

## **Windows System Debugger**



[About Windows System Debugger](#)




[Using the System Debugger](#)





[Reference](#)





## Windows System Debugger


 About Windows System Debugger


 Introduction


 Preparing to Start


 Connecting a Serial Cable

 Creating Symbol Files

 Installing Debug Environment

 Starting from the Command Line


 Starting as a VxD


 Using the System Debugger

 Reference




## **Windows System Debugger**


 About Windows System Debugger

 Using the System Debugger

 Introduction

 Breaking into the Debugger

 Determining the State of the Processor








 Setting Breakpoints

 Setting the Default Command

 Reference



## Windows System Debugger

-  About Windows System Debugger
-  Using the System Debugger
-  Reference
  -  **Command Groups**
  -  Command Syntax
  -  Commands
  -  Dot Commands



## About Windows System Debugger



## **Introduction**

Microsoft Windows System Debugger (WDEB386.EXE) is used to test and debug Windows applications, dynamic-link libraries (DLLs), and virtual device drivers (VxDs) running with the Microsoft Windows operating system. You use System Debugger commands to inspect and manipulate memory and registers, control the execution of code, and perform other debugging operations. The topic describes the debugger, explains how to start and use the debugger, and provides a detailed information about the debugger commands.



## Preparing to Start

You can start the System Debugger as an application from the real-mode command line or as a virtual device driver by specifying it in the SYSTEM.INI file. Before you start, however, you prepare for debugging by carrying out these steps:

- Connect a serial terminal or other computer by serial cable to a serial port on your debugging computer.
- Create symbol files for the applications, DLLs, and VxDs you intended to debug.
- Install the system debug environment if available. These are debugging versions of the system DLLs and VxDs and their symbol files that can make debugging easier. These files are available in the Windows 95 Device Driver Development Kit.



## **Connecting a Serial Cable**

The debugger displays output and receives user input through a serial communication port of the computer on which it runs. To use the debugger, you must connect a cable from this serial port to either a serial terminal or the serial port of another computer. A three-wire null modem cable is the minimum cable requirement for this connection. In a three-wire null modem cable, the TxD (transmit data) and RxD (receive data) lines are in opposite positions at the two ends of the cable, but the signal ground is connected straight through. DTR (data-terminal-ready) and CTS (clear-to-send) handshaking is not used.

Typically, users connect the cable to a second computer and use their favorite communications software to open the serial port and interact with the debugger.



## Creating Symbol Files

Symbol files provide the information the debugger needs to display functions, structures, variables, and absolute symbols by name rather than number. To prepare symbol files, perform the following steps:

1. Compile or assemble your source files, using the appropriate command-line option to generate object files with line-number information. For more information about compiler and assembler options, see the documentation that accompanied your compiler and assembler.
2. Link the compiled code with the standard libraries (as needed), using the appropriate linker option to create a symbol map (.MAP) file that includes **PUBLIC** symbols. You may also want to use the linker option for display of line-number information. For more information about linker options, see the documentation that accompanied your linker.
3. Run the Microsoft Symbol File Generator (MAPSYM.EXE) to create a symbol file for symbolic debugging. MAPSYM converts the contents of your application's symbol map (.MAP) file into a form suitable for loading with the debugger; then MAPSYM copies the result to a symbol (.SYM) file.

Following is the command-line syntax for MAPSYM:

**mapsym** [/I]/[n] *mapfilename*

**/I**

Directs MAPSYM to display information on the screen about the conversion. The information includes the names of groups defined in the application, the application start address, the number of segments, and the number of symbols per segment.

**/n**

Directs MAPSYM to ignore line-number information in the map file. The resulting symbol file contains no line-number information.

*mapfilename*

Specifies the filename for a symbol map file that was created during linking. If you do not give a filename extension, .MAP is assumed. If you do not give a full path, the current directory and drive are assumed. MAPSYM creates a new symbol file having the same name as the map file but with the .SYM extension.

