NULL type, it is an override operation, i.e., the variable will be overridden to the type specified; otherwise it is a cast operation, i.e., the variable value will be cast to the type specified.

For example, if doublevalue is a double type variable, then:

```
-> doublevalue = 9.99999
-> eva doublevalue
        9.99999000000000000e+0
> eva (char)doublevalue
        0000000000001001Y 11Q 9T 0009H
```

For example, if nulltype is a NULL type variable, then:

```
-> byte &nulltype = 12H,34H,56H,78H,3FH,3FH,3FH,3FH,3FH,3FH
-> eva (char)nulltype
        0000000000010010Y 22Q 18T 0012H
-> eva (short)nulltype
        0001001000110100Y 11064Q 4660T 1234H
-> eva (double)nulltype
        4.76792331334520E-4
```

Note

The arithmetic operator "%" and all bitwise logical operators may not be applied to floating point numbers.

Operator Precedence

The table below summarizes the rules for precedence and associativity of all operators.

Table 5.1 - Operator Precedence

Operator	Associativity
() [] -> .	left to right
! ~ ++ -- - (type) * & sizeof	right to left
* / %	left to right
+ -	let to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right

^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= etc.	right to left
,	left to right

Expressions

hyperSOURCE-386 supports high-level language expression evaluation. Variables can be assigned new values using expressions. In fact, you may enter a C expression on the command line. For example:

```
-> i = 4                    //Initialize i to 4.
-> my_ptr = my_ptr->i_link
-> i++
-> i_buf[i++] = *c_ptr++
-> j=eax * 2 + i
-> ecx = 4
-> evaluate A + B  + 7
```

In hyperSOURCE-386, zero produced from expression evaluation equals FALSE and non-zero equals TRUE. Expressions are used in commands that involve conditional tests such as the IF, SWITCH, FOR, WHILE, and REPEAT commands and in commands that set breakpoints or specify conditional command execution. Expressions can also be used to change the contents of a program variable, or to compute the index value for an array reference.

There are four types of expressions that can be used in a hyperSOURCE-386 command:

- Numerical expression
- Address expression
- Boolean expression
- CPU register expression

The expression syntax consists basically of operators and operands. Expressions are evaluated from left to right taking into account operator precedence (refer to Table 5.1). Parentheses may be used within expressions to explicitly delineate the operator's precedence.

Numerical Expressions

The operands in numerical expressions may be constants (real or integer) or variables. The expressions are evaluated in the precedence order of each operator defined in hyperSOURCE-386. Numerical expressions are generally used in assignment statements and in the EVALUATE commands.

Address Expressions

An address expression contains at least one address operand. Address operands can be specified in segment and offset combination or referencing the locations of variables.

Each address is a 32-bit value represented by a 16-bit segment and 16-bit offset. For example:

```
-> byte &start          //Display byte.
-> word cs:1000h        //Display WORD object at absolute location CS:1000H.
-> my_ptr = &start      //Assign the address value to a pointer.
```

Boolean Expressions

Boolean expressions are used in flow control commands that involve conditional tests such as IF/ELSE, REPEAT/UNTIL, WHILE/EWHILE, FOR/EFOR, and SWITCH/CASE. They produce a result of either TRUE or FALSE.

CPU Register Expressions

A CPU register expression deals with the CPU registers and status flags. The result is either TRUE or FALSE. The syntax is as follows:

```
register_name == data
```

register_name = EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, EFG, EIP, CS, DS, ES, FS, GS, SS

```
flag_name == one_bit_value
```

flag_name = CF, PF, AF, ZF, SF, TF, IF, DF, OF, IOPL, NT, RF, VM.

one_bit_value = 0, 1.

Any numerical data entered is assumed to be in decimal notation unless followed by a radix specifying otherwise. Radix suffixes are as follows: H (hexadecimal), T (decimal), Q (octal), and Y (binary). The RADIX command defaults to decimal notation. If you want to use another base, you can either suffix the decimal number with a radix suffix, or you can change the base to another base with the RADIX command.

The data operand is a 16-bit unsigned integer value. "x" can be used to mask out some don't care conditions. For example, 1011001111110001Y, 1110000000XXXXX0Y, 37547Q, 35X6XQ, 0A45CH, 6DEXH, 0X45XH, 0X45X, 389T.

Note that masking is not allowed in the decimal format. For example, 3X9T is not allowed. For example:

```
EAX == 20
ECS == 458H
EBX == 0X11X110Y
IF == 1
```

Chapter Six - Command Reference

This chapter details each command used in hyperSOURCE-386. The following syntax rules apply to all commands:

[ ]	Square brackets indicate that the parameter(s) is optional. If you do not specify a parameter, the default is used. For most hyperSOURCE-386 commands, the default is the value or setting that was used the last time the command was executed.
{ }	Curly braces indicate that the parameter(s) is optional. However, you MUST specify one of the optional parameters in curly braces.
...	An ellipsis indicates that you can repeat an item.
|	A vertical bar means either/or. Choose one of the separated items and type it as part of the command. For example, *ON|OFF* indicates that you can type either ON or OFF, but not both.

! (escape to command shell) !

Syntax: ! [command_string]

Function: The ! command lets you escape to the operating system's command shell. To leave the shell and return to the debugger, type EXIT from the shell. You can directly spawn a command by specifying the command after the !.

If a DOS command preceded by a ! is part of an include file, that command will be processed without waiting for a key to be pressed when the include file is run from within hyperSOURCE-386.

Examples:

```
->!                 //Escapes to DOS shell.
C>dir               //DOS command to display file directory.
C>exit              //Returns to the debugger.
->!cc test.c        //Spawns "cc test.c" directly.
```

See Also: FREe

#　　(specify variable name)　　#

Syntax: [$module_name][##procedure_name]#variable_name
[$module_name]#number

Function: # is a parser operator which can be used to access a variable outside of the current scope. It is also used as a prefix for line-numbers.

When used as a variable name prefix with no other prefix, i.e., "#variable", the variable is searched for using the normal scoping rules. This provides a method to circumvent conflicts with commands or keywords.

A module name and procedure name may be optionally specified in order to access a specific variable outside of the current scope. To access a global symbol, use the root module $. For example, $#foo refers to the public instance of variable foo.

Finally, when used as a numeric prefix, # indicates a line-number.

Examples:

```
->#USER = 4               //Assigns 4 to a variable called USER.
->$#USER = 4              //Assign to the global variable USER.
->? $MOD1##PROC1#VAR1     //Display a local variable to a procedure.
->$mod1#static1 = 3       //Assign to a variable local to a module.
->G #14                   //Go to line 14 in the current module.
->B $MOD1#14              //Break at line 14 in another module.
```

See Also: ##, $, &, EVAluate, EXAmine, SYMbol

(specify procedure name) *##*

Syntax: [$module_name]##procedure_name[#variable_name]

Function: *##* is a parser operator which allows the specification of a procedure name outside of the current scope.

If a module name precedes the *##* operator, only procedure names within that module are searched. If no leading module name is specified, only procedure names in the current module are searched.

Typically, *##* is needed only if two or more modules have local procedures with the same name.

Examples:

```
->B $MODULE1##PROC3      //Set a breakpoint at proc3 in module1
->GO FROM $##FOO         //Go from global procedure foo
```

See Also: #, $, &, B, Go, SYMbol

$ (specify module name) $

Syntax: $module_name

Function: $ is a parser operator which distinguishes module names from other symbolic names. If no module_name is specified and $ is used in conjunction with # or ##, then $ refers to the root module containing only global symbols.

Examples:

```
->SET $MOD2=SOURCE.C     //Associate a source file with a module.
->? $MOD1##PROC1#VAR1    //Display a local variable to a procedure.
```

See Also: #, ##, $, &, DIRectory MODule, EVAluate, SET, SYMbol

& (symbol address) &

Syntax: &symbol

Function: & is an address operator which gives the address of a symbolic object.

Examples:

```
->GO FROM &START FOREVER
->POINTER &POINTER_BUF LENGTH 20
->TYPE long *ptr_to_long, long_buf[8] at &buf
```

See Also: #, ##, $, &, B, EVAluate, Go, SYMbol

// (specify comment) //

Syntax: //comment

Function: Indicates that everything that follows "//" until the end of a line is a comment, and should not be interpreted as part of the command.

Examples:

```
->//A comment line
->DIR MAC          //Display directory of macros
```

: (invoke macro) :

Syntax: :macro_name [actual_parameter_list]

Function: Executes the specified macro. A macro may accept up to ten parameters which are separated by commas. "<*" and "*>" can be used as parentheses in the actual parameter list to delimit a parameter that contains commas.

The colon character is also used in macro definitions to specify a label.

Examples:

```
->:AA 34, <* AX,BX,CL *>, #20 //3 actual parameters
->:BB                         //No parameter
->:AA :BB, <* AL,5,3.5*BX *>, :CC
```

See Also: DIRectory MACro, DISplay MACro, EDit, EDit MACro, INClude, MACro, MLIst, PUT, REMove MACro

:: (line continuation) ::

Syntax: command_line ::
continuation_of_line

Function: Allows the continuation of a command line to the next line.

Example:

```
->BYT &array[12] = 1941, 42, 1776, 1812, ::
::>1492, 2001
```

= or EVAluate = or EVAluate

Syntax: {=} expr

{EVAluate}

Function: Evaluates an expression. All register and flag names are recognized. Full C syntax is supported. Moreover, the following key words are recognized:

NOT, AND, OR, XOR, BYTE, CHAR, WORD, SHORT, DWORD, LONG, FLOAT, DOUBLE, TREAL, SIZEOF.

Note that the QWORD key word is not allowed.

You can also evaluate a function in your program that returns a value. If a symbol name has the same name as the register or flag names, it should be prefixed with the # operator in the expression.

The following key words are allowed in the SIZEOF operator and type casting:

BYTE, CHAR, WORD, SHORT, DWORD, LONG, FLOAT, DOUBLE, TREAL.

Examples:

```
->eva 'A'
->= ax + 1
->eva ++i              //value of i is incremented by 1
->eva i + 1            //value of i is unchanged
->eva *(byte *)ptr_to_short + *val
->= fact(3) + 6
->eva  a * b >> 2
->eva sizeof(count)    //a symbol
->eva sizeof(long)     //a type
->eva count and 1      //logical and
->eva short(i)
->= i or j             //logical or
->eva ptr->size
->eva ary[1].length + count
->=  (double)realno + c
->=  ax | 148fh
->eva #ax              //ax is a variable in program module.
```

See Also: ?, SYMbol

? (examine symbol) ?

Syntax: ? lvalue

Function: Displays the type, value, and address of an lvalue. An lvalue is any C expression which can be used on the left-hand side of an assignment statement.

The value is displayed in a format consistent with the type. Alternative formats may be selected when the value is displayed using the EXAmine command.

Example:

```
->? *(char *)0     //Display address 0 as a character
->? foo            //Display foo
->? array[1]       //Display array element
->? *ptr           //Display data referenced by pointer
```

See Also: =, EXAmine, SYMbol

@ or INClude

Syntax: {@} ["]file_name["] [LISt]

{INClude}

Function: Executes the commands from the specified file. If LISt is specified, the commands in the command file are displayed on the console as they are being executed (default is NO LIST).

Remark: This command is identical to the INClude command.

Examples:

```
->INC "INIT.MAC" LIS
->@ C:\TMP\GR1.INC
```

See Also: JOUrnal, LISt, MACro

B (breakpoint)

Syntax:

```
B [n]  [OFF  ]
       [ON   ]
       [CLEar]
B [n] [=] addr [COUnt num] [IF (expr)] [THEN command]
```

where n = 0 to 31t.

Function: Breakpoints are used to specify addresses at which program execution is halted. The breakpoints are effective in all execution commands except for GO FOREVER.

B by itself will display all breakpoints; specifying n alone will display a specific breakpoint. OFF, ON, and CLEar are used to disable, enable, or remove a breakpoint, respectively. A disabled breakpoint will be inactive until enabled. Normally, a breakpoint is enabled when first set, but if an error occurs when setting a breakpoint, the breakpoint may be defined but disabled.

If no breakpoint number is specified when setting a breakpoint, the first unused breakpoint is allocated. If no breakpoint number is specified with the CLEar qualifier, the breakpoint at the current execution pointer is cleared. If no breakpoint number is specified with the ON or OFF qualifiers, all breakpoints are enabled or disabled. The assignment operator (=) is only required when the distinction between a breakpoint number and an address is ambiguous.

When a break is encountered and COUnt is specified, execution resumes automatically unless the event has been encountered num times. A conditional breakpoint condition may be specified using the IF qualifier. The condition can be any expression accepted by the EVALuate command. If the expression evaluates to non-zero after the rest of the break condition is satisfied, then execution will be halted, otherwise execution resumes.

A command may be specified which is executed after a breakpoint is hit using the THEN qualifier. IF and THEN qualifiers are not evaluated until after all other conditions are satisfied, including the COUnt condition. Complex command sequences may be invoked after a breakpoint by invoking a macro in the THEN statement.

Remarks: The B command uses software breakpoints to implement breakpoints, unless the address of the breakpoint is in ROM. If so, a debug register breakpoint is used. If you want to set a breakpoint based on bus-level events involving address, data, status, counter or logic module conditions, you have to define trigger specifications in transparent mode. Use the TM command to enter transparent mode, then use the TRIGx:WHEN command of MICE-V to define triggers.

Examples:

```
->B                 //Display all breakpoints.
->B 0               //Displays contents of breakpoint 0.
->B 0 CLE           //Clears contents of and disables breakpoint 0.
->B 0 OFF           //Disables breakpoint 0 but leaves it defined.
->B 0 ON            //Re-enables breakpoint 0.
->B OFF             //Disables all breakpoints but leaves them
->                  //defined.
->B 0 = 100h        //Sets breakpoint 0.
->B #12             //Sets an arbitrary breakpoint at line 12.
->B &FOO EXE COUNT 2     //Breaks after FOO is reached twice.
->B subr1 IF (i > 3)     //Breaks if i > 3 when entering subr1
->B cs:100h THEN ? i     //Display i before executing at 100h
->B subr3 IF (i > 3) THEN :mac1 a,42
```

See Also: CAUse, Go, IStep, Step, TM

BEEp BEEp

Syntax: BEEp [ON]
[OFF]

Function: Enables, disables, or displays the status of the error beeper.

Examples:

```
->BEEP
->BEEP ON
->BEEP OFF
```

See Also: ENV

BINary

Syntax:	BINary
Function:	Sets the default input radix to binary or base 2.
Example:	`->BIN`
See Also:	DECimal, HEX, OCTal, RADix

BREak BREak

Syntax: BREak

Function: Causes an immediate exit from the REPeat-UNTil, WHIle-EWHile, FOR-EFOr loop, or SWItch-ESWitch block.

Examples:

```
->macro test1   //Define a macro.
MD>switch( c )
MD>   case 30:
MD>   case 20:
MD>         c = a + b * c;
MD>         d = d / c + 8;
MD>         break;
MD>   default:
MD>         c >>= 1;
MD>eswitch
MD>emacro
->while 1
CD>   c = a - 1
CD>   if !C
CD>         break
CD>ewhile
```

See Also: CONtinue, FOR, GOTo, IF, REPeat, SWItch, WHIle

BRKgdt

Syntax:	BRKgdt [physical_addr] [BS16] [E] / [D]

```
BRKgdt [physical_addr]  [BS16]    [E]
                                  [D]
```

Function: Sets the global descriptor table address that the emulator will use for protected mode breakpoints.

'physical_addr' is the physical address (followed by the letter 'p') of the global descriptor table (GDT) when the processor is running in protected mode. The power-up default is BRKgdt 0p d.

BS16 indicates that the interrupt table resides in 16-bit memory. If BS16 is not specified, it is assumed that the interrupt table resides in 32-bit memory. E enables and D disables recognition of fetches from the global descriptor table at breakpoints.

Remarks: The BRKgdt command must be enabled before any type of protected mode breakpoints can be used.

Use BRKgdt only once during initial setup to configure the emulator to a particular target configuration.

This command is the same as the BRKGDT command of MICE-V 386.

Examples:

```
->BRK 400p e
->BRK
```

See Also: BRkPidt, BRkRidt, Go, GR, HALt, RBRk, TKB

BRkPidt BRkPidt

Syntax: BRkPidt [linear_addr physical_addr] [BS16] [E]
[D]

Function: Sets the interrupt descriptor address that the emulator will use for protected mode breakpoints.

'linear_addr' is the linear address (followed by the letter 'n') of the interrupt descriptor table (IDT) when the processor is running in protected mode. The power-up default is BRKgdt 0p d.

'physical_addr' is the physical base address (followed by the letter 'p') of the interrupt descriptor table (IDT) when the processor is running in protected mode. The power-up default is BRkPidt 0n 0p d.

BS16 indicates that the interrupt table resides in 16-bit memory. If BS16 is not specified, it is assumed that the interrupt table resides in 32-bit memory. E enables and D disables protected mode debug, software and hardware breakpoints. HALT and real-time analyzer (RTA) bus breakpoints may still be used with BRkPidt disabled.

However, 'linear_addr' must match the IDT_BASE register contents and 'physical_addr' must correspond to the 'linear_addr' at the time of the breakpoint.

Remarks: The BRkPidt command must be enabled before protected mode breakpoints can be used. If the BRkPidt address matches the IDT_BASE register contents, HALT and bus breakpoints may still be used.

This command is the same as the BRKPIDT command of MICE-V 386.

Examples:

```
->BRP 100N 100P E
->BRP
```

See Also: BRKgdt, BRkRidt, GO, GR, HALt, RBRk, TKB

BRkRidt BRkRidt

Syntax: BRkRidt [address] [BS16] [E]
[D]

Function: Sets the interrupt descriptor table address that the emulator will use for real mode breakpoints.

'address' is the physical base address of the interrupt descriptor table (IDT) when the processor is running in real mode. A physical address must be followed by the letter 'p.' The power-up default is BRkRidt 0p e.

BS16 indicates that the interrupt table resides in 16-bit memory. If BS16 is not specified, it is assumed that the interrupt table resides in 32-bit memory. E enables and D disables real mode debug, software and hardware breakpoints. HALT and real-time analyzer (RTA) bus breakpoints may still be used with BRkRidt disabled, provided 'address' matches the IDT_BASE register contents at the time of the breakpoint.

Remarks: The BRkRidt command must be enabled before real mode debug or software breakpoints are in use. HALT and bus breakpoints may be used whether BRkRidt is enabled or not.

This command is the same as the BRKRIDT command of MICE-V 386.

Examples:

```
->BRR 100P E
->BRR
```

See Also: BRKgdt, BRkPidt, GO, GR, HALt, RBRk, TKB

BYTe BYTe

Syntax: BYTe [address] [= expression [, expression]...]
[= "string"]
[[TO] address [= expression]]
[LENgth n [= expression]]

Function: Displays or alters memory contents in byte scope. The base of two addresses that define an address range must be the same. For example, BYT 200:40 TO 300:300 is invalid.

Examples:

```
->BYT 40          //Display byte content of address DS:40
->BYTE 100:40 TO 100:200
->BYTE &BUF LENGTH 20
->BYTE DS:SI = 23, 234Q, 4+6, AL, 38T
->BYT ARRAY LEN 100 = 0
->BYTE &string = "\tThis is a test program\n"
```

See Also: CHAr, DOUble, DWOrd, FLOat, POInter, QWOrd, TREal, WORd

CALlstack CALlstack

Syntax: CALlstack [expression]

Function: For use with high-level languages. Displays the current chain of procedure calls in the program being executed. A CALLSTACK is a fully qualified list of references to procedures. The reference listed first is the current execution address. The second entry is the current address for the procedure that called the current procedure, etc. CALLSTACK shows the dynamic, run-time nesting of the program as opposed to the static, lexical nesting.

The CALLSTACK display may be incorrect if invoked before a valid stack frame is built.

Example:

```
->CAL
```

See Also: DOWn, Go, TRAce, UP

CAUse

Syntax: CAUse

Function: Reports the cause of the last break in execution.

Remarks: This command is the same as the CAUSE command of MICE-V 386.

Example:

```
->CAUse
```

See Also: B, Go

CHAr

Syntax:

```
CHAr  [address] [= expression [, expression]...]
                [= "string"]
                [[TO] address [= expression]]
                [LENgth n [= expression]]
```

Function: Displays or alters memory contents in byte scope. If the byte value is an ASCII printable character, the character will be displayed. Otherwise, a "." will be displayed for that byte. The base of two addresses that define an address range must be the same. For example, BYT 200:40 TO 300:300 is invalid.

Examples:

```
->CHA 40          //Display ASCII character at address DS:40
->CHAR 100:40 TO 100:200
->CHAR &BUF LENGTH 20
->CHAR DS:SI = 23, 234Q, 4+6, AL, 38T
->CHA ARRAY LEN 100 = 0
->CHAR &string = "\tThis is a test program\n"
```

See Also: BYTe, DOUble, DWOrd, FLOat, POInter, QWOrd, TREal, WORd

CLOse CLOse

Syntax: CLOse [n [, n]...]

where n = 0, 1, 2, 3, 4, or 5.

Function: Closes previously opened file. If no file number n is specified, all the opened files are closed. Files are opened using the OPEN command.

Examples:

```
->CLOSE 1,2         //Closes file 1 and 2.
->CLOSE             //Closes all opened files.
```

See Also: OPEn, REAd, WRIte

CODe

Syntax:

```
CODe [ON ]
     [OFF]
```

Function: Enables or disables hex code display during code disassembly for both the source window and the command line. Disabling hex code display allows other windows to overlap the right side of the source window without completely obscuring useful disassembly information.

Examples:

```
->COD               //Displays current CODE setting
->COD ON            //Turns code display on
->COD OFF           //Turns code display off
```

See Also: DASm, NUMber, SOUrce, VIEw

COMpare

Syntax:

```
COMpare start_addr1  {[TO] end_addr1}  [NOT] start_addr2  [BYTe]
                     {LENgth n      }                     [WORd]
                                                          [DWOrd]
```

Function: Compares two regions of memory and reports all differences.

NOT displays the addresses that match, instead of the normal display of mismatches.

BYTe, WORd or DWOrd specifies the size of the memory reads used to gather data for the comparison. The power-up default is BYTe.

Remark: The syntax of this command is similar to that of the CMP command of hyperICE-386.

Examples:

```
->COM &byte1 LEN 1 &char1     //Compare char1 to byte1
->com 30:50 to 30:300 200:200 //Compare memory from 30:50 and
->                            //30:300 inclusive to that at
->                            //200:200 to 200:450.
->COM ARRAY_BUF len 50 NOT 40:50 WORD
->COM &STRING1 LEN 20 MEM2
```

See Also: COPy, FINd

CONtinue CONtinue

Syntax: CONtinue

Function: Causes the next iteration of the REPEAT-UNTIL, WHILE-EWHILE, FOR-EFOR loop to begin. In the REPEAT-UNTIL and WHILE-EWHILE loop, the test part is executed immediately. In the FOR-EFOR loop, control passes to the re-initialization step.

Examples:

```
->MACRO TEST1       //Define a macro.
MD>while( a )
MD>   b = subr( &c, d );
MD>   if( b == a )
MD>      continue;
MD>   else
MD>      c = c << 3;
MD>      d = d % 3;
MD>   eif
MD>ewhile
MD>EMACRO
->WHILE I--
CD>   IF A != SUB( I, D )
CD>      CONTINUE
CD>   ORIF
CD>      D = A
CD>   EIF
CD>EWH
```

See Also: BREak, FOR, GOTo, IF, REPeat, SWItch, WHIle

COPy COPy

Syntax:

```
COPy src_start_addr {[TO] src_end_addr}  dest_addr   [BYTe]
                    {LENgth n         }              [WORd]
                                                     [DWOrd]
```

Function: Copies a block of memory contents from one location to another.

BYTe, WORd, or DWOrd indicates how the data are read from and written to memory.

Remarks: The syntax of this command is similar to that of the MOVE command of hyperICE-386.

Examples:

```
->COPY &char1 LEN 1 &byte1    //Copy one byte from char1 to byte1
->COPY 30:50 TO 30:300 200:200      //Copy from memory 30:50 and
->                            //30:300 inclusive to destination
->                            //starting at 200:200.
->COPY 40:50 len 50 ARRAY_BUF WORD
->COPY &STRING1 LEN 20 MEM2
```

See Also: COMpare, FINd

CRRepeat CRRepeat

Syntax:
```
CRRepeat [ON ]
         [OFF]
```

Function: Enables, disables, or displays the status of command repeating on C/R. If enabled, the Enter key will repeat the last execution, disassembly, or memory display command from the point where the last command left off. Otherwise, Enter does nothing.

Example:
```
->CRR ON
```

See Also: ENV

CW CW

Syntax: CW

Function: Displays or changes the value of the 80387 control word. The value of the control word is displayed followed by a slash. The contents can be altered by entering a new hexadecimal value. A carriage return alone will preserve the contents.