In the following example, MAPSYM uses the symbol information in FILE.MAP to create FILE.SYM in the current directory on the current drive:

```
mapsym /I file.map
```

Information about the conversion is sent to the screen.

MAPSYM always places the new symbol file in the current directory on the current drive. MAPSYM can process up to 10,000 symbols for each segment in the application and up to 1024 segments.

If you have many components to debug, you can combine multiple symbol files into a single file by using the Symbol File Librarian (SYMLIB.EXE). This creates a symbol library file and lets you add, remove, or replace .SYM files in it.



## Installing Debug Environment

If you have the Windows 95 Device Driver Development Kit, you can install the system debug environment by using the DDKDEBUG.BAT batch file. This batch file installs the debugging version of selected system DLLs and VxDs, and creates a RUNWDEB.WRF file that you can use with the RUNWDEB.BAT batch file to start the debugger from the command line.

To install the debug environment, change directories to the DDK\DEBUG directory and type:

```
ddkdebug set
```

The RUNWDEB.WRF file contains command-line options for the debugger. This file along with RUNWDEB.BAT is copied to the WINDOWS\SYSTEM directory (provided that files with these names are not already present in that directory).

Before you can use RUNWDEB.BAT, you may need to edit it to contain the correct debug settings. These settings are:

- /c:** Number which represents the COM port to which debug terminal is attached. For example: /c:1
- /r:** Baud rate at which debug terminal will be operating. For example: /r:19200

You can add any other desired command line options to this file.

You may also need to edit the RUNWDEB.WRF to add **/s:** options for the symbol files of the components you intended to debug.

Once you have completed editing these files and verifying settings, you can start the debugger by running RUNWDEB.BAT (with one optional WDEB386 command line parameter) from the WINDOWS\SYSTEM directory.

You can also use DDKDEBUG.BAT to remove the system debug environment and restore the original system files. Type:

```
ddkdebug restore
```



## Starting from the Command Line

You can start the debugger from the real-mode command line by using the WDEB386 command line. (You cannot start the debugger from the command line after WIN.COM has started. To get the real-mode command line, press the F8 key as Windows starts and choose the Command Prompt Only option.) The command line has this syntax:

**wdeb386** [/A] [/B] [/C:*comport*] [/D:"*commands*"] [/E] [/F:*filename*] [/H] [/I] [/L] [/N] [/T:*hhhh*] [/S:*symfile*] [/V[P]] [/X] *winfile* [*parameters*]

Following are the command-line options and parameters:

### /A

Specifies that symbol files should not be automatically loaded.

### /B

Specifies that the debugger should stop just prior to VMM initialization, after all virtual devices have been loaded and the processor is running in protected mode.

### /C:*comport*

Specifies a COM port for debugger output. You can specify "1", "2", "3", or "4" for *comport*. If this option is not specified, The debugger checks first for COM2. If COM2 is not found, the debugger then checks for COM1. If neither COM1 nor COM2 exists, the debugger checks for any other COM port in the read-only memory (ROM) data area (40:0).

### /D:"*commands*"

Carries out the debugger command line specified by the string enclosed in quotation marks. Spaces, semicolons (;), and other punctuation can be included in the command string. To use a single quote (') on the command line, use double quotation marks (") before and after the single quotation mark.

The commands specified in this option are carried out after symbols are loaded. This means you can set breakpoints in code even before the code has been loaded. Before a segment or module has been loaded or defined, breakpoints can be set on the logical address (a combination of map number and group number) until the segment or module is defined, at which point the breakpoint turns into a real breakpoint.

### /E

Specifies that the debugger should stop at real-mode entry.

### /F: *filename*

Specifies a file containing command-line options for the debugger. Maximum file size is 4K, and the input file cannot contain the /F option.

### /H

Specifies that the debugger should be loaded as a VxD.

### /I

Specifies that the debugger should be invisible to int 41.

### /L

Specifies that line numbers should not be included in the .SYM file. This can make a large difference in memory use, and may be required on a machine with 4 megabytes of RAM.