Examples:

```
->CW
  CONTROL WORD = 0000H / 1324H
->CW
  CONTROL WORD = 1324H / <CR>
```

See Also: ST, SW, TW

DASm or U

Syntax: {DASm} [address1 [[TO] address2]] [MIX]

{U} [LENgth n]

Function: Displays a block of memory in assembly mnemonic form. The MIX qualifier causes source to be mixed in with the disassembly display.

Remark: This command is the same as the U command.

Examples:

```
->DASM               //Default address is CS:IP
->DASM CS:(IP+5) MIX
->DASM &MAIN LEN 20
```

See Also: SOUrce, VIEw

DECimal DECimal

Syntax: DECimal

Function: Sets the default input radix to decimal or base 10.

Example: `->DEC`

See Also: BINary, HEX, OCTal, RADix

DIRectory MACro

Syntax: DIRectory MACro

Function: Lists the names of all defined macros.

Example:

```
->DIR MAC
```

See Also: :, DISplay MACro, EDit MACro, INClude, MACro, MLIst, PUT, REMove MACro

DIRectory MODule

Syntax: DIRectory MODule

Function: Lists all module names and corresponding source file names.

Example:

```
->DIR MODULE
```

See Also: $, SET, SOUrce, SYMbol

DIRectory STRucture DIRectory STRucture

Syntax: DIRectory STRucture

Function: Lists all structure names which have been defined or loaded.

Example:

```
->DIR STR
```

See Also: DISplay STRucture, STRucture

DISplay MACro DISplay MACro

Syntax: DISplay MACro [macro_name [, macro_name]...]

Function: Displays all or some macro definitions.

Examples:

```
->DIS MAC                //Displays all macro definitions.
->DIS MAC AA, BB, CC     //Displays macros AA, BB, CC.
```

See Also: :, DIRectory MACro, EDit MACro, INClude, MACro, MLIst, PUT, REMove MACro

DISplay STRucture

Syntax: DISplay STRucture [structure_name [, structure_name]...]

Function: Lists the contents of the specified structures. If no structure name is specified, all structure definitions in the structure directory are listed.

Examples:

```
->DISPLAY STRUCTURE      //Lists all structure definitions.
->DIS STR STR1, STR2     //Lists STR1 and STR2 structure
->                       //definitions.
```

See Also: DIRectory STRucture, STRucture

DISplay TRAce or PRInt

DISplay TRAce or PRInt

Syntax: DISplay TRAce [start_line [end_line]] [CLEar]

Function: Displays the trace buffer. 'start_line' is the line number where the trace display begins. 'end_line' is the line number to end the display. CLEar clears the entire trace buffer (The CLEar key word can also be specified as CLR).

Remarks: This command is the same as the PRInt command.

The syntax of this command is similar to that of the DT command of MICE-V.

Examples:

```
->DIS TRA 0t 20t  //Prints bus cycle trace frames 0 to 20.
->DIS TRA CLE     //Clears trace buffer.
```

See Also: HTRc, PRInt

DOUble DOUble

Syntax: DOUble [address] [= expression [, expression]...]
[[TO] address [= expression]]
[LENgth n [= expression]]

Function: Displays or alters memory contents in double (8-byte) scope. The base of two addresses that define an address range must be the same. For example, DOUBLE 200:40 to 300:300 is invalid.

Examples:

```
->DOUBLE 40
->DOUBLE 100:40 TO 100:200
->DOUBLE &double_buf LENGTH 20
->DOUBLE DS:SI = 9.9, 8.8, 1.2+3.5
->DOUBLE double_array LEN 100 = 0.0
```

See Also: BYTe, CHAr, DWOrd, FLOat, POInter, QWOrd, TREal, WORd

DOWn DOWn

Syntax: DOWn [n]
[HOMe]

Function: Walks down the call stack allowing access to the source and local variables of any active procedure. If no argument is specified, the stack is walked down one level. If HOMe is specified, the active scope returns to what it was before any UP or DOWN command was issued.

If any execution command or command that directly changes the CS:IP or BP is given by the user while an UP or DOWN command is in effect, a DOWN HOME action is automatically performed before the command is executed.

Examples:

```
->DOWN              //Walk down one level
->DOWN 3            //Walk down three levels
->DOWN HOME         //Return to the initial scope
```

See Also: CALlstack, SOUrce, SYMbol, UP

DT

Syntax: DT \[selector_expr\] [.seg_element [= expr]]

where selector_expr is an expression that computes to a 16-bit number or the name of a 16-bit register (such as TR, LDTR, CS, DS, etc.) which is interpreted as a selector; seg_element is BASe, LIMit, P, DPL, C, R, W, A, E, WCO, SOFf, SSEl, or TYPe; expr is an expression. The notations \[and \] indicate that the square bracket pair is part of the required syntax.

Displays or modifies descriptors, descriptor components or descriptor tables.

The descriptor components are as follows:

Name	Size	Description
BASe	24 bits	Segment base linear address
LIMit	16 bits	Segment limit
P	1 bit	Segment present, 1 = present
DPL	2 bits	Descriptor privilege level
C	1 bit	Conforming segment
R	1 bit	Readable segment
W	1 bit	Writable segment
A	1 bit	Segment accessed
E	1 bit	Expand down segment
WCO	5 bits	Gate word count
SOFf	16 bits	Gate segment offset
SSEl	16 bits	Gate segment selector
TYPe	5 bits	Descriptor type

Examples:

```
->DT[OCH]           //Displays the descriptor referenced by a
->                  //selector value of OCH.
->DT[cs]            //Displays the descriptor referenced by cs
->                  //register.
->DT[cs].base       //Displays the base value of segment referenced
->                  //by cs register.
->DT[ldtr]          //Displays the descriptor referenced by ldtr
->                  //register.
->DT[tr]            //Displays the task state segment (TSS)
->                  //referenced by the TR register.
```

See Also: GDT, IDT, LDT, REGister, TSS

DWOrd

Syntax: DWOrd [address] [= expression [, expression]...]
[[TO] address [= expression]]
[LENgth n [= expression]]

Function: Displays or alters memory contents in double word (4-byte) scope. The base of two addresses that define an address range must be the same. For example, DWORD 200:40 to 300:300 is invalid.

Examples:

```
->DWORD 40          //Display double word at address DS:40
->DWORD 100:40 TO 100:200
->DWORD &unsigned_long_buf LENGTH 20
->DWORD DS:SI = 23, 234Q, 4+6, AL, 38T
->DWORD unsigned_long_array LEN 100 = 0
```

See Also: BYTe, CHAr, DOUble, FLOat, POInter, QWOrd, TREal, WORd

EDit EDit

Syntax: EDit filename

Function: Invokes the editor defined in hyperSOURCE-386 EDITOR environment variable to edit the specified file. It should be noted that editing a source file of the program being debugged can cause source line number information to be rendered incorrect.

Examples:

```
->ED foo.c
```

See Also: SOUrce

EDit MACro

EDit MACro

Syntax: EDit MACro name

Function: Invokes the editor defined in hyperSOURCE-386 EDITOR environment variable for the creation or editing of a macro.

Example:

```
->ED MAC FOO
```

See Also: :, DIRectory MACro, DISplay MACro, ENV, INClude, MACro, MLIst, PUT, REMove MACro

EGA

Syntax: EGA [ON]
[OFF]

Function: Enable or display EGA-43 line or VGA-50 line display mode.

Examples:

```
->EGA              //Displays the current display mode.
->EGA ON           //Display EGA-43 or VGA-50 line display mode.
->EGA OFF          //Displays 25 lines.
```

See Also: ENV

ENV

Syntax: ENV [filename]

Function: ENV displays parameters of the configurable environment, and optionally allows the environment to be saved to a file. Environment parameters displayed include:

BARLINE = n	Position of the horizontal bar dividing the source and dialog windows with respect to the top of the display screen. Default is 18.
BAUD = n	Baud rate of serial port connecting to MICE-V 386; n = 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, or 57600.
BEEP bool	Audible error beep setting.
BREAKWN=win	Break window size, position, and status.
CODE bool	Source-window object code display setting.
COLOR = file	Color values file for windows.
COM = n	Serial port number; 1 for COM1 or 2 for COM2.
CRREPEAT bool	Command repeat on C/R setting.
CSTACKWN=win	Call-stack window size, position and status.
DIALOG = n	Maximum depth of the dialog window; n is from 43 to 512 lines. Default is 80 lines.
EDITOR = file	Editor program to be invoked for file editing.
EGA bool	EGA-43/VGA-50 line mode setting.
EXTENSION=ext	Source lines counting method (C or LiST).
HISTORY = n	Maximum depth of the history window; n is from 1 to 512 lines. Default is 40 lines.
HOME name	Home window, name is SOUrce/COMmand.
<key>=string	Key macro definitions.
MAIN bool	Start-debugging-at-function-main setting.
MEMWN = win	Memory window size, position, and status.
NUMBER bool	Source-window line-number setting.
PROLOG = bool	Automatic prolog execution setting.
RADIX base	Input number base setting; base is HEX, DEC, OCT, or BIN.
REGWN = win	Register window size, position and status; win is described below.
RTRACEWN=win	Run-trace window size, position, and status.
SENSITIVE bool	Case-sensitive symbol matching setting.
SPATH path	Source file search path, multiple paths are separated by semicolons.

STARF = file	Start-up command filename.
TAB = n	Tab expansion size; n is from 1 to 8. Default is 4 spaces.

The "bool" argument is either ON or OFF.

The "file" argument is a file name.

The "string" argument is an ASCII string. Some key macro examples are as follows:

```
<F5> = pline
<ALT-F1> = time
<SHF-F2> = version
```

The "path" is a directory path name. Multiple paths are separated by semicolons. For example:

```
SPATH = c:\hs386\demo;d:\project
```

The "win" argument is an argument list as follows:

```
    y x h w s
```

where:
y is the y-coordinate (vertical-axis) with respect to the upper left hand corner of the screen. Its range is from 0 to 24.

x is the x-coordinate (horizontal-axis) with respect to the upper left hand corner of the screen. Its range is from 0 to 79.

h is the height of the window. Its range is from 1 to 24.

w is the width of the window. Its range is from 1 to 80.

s is the status of the window. It is either OPEN or CLOSE.

The start-up environment file's name can be set in the HS386ENV MS-DOS environment variable. The default name is hs386.env. In the environment file, the # prefix specifies a comment. If an environment file is not found, hyperSOURCE-386 will use default settings. You can find out the default settings by writing them to an output file.

Examples:

```
->ENV
->ENV HS386.ENV
```

See Also: BEEp, CODe, EDit, EGA, EXTension, HOMe, NUMber, RADix, SENsitive, SPAth

ESCape ESCape

Syntax: ESCape

Function: The ESCape command stops macro processing. It is used in macro definitions to abort processing.

Example:

```
->IF AX > 0
CD>ESCAPE          //Returns to command mode.
CD>ELSE
CD>AX++
CD>EIF
->
```

See Also: BREak, CONtinue, GOTo

EVALuate or = EVAluate or =

Syntax: {EVAluate} expr

{=}

Function: Evaluates an expression. All register and flag names are recognized. Full C syntax is supported. Moreover, the following key words are recognized:

NOT, AND, OR, XOR, BYTE, CHAR, WORD, SHORT, DWORD, LONG, FLOAT, DOUBLE, TREAL, SIZEOF.

Note that the QWORD key word is not allowed.

You can also evaluate a function in your program that returns a value. If a symbol name has the same name as the register or flag names, it should be prefixed with the # operator in the expression.

The following key words are allowed in the SIZEOF operator and type casting:

BYTE, CHAR, WORD, SHORT, DWORD, LONG, FLOAT, DOUBLE, TREAL.

Examples:

```
->eva 'A'
->= ax + 1
->eva ++i               //value of i is incremented by 1
->eva i + 1             //value of i is unchanged
->eva *(byte *)ptr_to_short + *val
->= fact(3) + 6
->eva  a * b >> 2
->eva sizeof(count)     //a symbol
->eva sizeof(long)      //a type
->eva count and 1       //logical and
->eva short(i)
->= i or j              //logical or
->eva ptr->size
->eva ary[1].length + count
->=  (double)realno + c
->=  ax | 148fh
->eva #ax               //ax is a variable in program module.
```

See Also: ?, SYMbol

EXAmine or E

Syntax: {EXAmine} lvalue

{E}

Function: Opens a symbol window to display the value and type information of the specified lvalue.

If a symbol window has already been opened for the specified lvalue, that symbol window will be selected.

Examples:

```
->EXA TOP          //Examine a symbol called top.
->E COUNT
->E *(char *)0
```

See Also: ?, =, SYMbol

EXIt

Syntax: EXIt

Function: Exits from the debug session. This command closes all opened files and deletes all temporary files that are created by hyperSOURCE-386. You can also press the <Alt>x keys to terminate the debug session.

Remark: This command is the same as the QUIt command.

Example:

```
->EXIT
```

See Also: QUIt

EXTension EXTension

Syntax: EXTension [C | LiST]

Function: Displays or sets the method for counting line numbers in source files. The default parameter is 'C.'

When the 'C' parameter is in effect, the source files are assumed to have C as file extension. The line numbers in these source files are assumed to be continuous without any break, starting from 1 for the first line to the last line, even though the source files may have other include files. You should use this file extension for C compilers from most vendors except Intel.

When the 'LST' parameter is in effect, the source files are assumed to have LST as file extension. The line numbers in these source files are not continuous, with breaks to accommodate any include files. You should use this parameter for Intel compilers - C, PL/M, PASCAL, etc.

If no argument is specified, this command displays the current setting.

The file extension can also be set using the EXTENSION parameter in hyperSOURCE-386's environment file. Both the command and the environment parameter have the same syntax.

Note that both the EXTENSION command and the EXTENSION parameter in hyperSOURCE-386's environment file affect how the line numbers are counted in the source files. If you want to override the default file extension you have to specify the actual file extension of your source files with the LOAd command's EXTension="string" argument.

Examples:

```
->EXT LST          //Object file is generated by Intel compiler.
->LOAD EXT="PAS"   //File extension of source listing is PAS.
```

See Also: ENV, LOAd

FINd FINd

Syntax:

```
{FINd} addr1    {[TO] addr2}  [NOT]    {"string"}
                {LENgth n  }           {expr [expr]...}  [BYTe]
                                                         [WORd]
                                                         [DWOrd]
```

Function: Searches the specified memory range for the specified value or string of values.

NOT indicates to display all locations that do not match the value. NOT is the exact inverse of the normal operation. If a pattern is matched, the address is not displayed. If a pattern is not matched, the address is displayed.

'expr' is one or more data values to be searched (or NOT searched) for. BYTe, WORd or DWOrd indicates how the data will be read. The power-up default is BYTe.

Remarks: The syntax of this command is similar to that of the FIND command of MICE-V 386.

Examples:

```
->FIN &buf[12] LEN 8192 "Copyright"
->FIN DS:100H TO DS:0FFFFH 50H 4CH
->FIN DS:40H LEN 20 0x56 WOR  //Search for 0056h
```

See Also: COMpare, COPy

FLAg

Syntax: {FLAg}
{status_flag [,status_flag]...}
{status_flag = value [, status_flag = value]...}

where status_flag may be:
AF, CF, DF, IF, IOPL, NT, OF, PF, RF, SF, TF, VM, or ZF.

Function: Displays or alters the 80386 flags register. When changing flag value, any value at the right hand side of "=" which is not 0 is treated as 1. Note that IOPL is two bits; the rest of the flags are one bit.

Remark: You can display the status flag register with the register name FS.

Examples:

```
->FLA                          //Display all flags
->CF, OF, ZF                   //Display CF, OF, ZF
->CF = 1, IF = 1, AF = 1       //Set flag value
->FS                           //Display the status flag register.
```

See Also: REGister

FLOat

Syntax:

```
FLOat [address]  [= expression [, expression]...]
                 [[TO] address [= expression   ]]
                 [LENgth n   [= expression     ]]
```

Function: Displays or alters memory contents in float (4-byte) scope. The base of two addresses that define an address range must be the same. For example, FLOAT 200:40 TO 300:300 is invalid.

Examples:

```
->FLOAT 40
->FLOAT 100:40 TO 100:200
->FLOAT &float_buf LENGTH 20
->FLOAT DS:SI = 9.9, 8.8, 1.2+3.5
->FLOAT float_array LEN 100 = 0.0
```

See Also: BYTe, CHAr, DOUble, DWOrd, POInter, QWOrd, TREal, WORd

FOR

Syntax:

FOR ([expr1a[,expr1b]...];[expr2];[expr3a[,expr3b]...])
[command1[,command2]...]
EFOr

The FOR-EFOr loop command is equivalent to:
[expr1a]
[expr1b]
:
WHILE [expr2]
[command1]
[command2]
:
[expr3a]
[expr3b]
:
EWHILE

Note: If expr2 is not specified, it is taken as permanently TRUE.

Function: In the FOR-EFOR loop command, **expr1** is the initialization expression, **expr2** is the loop control expression and **expr3** is the re-initialization expression. There may be multiple initialization and re-initialization expressions, but only one conditional expression.

Examples:

```
->MACRO TEST1      //Define a macro.
MD>for( a=0,b=0; a<=0; a++ )
MD>   c=abc( &b, *ptr )
MD>   c = c >> 4
MD>   d |= b
MD>efor
MD>emacro
->macro test2      //Define a macro.
MD>for(;;)
MD>   if( !( c = get_data() ) )
MD>       break;
MD>   else
MD>       buf[i++] = c;
MD>   eif
MD>efor
MD>emacro
->for (i = 0; I < 10; i++)
CD>   buf[i] = 0;
CD>efor
->
```

See Also: BREak, CONtinue, GOTo, IF, REPeat, SWItch, WHIle

FREe FREe

Syntax: FREe

Function: Displays the size of remaining free DOS memory on the PC host. You can determine whether there is enough DOS memory to spawn a new command shell to run another application.

Examples:

```
->FRE
```

See Also: !, LOAd

GDT

Syntax: GDT [\[expr\] [.LDT\[expr\]] [= expr]
[.seg_element]

where seg_element is BASe, LIMit, G, B, P, D, DPL, C, R, W, A, E, V, WCO, SOFf, SSEl, or TYPe.

Function: Displays or modifies descriptors or descriptor components of the global descriptor table.

Examples:

```
->GDT                      //Displays the GDT.
->GDT[1T].P=1              //Writes the present bit of GDT(1).
->GDT[10]=GDT[5].LDT[7]    //Makes one table entry the same as
->                         //another.
->GDT[7T].LDT              //Displays the LDT whose descriptor is the
->                         //seventh entry in the GDT.
->gdt[7].ldt[4].limit=12345h  //Writes the limit field of entry
->                            //four in the LDT whose descriptor
->                            //is entry seven in the GDT.
```

See Also: DT, IDT, LDT, PD, REGister, TSS

GLObal GLObal

Syntax: GLObal

Function: Displays all global symbols. The value of any global variable may be examined via the EXAmine command.

Example:

```
->GLOBAL
```

See Also: ?, =, EVAluate, EXAmine, LOCal, SYMbol

Go

Syntax:

```
Go [FROm addr]  [FORever]
                [[TILl] address]
                [[TILl] RETurn [level]]
                [[TILl] CALl]
```

Function: Starts emulation. If no argument is specified, program execution begins from the current program counter. Program execution will be halted if a breakpoint is reached.

The "FROm addr" parameter is used to specify the starting address for program execution. The address must be a virtual address or an offset into the current code segment. It cannot be a physical nor a linear address.

If the argument is FORever, all previously specified breakpoints (see the B command) are disabled, and the emulator runs forever (or until the program is killed). You can break emulation by pressing <Esc> or <Ctrl>c and then entering the HALt command.

If RETurn is specified, execution terminates after the code has returned from the specified number of levels of call nesting. The default is to return from the current procedure (i.e., 1 level).

If CALl is specified, execution continues until a CALL or INT instruction is executed.

Remarks: Before using GO, you must first set up the stack pointer as follows:

For virtual mode addressing: ESP > = 24h

For protected mode privilege level zero: ESP > = 0Ch

For protected mode privilege level non-zero: ESP > = 14h

GO enters emulation with a long jump instruction.

Examples:

```
->GO FROM &START FOREVER      //Go from address of start with
->                            //breakpoints disabled
->GO TIL #40        //Go until ready to execute line #40
->GO RET            //Go until first RET instruction
->GO CALL           //Go until next CALL instruction
```

See Also: B, CALlstack, GR, Step, ISTep, LOAd

GOTo GOTo

Syntax: GOTo macro_label

Function: Causes program execution to be transferred to the specified macro-label. Note that GOTO command cannot be used to jump into REPEAT-UNTIL, WHILE-EWHILE, FOR-EFOR loop, or IF-EIF, SWITCH-ESWITCH block, but can be used to jump out of these loops or blocks.

Examples:

```
->MACRO TEST1       //Define a macro.
MD>a = 8;
MD>aa:
MD>if( c >= 999 )
MD>   goto aa
MD>orif( c <= 0 )
MD>   goto bb
MD>eif
MD>bb:
MD>EMACRO
->
```

See Also: BREak, CONtinue, ESCape, MACro

GR GR

Syntax: GR [FORever]
[[TILl] address]

Function: Starts emulation with a "go reset." Program execution will be halted if a breakpoint is reached.

If the argument is FORever, all previously specified breakpoints (see B) are disabled, and the CPU executes forever (or until the program is killed).

Remarks: Before using GR, you must first set up the stack pointer as follows:

For virtual mode addressing: ESP > = 24h

For protected mode privilege level zero: ESP > = 0Ch

For protected mode privilege level non-zero: ESP > = 14h

GR enters emulation with a short jump instruction. GR is necessary if your target system's boot-up code is physically located between FFFF0000-FFFFFFFF at reset.

Examples:

```
->GR FOREVER
->GR TIL 300H:45H
```

See Also: B, CALlstack, GO, IStep, LOAd, STEp

HALt

Syntax: HALt

Function: Halts emulation.

After you have entered the Go command, you can halt emulation by first pressing <Esc> or <Ctrl>c and then entering the HALt command.

Remark: This command is the same as the HAlt command of MICE-V 386.

Example:

```
->HALt
```

See Also: Go

HELp

Syntax: HELp [["]command_keyword["]]

Function: Shows you how to use hyperSOURCE-386 commands. If no parameter is specified, the command summary will be displayed. If a command key word is given, the syntax and example of usage of the command will be displayed.

Examples:

```
->HELP              //Enters the HELP menu system.
->HELP EVALUATE     //Displays help on the EVALUATE command.
->HELP "BYTE"
```

HEX

Syntax: HEX

Function: Sets the default input radix to hexadecimal or base 16.

Examples:

```
->HEX
```

See Also: BINary, DECimal, OCTal, RADix

HOLdtp HOLdtp

Syntax: HOLdtp [[=] ON]
[[=] OFF]

Function: Places the 80386 target processor into a HOLD state while plugging into a power-up target.

With HOLdtp set to ON, functions requiring the target processor cannot be performed. With HOLdtp set to OFF, the emulator operates normally. The power-up default is OFF.

Remarks: HOLdtp places the processor pins into high impedance which allows the probe to be plugged into the target while power is on. If you do not issue the HOLdtp command before plugging into a target when the emulator is powered up, you can cause damage to the emulator, the target, or both.

This command is the same as the HOLDTP command of MICE-V 386.

Examples:

```
->HOL ON
->HOL OFF
```

HOMe

Syntax: HOMe [COMmand]
[SOUrce]

Function: Sets the default window from which commands are issued and to which commands return after execution. COMmand specifies the command-line, and SOUrce specifies the source window as the home base. The command-line is the default home base, but the default may be changed in the environment file.

Examples:

```
->HOM SOU          //Issue commands from the source window
->HOM COM          //Issue commands from the command-line
```

See Also: ENV, SOUrce

HTRc HTRc

Syntax: HTRc

Function: Starts trace collection while emulating.

Example:

```
->HTRC
```

IDT IDT

Syntax: IDT [\[expr\]][.seg_element] [= expr]

where seg_element is BASe, LIMit, G, B, P, D, DPL, C, R, W, A, E, V, WCO, SOFf, SSEl, or TYPe.

Function: Displays or modifies descriptors or descriptor components of the interrupt descriptor table.

Examples:

```
->IDT               //Displays the IDT.
->IDT[1T]           //Displays an entry in the IDT.
```

See Also: DT, GDT, LDT, PD, REGister, TSS

IF IF

Syntax:

IF expression
[command]
:
[ORIf expression]
[command]
:
[ELSe]
[command]
:
EIF

Function: If any expression is TRUE (non-zero), the commands associated with it are executed. If all of the expressions are FALSE (zero), either no action at all or the commands associated with ELSE are executed, This command can be formed by an IF-EIF, IF-ELSE-EIF, IF-ORIF-EIF, or IF-ORIF-ELSE-EIF clause.

Examples:

```
->MACRO TESTIF    //Define a macro.
MD>IF A + B >= %0
MD>LINE
MD>ORIF A + B < %1
MD>GO TIL %2
MD>ELSE
MD>A
MD>B
MD>EIF
MD>EMACRO
->MACRO TTT
MD>if( a > 0 )
MD>    if( b++ != 0 )
MD>     c--
MD>     d--
MD>    orif( b < -3 )
MD>     c >>= 2;
MD>    eif
MD>else
MD>    a = 0
MD>eif
MD>EMACRO
->
```

See Also: EIF, FOR, INClude, MACro, REPeat, SWItch, WHIle

INClude or @ INClude or @

Syntax: {INClude} ["]file_name["] [LISt]

{@}

Function: Executes the commands from the specified file. If LISt is specified, the commands in the command file are displayed on the console as they are being executed (default is NO LIST).

Remark: This command is identical to the @ command.

Examples:

```
->INC "INIT.MAC" LIS
->@ C:\TMP\GR1.INC
```

See Also: JOUrnal, LISt, MACro

INPut

Syntax: INPut [address =] port_no [W]
[D]

Function: Reads the contents from the specified input port and displays it on the console. The contents are the data involved in the last I/O operation performed via the port. If the qualifier "W" is specified, the port is 16-bit. Otherwise, it is 8-bit. If the qualifier "D" is specified, the port is 32-bit. If an "address" is specified, the contents of the input port will be stored at the specified address.

Examples:

```
->INPUT 20H
->INP port_index W
```

See Also: OUTput

IStep IStep

Syntax: IStep [INto] [n]

Function: IStep causes the program to execute n machine instructions before breaking. If n is not specified, the default is 1, which allows single-step debugging.

IStep INto will step into the called procedure.

Examples:

```
->IS 10             //Execute 10 instructions and stop.
->IS                //Execute one instruction and stop.
->IS IN             //Step into the called procedure.
```

See Also: Go, Step

JOUrnal, NO JOUrnal

Syntax: JOUrnal ["]file_name["] [KEYboard] [APPend]

NO JOUrnal

Function: Creates a text file with the specified file name and records the user's entered commands into the specified file. The command file created may be used in the INClude command.

If KEYboard is specified, the journal file will store all entered keystrokes, including those used in windows, rather than only the commands entered on the command line.

If the specified file already exists, and the APPend qualifier is specified, then the existing file will be appended rather than over-written.

This command may be disabled with the NO JOUrnal command.

Examples:

```
->JOURNAL "MYDEBUG.LOG" //Record all entered commands to a file
->                      //named "MYDEBUG.LOG."
```

See Also: INClude, LISt

LDT LDT

Syntax: LDT [\[expr\]][.seg_element] [= expr]

where seg_element is BASe, LIMit, G, B, P, D, DPL, C, R, W, A, E, V, WCO, SOFf, SSEl, or TYPe.

Function: Displays or modifies descriptors or descriptor components of the local descriptor tables.

Examples:

```
->LDT                          //Displays the LDT.
->LDT[12H]                     //Displays an entry in the LDT.
->LDT[2T].BASE=12345678H       //Writes base field of LDT(2).
```

See Also: DT, GDT, IDT, PD, REGister, TSS

LiNEar

Syntax: LiNEar address

Function: Converts the address to a linear address. Note that the abbreviation for this command is LNE and not LIN, in order to distinguish it from the LINE command.

Examples:

```
->LNE 0D420000P    //Converts a physical address to a linear
->                 //address.
->LNE 1444:5678    //Converts a real mode virtual address.
->LNE 17H:12H:0    //Converts a protected mode virtual address.
```

See Also: PHYsical, VIRtual

LISt, NO LISt

Syntax: LISt ["]file_name["] [APPend]

NO LISt

Function: Creates a text file with the specified file name and records the console display into the specified file. This command enables you to make a copy of a debugging session.

If the APPend qualifier is given and the specified file already exists, then the logged data will be appended to the specified file.

Remark: This command is disabled with the NO LIST command which closes the list file.

Examples:

```
->LIST  "MYDEBUG.LOG"    //Record current session.
->NO LIS                 //Terminate logging.
```

See Also: JOUrnal

LOAd LOAd

Syntax: LOAd ["]file_name["] [NOCode] [EXTension="string"]

Function: Loads an object file from the host into the target memory. The file must be an absolute file in Intel OMF86, OMF286, or OMF386 formats.

If NOCode is specified, executable code is not loaded into memory; only symbol, line number and type information is loaded into hyperSOURCE-386.

The default file extension of source files is ".c"; EXTension can be used to specify an alternate source file extension. The EXT entry in the *.env file will do the same function.

Remark: The symbol and type information is loaded into the extended memory (RAM above 1M bytes) of the PC host. Each symbol requires 40 bytes of RAM as overhead plus the length of the symbol. Each type description requires five bytes of RAM overhead plus the length of the type description.

Examples:

```
->LOAD "MYPROG.OMF"
->LOA MYPROG NOCODE
->LOAD C:\PROG\TSTPRO EXT = "pas"
```

See Also: EXTension, FREe, Go

LOCal LOCal

Syntax: LOCal

Function: Displays all active local symbols - type and address only. You need to use the = or EVAluate command to examine their values.

Example:

```
->LOC
```

See Also: EVAluate, GLObal, SYMbol

MACro

Syntax:

```
MACro  macro_name
[command]
  :
EMAcro
```

Function: Defines a macro body. The text lines enclosed between the **MACro** definition command and the **EMAcro** command will be stored in the macro symbol table. The text lines can be any commands or comment lines. The command can even be a macro invocation (see below) command. However, it cannot be another macro definition command. In other words, nested macro definition is not supported.

The macro can be invoked by prefixing the ':' before the name of the specific macro. As part of the macro invocation, up to 10 parameters can be passed to the macro. The macro parameters can be specified within the macro definitions as %0 to %9. Macro parameters are passed as text strings.

Examples:

```
->MAC AA            //Define macro AA.
MD>$stub#m=%0       //Pass one parameter.
MD>emacro
->
```

See Also: :, DIRectory MACro, DISplay MACro, EDit MACro, INClude, MLIst, PUT, REMove MACro

MAP MAP

Syntax:

```
MAP  [start_addr   [[TO] end_addr] ]   [FAST]   [RAM]
                   [LENgth n]          [TARG]   [ROM]
     [CLEar]                                    [NONE]
     [E]
     [D]
```

Function: Displays or sets up the user system memory layout.

CLEar clears any existing memory map by setting all memory to TARG RAM (The CLEar key word can also be specified as CLR). E enables the map. E can be used to restore the memory map. D disables the map. start_addr is the starting address of the map. The range must be in multiples of 64K bytes. If it is not, then it is rounded to the beginning of the 64K-byte boundary block. end_addr is the ending physical address of the map. n is the number of bytes in the memory range. If end_addr or n is not on a 64K-byte boundary, it is rounded up to the nearest boundary. If end_addr or n is not given, one 64K-byte block is mapped. FAST indicates to use overlay RAM located on the in-circuit probe. TARG indicates that the memory range exists in TARGet (user) memory. RAM, ROM or NONE indicates the valid read/write access to be checked whenever an address in the memory range appears on the processor bus. RAM indicates the memory is both read and write. ROM indicates the memory is read-only. NONE indicates no memory exists for the specified range (no read or write).

Examples:

```
->MAP
->MAP OP OFFFFP FAST RAM
->MAP D
```

See Also: RAMtst, RAmtstP

MEMory

Syntax: MEMory [address]

Function: Enters the memory window. If no address is specified, the last specified address will be used.

Examples:

```
->MEM
->MEM ds:100h
->MEM big_array
```

See Also: BYTe, CHAr, DOUble, DWOrd, FLOat, POInter, QWOrd, TREal, WORd

MLIst MLIst

Syntax: MLIst [ON]
[OFF]

Function: Causes the macro bodies to be displayed on the console as the macros are expanded. The MLIST OFF command may be used to disable this command. On start up the default is MLIST OFF.

Example:

```
->MLI               //Display current setting.
->MLI ON            //Enable the display of macro body.
->MLI OFF           //Disable the display of macro body.
```

See Also: INClude, LISt, MACro

NUMber   NUMber

Syntax: NUMber [ON]
[OFF]

Function: Enables or disables the displaying of line numbers in the source window. If no argument is specified, the current setting will be displayed. The default is OFF.

Examples:

```
->NUM ON
->NUM OFF
```

See Also: #, B, Go, VIEw

OCTal

Syntax: OCTal

Function: Sets the default input radix to octal or base 8.

Example:

```
->OCT
```

See Also: BINary, DECimal, HEX, RADix

OPEn OPEn

Syntax: OPEn n = ["]file_name["]

where n = 0, 1, 2, 3, 4, or 5

Function: Opens a file and associates it with a number n. The opened file may later be used in READ or WRITE commands to read/write data from/to the file. If the specified file does not exist, it will be created.

Examples:

```
->OPEN 1 = INPUT.DAT    //Open file INPUT.DAT
->OPEN 2 = "OUTPUT.DAT" //Open file OUTPUT.DAT
->READ A, B FROM 1
->WRITE "A = ", A TO 2
```

See Also: CLOse, INClude, MACro, REAd, WRIte

OUTput OUTput

Syntax: OUTput port_no = value [W]
[D]

Function: Assigns an 8-bit value to the specified output port. If the qualifier "W" is specified, a 16-bit value will be assigned. If the qualifier "D" is specified, a 32-bit value will be assigned.

Examples:

```
->OUTPUT 20H = 0C2H
->OUT out_port_index = 1011H W
->OUT out_port_index + 6 = 10H
```

See Also: INPut

PAUse PAUse

Syntax:

```
PAUse   [ON]
        [OFF]
```

Function: Enables or disables the scrolling of the display in the dialog window. When PAUse ON is in effect, the display will stop if the display lines fill up the dialog window. The display will continue to scroll after any key is pressed. PAUse OFF turns off the effect such that the display will continue scrolling without any pause.

The default is PAUse ON. PAUse OFF is needed if the debugging session must be run unattended using a command file. PAUse without any parameter will display the current setting.

Examples:

```
->PAUse           //Displays current setting.
->PAU OFF         //No pause during text scrolling.
```

See Also: INClude, LISt, WAIt

PD PD

Syntax: PD [\[expr\]][[.PT[\[expr\]].pt_part [= expr]]

where pt_part is PTA, PFA, AVL, P, RW, US, D, or A.

Function: Displays page directory. Displays or modifies page table and page table entry.

Examples:

```
->PD[DT]             //Displays the page table at page directory
->                   //index zero.
->PD[OT].PT[4T]      //Displays the 5th entry in the first page
->                   //table.
```

See Also: DT, GDT, IDT, LDT, REGister, TSS

PHYsical

Syntax: PHYsical address

Function: Converts the address to a physical address.

Examples:

```
->PHYSICAL 1234:8767     //Converts a real mode virtual address.
->PHYSICAL 10:20         //Converts a protected mode virtual
->                       //address.
->PHY 12898778           //Converts a linear address.
```

See Also: LiNEar, VIRtual

PMOde PMOde

Syntax: PMOde

Function: Displays the current mode of the processor.

Displays REAL, V86, PROTECT16, or PROTECT32 mode, depending on the mode of the 80386 at the last breakpoint or HALT.

Remarks: This command is the same as the PMODE command of MICE-V 386.

Example:

```
->PMO
```

POInter POInter

Syntax: POInter [address] [= expression [, expression]...]
[[TO] address [= expression]]
[LENgth n [= expression]]

Function: Displays or alters memory contents in pointer (4-byte) scope. The base of two addresses that define an address range must be the same. For example, POINTER 200:40 to 300:300 is invalid.

Examples:

```
->POI 40
->POINTER 100:40 TO 100:200
->POI &pointer_buf LENGTH 20
->POINTER DS:SI = 9:6, CS:IP, SS:BP+SP
->POI pointer_array LEN 100 = 0:0
```

See Also: BYTe, CHAr, DOUble, DWOrd, FLOat, QWOrd, TREal, WORd

PRInt or DISplay TRAce

PRInt or DISplay TRAce

Syntax: {PRInt} [start_line [end_line]] [CLEar]

{DISplay TRAce}

Function: Displays the trace buffer. 'start_line' is the line number where the trace display begins. 'end_line' is the line number to end the display. CLEar clears the entire trace buffer (The CLEar key word can also be specified as CLR).

Remarks: This command is the same as the DISplay TRAce command.

The syntax of this command is similar to that of the DT command of MICE-V 386.

Examples:

```
->PRI 0 20          //Prints trace frames 0 to 20.
->PRI CLE           //Clears trace buffer.
```

See Also: DISplay TRAce, HTRc

PROlog PROlog

Syntax:

```
PROlog   [ON ]
         [OFF]
```

Function: Enables, disables, or displays the status of automatic prolog execution. If enabled, function prolog code will be automatically executed whenever the function is entered via Go, Step or ISTep. The default setting is ON. You can set the PROLOG variable to override the default setting in the environment file.

The prolog of a C function is the instructions at the beginning of the function that set up the local stack frame for the C function when it is entered.

Example:

```
->PRO OFF          //Disables automatic prolog execution.
```

See Also: B, ENV, Go, ISTep, Step

PUT PUT

Syntax: PUT "file_name" MACro [macro_name [, macro_name]...]

Function: Writes some or all macro definitions to a specified file.

Examples:

```
->PUT "mac.inc" MAC          //Write all macro definitions to the
->                           //file MAC.INC
->PUT "ABC.INC" MAC AA,BB,CC //Write macro definitions AA, BB,
->                           //and CC to the file ABC.INC
```

See Also: DIRectory MACro, DISplay MACro, EDit MACro, INClude, MACro, MLIst, REMove MACro

QUIt QUIt

Syntax: QUIt

Function: Terminates the debug session. This command closes all opened files and deletes all temporary files that are created by hyperSOURCE-386. You can also press the <Alt>x keys to terminate the debug session.

Remark: This command is the same as the EXIt command.

Example:

```
->QUIT
```

See Also: EXIt

QWOrd

Syntax:

```
QWOrd [address]    [= expression [, expression]...]
                   [[TO] address [= expression]]
                   [LENgth n [= expression]]
```

Function: Displays or alters memory contents in quad-word (8-byte) scope. The base of two addresses that define an address range must be the same. For example, QWORD 200:40 to 300:300 is invalid.

Examples:

```
->QWORD 40          //Display quad-word content of address DS:40
->QWORD 100:40 TO 100:200
->QWORD &unsigned_long_buf LENGTH 20
->QWORD DS:SI = 23, 234Q, 4+6, AL, 38T
->QWORD unsigned_long_array LEN 100 = 0
```

See Also: BYTe, CHAr, DOUble, DWOrd, FLOat, POInter, TREal, WORd

RADix

Syntax:

```
RADix   [HEX    ]
        [DECimal]
        [OCTal  ]
        [BINary ]
```

Function: Examines or sets radix for input numbers. Default radix is decimal. The qualifiers HEX, DEC, OCT, and BIN indicate hexadecimal, decimal, octal and binary, respectively.

Remarks: This command only affects the number input. Variable values are displayed in the radix that is consistent with the type. For examples, the values for short, int and long variables are displayed as decimal numbers, the values for char variables are displayed as ASCII characters and hexadecimal numbers, the values for double and float variables are displayed as floating point numbers. Use the EVAluate or = command to display the variable values as binary, octal, decimal and hexadecimal numbers.

Examples:

```
->RAD              //Examines current input radix
->RAD HEX          //Sets input radix to hexadecimal
```

See Also: BINary, DECimal, HEX, OCTal

RAMtst RAMtst

Syntax: RAMtst [load_addr]

Function: Loads a RAM test program into memory. BRkRidt, BRkPidt, and BRKgdt are set up automatically by this command.

'load_addr' is the address where RAM test is loaded into memory. If 'load_addr' is not specified, it is loaded at the same location as the last time the command was executed (0:0 on power-up). The 'load_addr' must be virtual. If you supply a virtual address, but do not specify a segment, then the DS register is used.

Remarks: This command is the same as the RAMTST command of MICE-V 386.

Examples:

```
->RAM 1000H:0
->DS=0
->ESI=0
->EDI=0FFFFH
->GO TIL 0FCH
```

See Also: MAP, RAmtstP, VeRiFy

RAmtstP RAmtstP

Syntax: RAmtstP [load_addr]

Function: Loads a protected mode RAM test program into memory. BRkRidt, BRkPidt, and BRKgdt are set up automatically by this command. The test program begins in real mode and switches to protected mode.

'load_addr' is the address where RAM test is loaded into memory. If 'load_addr' is not specified, it is loaded at the same location as the last time the command was executed (0:0 on power-up). The 'load_addr' must be virtual. If you supply a virtual address, but do not specify a segment, then the DS register is used.

Remarks: This command is the same as the RAMTSTP command of MICE-V 386.

Examples:

```
->RAP 1000H:0
->DS=0
->ESI=0
->EDI=0FFFFH
->GO TIL 0FCH
```

See Also: MAP, RAMtst, VeRiFy

RBRk RBRk

Syntax: RBRk [FROm address] register [NOT] expression [high_value]

Function: Single steps the processor until a register matches the specified value or falls within or outside the specified range.

FROm 'address' sets the CS and EIP registers to 'address' before the single stepping starts. 'address' must be a virtual address. 'register' is a 80386 registers. NOT breaks when the register does not match the specified value or is outside the specified range. 'expression' is the value to match. 'high_value' specifies the high end of a range, that is, from 'expression' through 'high_value.' The range bounds are inclusive.

Remark: Before using RBRk, you must first set up the stack pointer as follows:

For real mode addressing: ESP > = 6h

For virtual mode addressing: ESP > = 24h

For protected mode privilege level zero: ESP > = 0Ch

For protected mode privilege level non-zero: ESP > = 14h

The syntax of this command is similar to that of the RBRK command of MICE-V 386.

Examples:

```
->RBR AX 1234H
->RBR FROM 1000 SI NOT 5555H
```

See Also: REGister

RDYbrk RDYbrk

Syntax: RDYbrk [[=]ON]
[[=]OFF]

Function: Displays, enables, or disables the emulator's ready signal timeout break hardware.

ON stops the emulator with a break message when a cycle with more than RDYTO wait-states is encountered. If OFF, the emulator supplies READY for the current cycle and execution continues without a message provided that the RDYTO symbol is enabled. The power-up default is OFF.

Remarks: This symbol may be used to detect READY hang conditions in the target system hardware.

RDYBRK has no effect if the $READY signal is set to OFF.

This command is the same as the RDYBRK command of MICE-V 386.

Examples:

```
->RDY
->RDY ON
```

See Also: RDyTo, SIG, WSTate

RDyTo

Syntax: RDyTo [=expression]

Function: Specifies the emulator's ready timeout as 'expression.'

'expression' is the number of ready timeouts. 'expression' must be between 0 and 1Fh. The emulator supplies a READY for the current bus cycle after 'expression' wait-states. If RDYBRK is set to ON, then a "Ready Timeout Break" error will occur if 'expression' is set to more than the longest known wait-state used in the target system. The power-on default is 0, meaning the RDYTO function is disabled.

Remarks: RDYTO has no effect if the $READY signal is set to OFF.

This command is the same as the RDYTO command of MICE-V 386.

Examples:

```
->RDYBRK ON
->RDT=0F
->RDT=0
```

See Also: RDYbrk, SIG, WSTate

REAd REAd

Syntax: REAd symbol [, symbol]... [FROm n]

Function: Reads in symbol values from specified file n which is opened using the OPEN command. Default is from user's terminal.

Examples:

```
->READ A, *ptr_to_byte, ARRAY[4][3] FROM 2
->READ STRUCT.MEMBER, PTR_STR->FIELD
```

See Also: CLOse, INClude, OPEn, WRIte

REGister

Syntax: REGister
reg_name[=expression] [, reg_name[=expression]]...
reg_name:reg_name[=expression:expression]

Function: Displays or changes 80386 register values.

General purpose registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, AX, BX, CX, DX, SI, DI, BP, SP, AH, AL, BH, BL, CH, CL, DH, DL.

Segment registers: DS, ES, FS, GS, SS, CS.

Instruction pointers: EIP, IP.

Status registers: AF, CF, DF, IF, IOPL, NT, OF, PF, RF, SF, TF, VM, ZF, FLAG.

System table registers: GDTBAS, GDTLIM, IDTBAS, IDTLIM, LDTR, TR.

Control registers: CR0, CR2, CR3.

80287/387 registers: CW, TW, SW, ST0..ST7

Examples:

```
->REG                            //Displays current register values
->AX, BL, CH                     //Displays register AX, BL, AND CH
->AH=56Q, CL=101Y, IP=78T        //Changes register values
->GDTLIM
->CS:IP
->AX:BL = (678 + 6) : 4445
->cs:ip = &start
```

See Also: DT, GDT, IDT, LDT, PD, TSS

REMove MACro

Syntax: REMove MACro [macro_name [,macro_name]...]

Function: Removes the designated macro definition(s) from the macro symbol table. If no macro name is specified in the command, all macros are removed.

Examples:

```
->REM MAC                //Remove all macro definitions.
->REM MAC AA, BB, CC     //Remove macro AA, BB, and CC.
```

See Also: :, DIRectory MACro, DISplay MACro, EDit MACro, INClude, MACro, MLIst

REPeat REPeat

Syntax:

REPeat [n]
[command]
:
UNTil expression

Function: Executes the group of commands included between REPEAT and UNTIL, then evaluates the expression. If it is TRUE (non-zero), the group of commands are executed again and the expression is reevaluated. The loop continues until the termination condition is satisfied, i.e., the expression becomes FALSE (zero) or has looped n times if n is specified.

Examples:

```
->MACRO TEST1           //Define a macro that
MD>REPEAT %0            //at most repeats %0 times.
MD>A1 = A1 / 2.
MD>B = ROUTINE( A )     //Call routine, return value to B
MD>UNTIL B == %1        //Break if B equals to %1
MD>EMACRO
->
```

See Also: FOR, IF, INClude, MACro, UNTil, WHIle

RESet

Syntax: RESet

Function: Resets the 80386 emulator or processor.

Example:

```
->RESET
```

See Also: HALt, REGister

RUNning RUNning

Syntax: RUNning

Function: Displays the 80386 processor status. The status is either ON or OFF. ON indicates that the processor in the probe is executing code and that all monitored systems are working. OFF means that the probe is not executing user code. When stopped, the emulator can be used to modify memory, load a program, etc.

Remark: This command is the same as the RUNNING command of MICE-V 386.

Examples:

```
->RUN
```

See Also: RESet

SENsitive

Syntax:

```
SENsitive [ON ]
          [OFF]
```

Function: Sets, disables, or examines the status of case sensitivity in matching symbol names.

If SENsitive is off, symbolic reference will be case insensitive. If SENsitive is on, symbolic reference will be case sensitive. The default setting is SENsitive OFF.

Examples:

```
->SEN                //Examines case sensitivity.
->SEN ON             //Makes symbolic reference case sensitive.
->SEN OFF            //Makes symbolic reference case insensitive.
```

See Also: EVAluate, SYMbol

SET SET

Syntax: SET $module_name = ["]file_name["]

Function: Associates the specified file to the specified module name. The specified file is treated as the source listing file for the specified module.

Examples:

```
->SET $MAIN = "START.C"        //Default is MAIN.C
->SET $MEMORY = "STORAGE.C"    //Default is MEMORY.C
```

See Also: DIRectory MODule, SOUrce

SIG SIG

Syntax:

```
SIG  [E]
     [D]
     [COP     [=ON ]]
              [=OFF]
     [HOLd    [=ON ]]
              [=OFF]
     [INTr    [=ON ]]
              [=OFF]
     [NMI     [=ON ]]
              [=OFF]
     [REAdy   [=ON ]]
              [=OFF]
     [RESet   [=ON ]]
              [=OFF]
```

Function: Displays current status of target signals and enables or disables them.

E enables and D disables all target signals. The power-up default is D. If no argument is given, a list of signals and their status is displayed. ON means the input signals generated by the target can reach the 80386. OFF means the input signals are masked and cannot reach the 80386.

COP is the coprocessor signals ERROR#, BUSY#, and PEREQ. HOLd is the hold processor input signal. INTr is the INTR processor input signal. NMI is the non-maskable interrupt processor input signal. REAdy is the processor's ready input signal. RESet is the processor's reset input signal.

Remark: The syntax of this command is similar to the SIG, $COP, $HOLD, $INTR, $NMI, $READY, and $RESET commands of MICE-V 386.

Examples:

```
->SIG
->SIG E
->SIG D
->SIG COP = ON
->SIG HOL = OFF
->SIG NMI
```

See Also: RDYbrk, RDyTo, WSTate

SOUrce

Syntax: SOUrce [["]file_name["]]
[$module_name]
[##procedure_name]

Function: Enters the source window for the specified file.

Examples:

```
->SOU $MAIN
->SOU ##RFREE
->SOU PROG.C
```

See Also: #, DIRectory MODule, NUMber, SET, SPAth, VIEw

SPAth SPAth

Syntax: SPAth [=] [directory [; directory]...]

Function: Displays or sets the search path for source files. Source files are always searched for in the current directory first.

Examples:
```
->SPA             //Display the current path.
->SPA c:\proj1\src;d:\proj2\src
```

See Also: DIRectory MODule, ENV, SET, SOUrce

ST0, ST1, ST2, ST3, ST4, ST5, ST6, ST7

ST0, ST1, ST2, ST3 ST4, ST5, ST6, ST7

Syntax: STn

where n = 0 to 7.

Function: Displays or updates the 80387 stack elements and the corresponding tag words. The value of the stack element is first displayed as a treal number. It can be altered by entering a new value. A carriage return will preserve the contents. If a carriage return is entered, the stack element is once again displayed, but this time it is displayed as ten hexadecimal values. It can be altered by entering a new set of ten hexadecimal values. A carriage return will preserve the contents. Next, the tag word that corresponds to the stack element is displayed. It can be altered by entering a new value. A carriage return will preserve the contents.

Examples:

```
->ST4
  ST(4) = 0./ <CR>
  ST(4) = 00H 00H 00H 00H 00H 00H 00H 00H 00H 00H
        / 0F1H,0FFH,0F7H,98H,00H,56H,43H,69H,8AH,7CH
  TAG(4) = 3 / 0
->ST3
  st(3) = 7.8 / 0.0
  tag(3) = 0 / 3
```

See Also: CW, SW, TW

Step

Syntax: Step [INto] [n]

Function: Step causes the program to execute n statements before halting for debugging purposes. If n is not specified, the default is 1, which allows single-statement debugging.

Step INto will step into the called function.

Examples:

```
->S 10           //Execute 10 statements and stop.
->S              //Execute one statement and stop.
->S IN           //Step into the called function.
```

See Also: Go, IStep

STRucture

Syntax:

```
STRucture structure_name = (
{data_type}field_name[,field_name]...
{<symbol>}
  :
)
```

Function: Defines a new data structure and the data type of each field in it. However, existing data structures cannot be redefined. When you enter the left parenthesis, hyperSOURCE-386 enters the structure field definition mode with the **STR>** prompt. You must enter the correct field definition on each line. When you enter the right parenthesis, the command is completed.

Examples:

```
->STR NEWSTR = (
STR> WORD *i_link
STR> BYTE i_type
STR> DWORD i_ecw[7]
STR> DOUBLE *p_real
STR> STR NEWSTR *p_str
STR>)
->type struct newstr new_name at &start   //Use the new
->                                        //structure
->str newst =(                            //variable x is of type
->                                        //byte.
STR> type <x> *i_link, float i_type, word i_ecw)
->STR STRTYP =(
STR>WORD AA, BB, CC
STR>CHAR *P_TO_CHAR, CHAR_ARRAY[7][8]
STR>LONG *LL[6], (*FF)[5]
STR>STR STRTYP *LINK
STR>TYPE <AA> ZZ
STR>)
```

See Also: DIRectory STRucture, DISplay STRucture, SYMbol, TYPe

SW SW

Syntax: SW

Function: Displays or changes the value of the 80387 status word. The value of the status word is displayed followed by a slash. The contents can be altered by entering a new hexadecimal value. A carriage return will preserve the contents.

Examples:

```
->SW
  STATUS WORD = 0000H / 1324H
->SW
  STATUS WORD = 1324H / <CR>
```

See Also: CW, STn, TW

SWItch SWItch

Syntax:

SWItch expression
CASe constant_expression :
[command]
:
[CASe constant_expression :]
[command]
:
[DEFault :]
[command]
:
ESWitch

Function: A multi-way decision maker that tests whether an expression matches one of a number of constant values, and branches accordingly. If none is matched, control flow is branched to the DEFault case. After having executed the group of commands associated with the matched case, control flow falls through to the next CASe/DEFault unless a BREak command is encountered. The BREak command causes an immediate exit from the SWItch.

Examples:

```
->MACRO TEST1      //Define a macro.
MD>SWITCH A
MD>  CASE 0 :
MD>        B = 10
MD>        BREAK
MD>        BREAK
MD>  DEFAULT:
MD>        LINE 5
MD>ESWITCH
MD>EMACRO
```

See Also: ESWitch, FOR, IF, INClude, MACro, REPeat, WHIle

SYMbol

Syntax: SYMbol [GLObal]
[LOCal]
[$module_name]
[symbol_name]
[address]
["reg_exp"]

Function: Displays symbol declarations. If GLOBAL is specified, only the global symbols are displayed. Likewise, if LOCal is specified, only the currently active local symbols are displayed. If a module name is specified, the symbols belonging to the specified module are displayed.

If a symbol name is specified, the declaration for that symbol is displayed. If an address is specified, the global symbol with the closest matching address is displayed. And if a string is specified, the symbols matching the regular expression in quotes will be displayed.

Examples:

```
->SYMBOL
->SYM GLO
->SYM LOC
->SYM $MODULE_AA
->SYM main
->SYM CS:100H
```

See Also: EVAluate, STRucture, TYPe

TIMe

Syntax:	TIMe
Function:	Displays the current time and date.
Example:	`->TIM`
See Also:	VERsion

TKB

Syntax:

```
TKB [selector [,selector]...] [PERM]
                              [TEMP]
                              [CLEar]
                              [E]
                              [D]
```

Function: Displays, sets or clears task switch breakpoints. The CLEar key word can also be specified as CLR.

'selector' is the task state segment (TSS) to be displayed, set, or cleared. Any number of breakpoints are allowed. If 'selector' is not specified, all current breakpoints are affected according to the specified key word (PERM, TEMP, CLEar/CLR, E, or D). If 'selector' is specified without another parameter (PERM, TEMP, CLEar/CLR, E or D), then all current task switch breakpoints are PERManent. 'selector' must reference a TSS descriptor in the global descriptor table (GDT). PERM is a permanent task switch breakpoint that remains until it is cleared. If 'selector' is not specified, then all current task switch breakpoints are PERManent. TEMP is a temporary task switch breakpoint that is automatically cleared when it is reached. If 'selector' is not specified, then all current task switch breakpoints are TEMPorary. CLEar or CLR clears the selector(s) specified from the list. If 'selector' is not specified, then all current task switch breakpoints are cleared. E enables and D disables one or more breakpoints. Disabling breakpoints leaves the definitions in the table but does not affect emulation until they are re-enabled. If 'selector' is not specified, then all task switch breakpoints are affected.

Remarks: The emulator implements task switch breakpoints by setting the DEBUG_TRAP bit in the specified TSS. The TSS must be in RAM and accessible when the TKB command is entered. If the 80386 invokes the selected task during emulation, a breakpoint occurs. The DEBUG_TRAP bit remains set until the task switch breakpoint is cleared.

This command is the same as the TKB command of MICE-V 386.

Examples:

```
->TKB 20 28 38 PERM
->TKB
->TKB 20 28 CLE
```

See Also: BRKgdt, BRkPidt, BRkRidt, Go, GR

TM

Syntax: TM [MICE_command_sequence]

Function: Enters transparent mode to issue MICE-V commands directly. The MICE-V prompt is displayed. To leave transparent mode, press <Ctrl>a.

If a command sequence follows the TM command, it will be sent directly to the MICE-V 386. No parsing is done. No error checking is done. No results are passed back up to hyperSOURCE-386. This facility is provided solely for automating tested MICE-V 386 commands via include files and/or macros.

Note

Changes made in the emulation environment while in transparent mode are not tracked by hyperSOURCE-386 and can result in erroneous operation.

Remarks: Transparent mode provides access to the low-level command line and command interpreter resident in the MICE-V emulator. This command line interface is a whole input/output system by itself. It has the ability to parse commands and store results of command usage in its own variables. This extra layer of control variables can lead to confusion.

If you issue a command from hyperSOURCE-386, such as MAP, the results of the command are stored by the hypeSOURCE-386 interface. If you now enter transparent mode and view the MICE-V command line variables, it will appear that the command has not been executed. This is because the variables in the MICE-V command line interface are not set by the command issued from hyperSOURCE-386.

Because this is true with many of the commands in transparent mode, use hyperSOURCE-386 to issue commands and check variable values; use transparent mode only to view the trace buffer and set high level triggers.

If your PC host is not able to keep pace with the rate that characters are displayed when in Transparent Mode, try altering the value of the TMDELAY variable in your environment file.

Examples:

```
->TM                //Enters transparent mode.
>
> ^A                //Exits transparent mode.
```

See Also: ENV

TRCmode

Syntax: TRCmode [PRE]
[POST]
[CENTER]
[SINGLE [count]]

Function: Specifies the type of trace collection to use.

PRE collects up to 8192 frames of trace before the trigger, then forces a breakpoint. The trigger event will be nearly the last bus cycle in the buffer. Actually, a few bus cycles "slide" before emulation and the trace finally stop. POST collects 8192 frames of trace after the trigger, then forces a breakpoint. CENTER collects 4096 frames before and after the trigger, then forces a breakpoint. SINGLE collects a single frame of trace. No breakpoint is forced. The power-up default is SINGLE 1. 'count' specifies how many cycles to trace when in SINGLE mode. 'count' can be any number from 1 through 16t. If 'count' is not specified, it defaults to 1.

Remarks: If TRCmode is set to POST, CENTER, or SINGLE with CNT greater than 1, then the occurrence counter (CNT) cannot be used in the trigger definition.

The TRCmode command is the same as the TRCMODE command in MICE-V 386.

Examples:

```
->TRC SINGLE 3
->TRC
->TRC PRE
```

TREal

Syntax:

```
TREal [address]  [= expression [, expression]...]
                 [[TO] address [= expression]]
                 [LENgth n [= expression]]
```

Function: Displays or alters memory contents in treal (10-byte) scope. The base of two addresses that define an address range must be the same. For example, TREAL 200:40 to 300:300 is invalid.

Examples:

```
->TRE 40
->TREAL 100:40 TO 100:200
->TRE &REAL LENGTH 20
->TREAL DS:SI = 8.8, 3.5+1, 0.0
->TRE pointer_to_treal LEN 100 = 0:0
```

See Also: BYTe, CHAr, DOUble, DWOrd, FLOat, POInter, QWOrd, WORd

TSS TSS

Syntax: TSS [\[expr\]][.tss_element]]

where tss_element is LINk, {[E]SP|SS}{0|1|2}, or a register.

Function: Displays or modifies the contents of a task state segment.

Examples:

```
->TSS                     //Displays current TSS.
->TSS[8].LINK=6890        //Modifies the link field of the TSS whose
->                        //descriptor is at GDT(1) or the TSS
->                        //selector equals 8.
```

See Also: DT, GDT, IDT, LDT, PD, REGister

TW TW

Syntax: TW

Function: Displays or changes the value of the 80387 tag word. The value of the tag word is displayed followed by a slash. The contents can be altered by entering a new hexadecimal value. A carriage return will preserve the contents.

Examples:

```
->TW
  TAG WORD = 0000H / 0FFFFH
->TW
  TAG WORD = FFFFH / <CR>
```

See Also: CW, STn, SW

TYPe

Syntax: TYPe {data_type} symbol [, symbol]... [AT address]
{<symbol>}

Declares or redefines symbols. The data type can be BYTe, CHAr, WORd, SHOrt, DWOrd, LONg, INTeger, FLOat, DOUble, TREal, STRuct, or a pointer to these basic types. If a POInter is declared, the size of offset, 16-bits or 32 bits, depends on the setting of the USE or WIDth command.

The <...> construct can be used to declare the variable to be of the same type as the variable enclosed in the <...> pair.

If no AT address is specified, the symbol uses internal debugger memory. These symbols are called internal variables. Pointers to internal variables are not allowed.

Examples:

```
->TYPE long *ptr_to_long, long_buf[8] at &buf
->TYPE CHAR CH1, CH2[3][4], (*CH3)[7]
->TYPE struct str_aa str1, *str2, str3[7] at &str_buf
->TYPE <yy> xx AT 200:10
->type char $m##pro#a, ##pp#b, #c at 8:9
```

See Also: STRucture, SYMbol

U or DASm

Syntax:

```
{U}       [address1    [[TO] address2]]  [MIX]
{DASm}                 [LENgth n]
```

Function: Displays a block of memory in assembly mnemonic form. The MIX qualifier causes source to be mixed in with the disassembly display.

Remark: This command is the same as the DASm command.

Examples:

```
->U                   //Default address is CS:IP
->U CS:(IP+5) MIX
->U &MAIN LEN 20
```

See Also: SOUrce, VIEw

UP UP

Syntax: UP [n]

Function: Walks up the call stack allowing access to the source and local variables of any active procedure. If no argument is specified, the stack is walked up one level.

If any execution command or command that directly changes the CS:IP or BP is given by the user while an UP or DOWN command is in effect, a DOWN HOME action is automatically performed before the command is executed.

Examples:

```
->UP                  //Walk up one level
->UP 3                //Walk up three levels
```

See Also: CALlstack, DOWn, SOUrce, SYMbol

USE USE

Syntax: USE [16]
[32]

Function: Displays or modifies the default setting for disassembling code. The default setting causes the debugger to (1) assume 16-bit or 32-bit code when disassembling code from linear or physical addresses using the ASM command, (2) assume 16-bit or 32-bit offset when handling POInter type, (3) assume 16-bit or 32-bit offset when a symbol is declared as pointer type using the TYPE command.

Examples:

```
->USE 16            //Sets to 16-bit code.
->USE               //Displays the current setting.
```

See Also: DASm, TYPe, STRucture, POInter

VeRiFy VeRiFy

Syntax: VeRiFy [ON]
[OFF]

Function: Displays, enables, or disables the verification of memory write operations. If no argument is specified, the current setting is displayed.

Remark: This command is the same as the VERIFY command of MICE-V 386.

The abbreviation is VRF in order to avoid conflict with the VERsion command.

Examples:

```
->VRF
->VRF ON
->VRF OFF
```

See Also: BYTe, CHAr, DOUble, DWOrd, FLOat, POInter, QWOrd, TREal, WORd

VERsion

Syntax:	VERsion
Function:	Displays hyperSOURCE-386 and emulator version numbers.
Example:	`->VER`
See Also:	DATe

VIEw VIEw

Syntax:

```
VIEw [HL ]
     [ASM]
     [MIX]
```

Function: Displays or modifies the mode of source line display in the source window. HL sets the source window to display high level language source statements only. ASM sets the source window to display assembly language mnemonics only. MIX sets the source window to display high level language source statements interleaved with assembly language mnemonics. If no parameter is specified, it displays the current display mode of the source window.

Examples:

```
->VIEW            //Displays current setting.
->VIEW MIX        //Displays high level source and assembly.
```

See Also: #, NUMber, SOUrce

WAIt

Syntax: WAIt

Function: Stops command processing, resumes when user presses any key from the keyboard. The WAIt command is useful in command files for demonstrations or interactive automated testing. A typical usage is to halt the display until the user presses a key.

Example:

```
->WAIT
```

See Also: INClude, PAUse

WHIle

WHIle

Syntax: WHIle expression
[command]
:
EWHile

Function: In the WHILE-EWHILE loop command, the expression is evaluated first. If it is TRUE (non-zero), the group of commands listed between WHILE and EWHILE are executed and the expression is evaluated again. This loop is repeated until the expression becomes FALSE (zero).

Examples:

```
->MACRO TEST1      //Define a macro.
MD>WHILE NOT_ZERO
MD>LINE            //Single line.
MD>NOT_ZERO        //Display NOT_ZERO value.
MD>EWHILE
MD>EMACRO
->
```

See Also: EWHile, FOR, IF, INClude, MACro, REPeat, SWItch

WIDth

Syntax: WIDth [16]
[32]

Function: Displays or modifies the default setting for disassembling code. The default setting causes the debugger to (1) assume 16-bit or 32-bit code when disassembling code from linear or physical addresses using the ASM command, (2) assume 16-bit or 32-bit offset when handling POInter type, (3) assume 16-bit or 32-bit offset when a symbol is declared as pointer type using the TYPE command. This command is identical to the USE command.

Examples:

```
->WID 16          //Sets to 16-bit code.
->WID             //Displays the current setting.
```

See Also: DASm, POInter, STRucture, TYPe

WORd WORd

Syntax:

```
WORd [address] [= expression [, expression]...]
               [[TO] address [= expression]]
               [LENgth n [= expression]]
```

Function: Displays or alters memory contents in word (2-byte) scope. The base of two addresses that define an address range must be the same. For example, WORD 200:40 to 300:300 is invalid.

Examples:

```
->WORD 40
->WOR 100:40 TO 100:200
->WORD pointer_to_word LENGTH 20
->WOR DS:SI = 5, 8*6, AX+BX
->WORD word_array LEN 100 = 0:0
```

See Also: BYTe, CHAr, DOUble, DWOrd, FLOat, POInter, QWOrd, TREal

WRIte

Syntax:

```
WRIte   { "string" } [,   { "string" }]... [TO  n]
        {expression}      {expression}
```

Function: Writes strings or values of expressions to the specified file n which must have been opened using the OPEN command. Default is to the console.

Certain non-graphic characters, the double quote and the backslash characters may be represented by escape sequences and included in the character string as follows:

newline	\n
horizontal tab	\t
backspace	\b
carriage return	\r
backslash	\\
double quote	\"

Examples:

```
->WRITE "\tIOPB=", IOPB        //"\t" is a tab.
->WRITE "\nVALUE OF X IS", X  //"\n" is a newline.
->WRITE a[6] + *ptr_to_short + struct.a1 to 3
->WRITE "a1 = ", a1, "a2 = ", a2, "a3 = ", a3
```

See Also: CLOse, INClude, OPEn, REAd

WSTate WSTate

Syntax: WSTate [=expression]

Function: Displays or sets the number of wait-states to insert when the $READY signal is internal (OFF).

'expression' is the number of wait states. 'expression' must be 0 through 1Fh. WSTate may be set at any time but is effective only if the $READY signal is set to OFF.

Remark: This command is the same as the WSTATE of MICE-V 386.

Examples:

```
->SIG READY OFF
->WST=4
```

See Also: RDYBRK, RDYTO, SIG

XLT XLT

Syntax: XLT address

Function: Translates a virtual address to a linear and physical address, or a linear address to a physical address using segmentation and paging rules.

'address' is the address to translate. This may be a virtual address, linear address (followed by the letter 'n'), or physical address (followed by the letter 'p').

Remark: This command is the same as the XLT command of MICE-V 386.

Examples:

```
->XLT CS:4567
->XLT 1234N
```

Chapter Seven - Macros

This chapter describes the macro facility. It enables you to use hyperSOURCE-386's commands to form aggregate commands called macros. Macros may contain arguments that can be substituted by actual values when a macro is being expanded.

Using the Flow Control Commands

The flow control commands are used to control the sequence of command execution. The flow control commands are most often used in macros, although they can be used in the command line.

The syntax of the flow control commands is very similar to the C language. The flow control command set includes the following commands which are described in Chapter Six - Command Reference.

Program sequence alteration control:

BREAK
CONTINUE
GOTO

Decision making blocks:

IF/ORIF/ELSE/EIF
SWITCH/CASE/DEFAULT/ESWITCH

Command loops:

FOR/EFOR
REPEAT/UNTIL
WHILE/EWHILE

Defining Macros

A macro definition consists of the following three parts:

1. Macro command name, recognized by the first six alphanumeric characters.

2. Argument list, up to ten arguments.

3. Macro body which consists of a sequence of commands.

You can define macros in the debug session using the MACRO command.

An easier way is to use an editor to define macros. You can invoke an editor in hyperSOURCE-386 in any one of the following ways:

1. Enter the EDIT MACRO command with the name of the macro on the command line.

2. Press <Alt>a to popup the mAcro menu, select **Edit** and enter the name of the macro.

The editor is defined by the EDITOR variable in the environment file. The EDITOR variable is described in *Preparing the Environment File* in Chapter Two.

A third way is to first use an editor to define macros in a text file before you start a debug session, then load in the macros during the debug session.

Before the debug session ends, you can save any or all of the macros to a file for future use (use the PUT command).

When you define a macro with the MACRO command on the command line, the prompt changes from **->** to **MD>**. After you have entered the EMACRO command, the prompt changes back to **->**. For example,

```
->load mac.inc              //Load macro definitions.
->directory macro           //List directory of macros.
->macro mymac               //Define a new macro called mymac.
MD>map 0 len 1/r            //First input line for macro body.
MD>if %0 > 0                //A macro argument.
MD>goto done
MD>step                     //Single step one instruction
MD>eif
MD>done:                    //A macro label.
MD>emacro                   //End of macro definition.
->put "mac1.inc macro"      //Save all macros to file.
```

Each line of a macro definition should contain only one command. The command line may include macro arguments. The size of a macro definition and the number of macro definitions that can be loaded into the debug session are limited only by the available memory.

A macro definition may contain labels. A label is a symbol suffixed with a colon. A label specifies an entry point for command execution. Labels are often used in the GOTO command.

HyperSOURCE-386 does not check for command syntax errors in the macro definition input mode, except the flow control constructs. If you invoke the MACRO command to define a macro, hyperSOURCE-386 checks for completion of the flow control constructs. For example, if a FOR command is specified, hyperSOURCE-386 expects a EFOR command is also specified in the macro definition. HyperSOURCE-386 also checks the syntax of the expressions used in the flow control commands.

A macro may be used to invoke another macro, but not to define another macro. In other words, the macro body may not contain any MACRO command. Each macro may have up to ten arguments, identified as %0 through %9. For example,

```
->macro config          //Create a macro named config.
MD>load %0              //Load file.
MD>setbrk               //Invoke another macro to set breakpoints.
MD>ema                  //End of macro definition.
->config test           //Execute macro named config which loads a file
                        //named "test".
```

Displaying Macros

Once the macros are loaded in hyperSOURCE-386, you can display the macro definitions in any one of the following ways:

1. Enter the DISPLAY MACRO command on the command line.

2. Press <Alt>a to popup the mAcro menu, select **Dir** to display the macro directory. Move the highlight to the desired macro name and press <Enter>.

In the DISPLAY MACRO command, if you specify more than one macro name as arguments, the arguments have to be separated by commas. If you do not specify any macro names, all macro definitions will be displayed. For example,

```
->display macro mymac,config    //Display mymac and config.
->dis mac                       //Display all macros.
->dis mac config                //Display config only.
```

Displaying Macro Directory

You can display the macro directory in any one of the following ways:

1. Enter the DIRECTORY MACRO command on the command line.
2. Press <Alt>a to popup the mAcro menu, select **Dir**.

The names of all the macros that are present in hyperSOURCE-386 will be displayed.

Deleting Macros

You can delete macros in any one of the following ways:

1. Enter the REMOVE MACRO command on the command line.
2. Press <Alt>a to popup the mAcro menu, select **Remove** and specify the macro names.

If you specify more than one macro name as arguments, the arguments have to be separated by commas. If you do not specify any macro names, all macro definitions will be deleted. For example,

```
->remove macro mymac,config    //Delete mymac and config.
->rem mac                      //Delete all macros.
->rem mac config               //Delete config only.
```

Invoking Macros

You can invoke macros in any one of the following ways:

1. Enter the macro name on the command line.
2. Press <Alt>a to popup the mAcro menu, select **Invoke** and specify the macro name.

If the macro accepts arguments, you may enter values for the arguments after the macro name. If an expected argument is missing, the macro expansion may produce unpredictable results. A macro may accept up to 10 arguments. The argument values are separated by commas. If an argument value contains commas, the "<*" and "*>" operators can be used as parentheses in specifying the value. For example,

```
->config test                        //Invoke macro config with argument.
->abc 24,count                       //Two arguments.
->aa 34, <* ax,bx,cl *>, #20         //Three arguments.
->bb                                 //No argument.
```

Saving Macros to a File

You can save macros to a file in any one of the following ways:

1. Enter the PUT command on the command line.

2. Press <Alt>a to popup the mAcro menu, select **Save** and specify the file name and the macro names.

More than one macro definitions can be saved to a text file. If you do not specify any macro names, all macro definitions will be saved. For example,

```
->put mac.inc macro mymac,config     //Save two macros to file.
->put macdef.inc macro               //Save all macros to file.
```

Loading Macros from a File

You can load macros from a file in any of the following ways:

1. Enter the @ or INCLUDE command with the file name as argument on the command line.

2. Press <Alt>a to popup the mAcro menu, select **Load** and specify the file name.

Debugging Macros

The MLIST command is used primarily for debugging macros. You can enter the MLIST in any of the following ways:

1. Enter MLIST command on the command line.

2. Press <Alt>c to popup the Config menu, select **mAcro listing** and then select **On** or **oFf.**

This MLIST ON command causes the commands in the macro definition to be displayed on the dialog window as the macro is being expanded. The MLIST OFF command can be used to disable the display. The default setting is MLIST OFF. The MLIST command without any arguments displays the setting. For example,

```
->mlist                     //Examine setting.
->mlist on                  //Enable macro debug.
->mymacro                   //Display command during macro expansion.
->mlist off                 //Disable macro debug.
```

Index