### /N

Sets the following options:

**dislwr**  
**codebytes**  
**symaddrs**  
**int3line**  
**newvec**  
**newreg**  
**newprompt**

For information about these options, see the **y** command in [Reference](#).

### /R:*dddd*

Sets the baud rate for the debugging terminal.



***/S: symfile***

Specifies a symbol file to be loaded. This option can be repeated to load more than one symbol file. If the symbol files are not in your current directory, you must supply a full path, because the debugger does not use the PATH environment variable to locate any of the files supplied on the command line.

When memory is low, you can use more symbol files by running the Debugger in the Windows directory and specifying the full path of VMM32.VXD (such as \WINDOWS\SYSTEM\VMM32.VXD) instead of WIN.COM.

***/T:hhhh***

Sets the port number for the timing card. (The default number is 250h.)

***/V***

Enables verbose mode, which displays messages indicating which segments are being being loaded.

***/VP***

Enables verbose mode, which displays messages indicating which segments are being loaded. This option displays the messages for applications only.

***/X***

Causes symbols to be loaded into Extended Memory Specification (XMS) memory.

***winfile***

Specifies the Windows application to run under debugger control. You will usually specify WIN.COM.

***parameters***

Specifies any parameters to be passed to the application.

The length of the command line cannot exceed 128 characters.

The following example shows a valid command line:

```
wdeb386 /C:1 /R:9600 /F:RUNWDEB.WRF /V \windows\win.com
```



## Starting as a VxD

Running WDEB386 as a VxD requires approximately 90KB less of conventional memory than running it from the command line. Its only disadvantage is that loaded symbols aren't available until after SYS\_CRITICAL\_INIT. You can start the debugger as a virtual device driver by placing the following line in your CONFIG.SYS file:

```
device=c:\windev\wdeb386.exe
```

You must specify the full path to the WDEB386.EXE file. You can specify any command-line options on the **device** line with (for example, you can load symbol files), or you can set options by adding one or more of the following debug settings to the [386Enh] section of SYSTEM.INI:

### **DebugCom=***comport*

Sets the communications port that the debugger uses for input and output. The *comport* can be 1, 2, 3 or 4, as described for the */C* command line option.

### **DebugBaud=***baudrate*

Sets the baud rate for the communications port used by the debugger.

### **DebugCmd=***parameters*

Lists the debugger commands to execute as the debugger starts. This is typically used to set command-line parameters. For example, use DebugCmd=y /n /b to enable new options and to initially break into debugger at SYS\_CRITICAL\_INIT. Multiple commands must be separated by semicolons (;).

### **DebugSym=***symbol-files*

Specifies the symbol file or files to load.

### **DebugSymCmd=***cmds*

Lists the debugger commands to execute immediately after all symbols have been loaded. Multiple commands must be separated by semicolons (;).

### **BreakInDebugVxD=***value*

Sets the behavior of the CTRL+C key when pressed on the debug terminal or the CTRL+ALT+SYSREQ key combination when pressed on the computer on which the debugger is running. If *value* is TRUE, pressing the key stops execution at the current instruction regardless of the privilege (ring) of the code. If FALSE, pressing the keys stops execution in ring 3 code only. The default setting if this entry is not given is TRUE.

If you have the Windows 95 Device Driver Development Kit, you can use the DDKDEBUG.BAT batch file to create a WDEBVXD.INS file that contains the statements that you need to add to the [386Enh] section of SYSTEM.INI. In particular, DDKDEBUG.BAT includes the DEBUGCMD.VXD file that provides additional debugging support. You may need to set the **DebugCom** and other options to appropriate values.

When you no longer want to run the debugger as a VxD, insert a semicolon before the device=WDEB386.EXE (and device=DEBUGCMD.VXD if present) entry in the SYSTEM.INI file. Any debug options settings you added to the file do not need a semicolon; they will be ignored on subsequent loads of Windows 95.



## Using the System Debugger



## Introduction

There are many situations in which the WDEB386 debugger is the best or even the only debugger you should use. For example, you might use WDEB386 to diagnose problems such as the following:

- Tracing through low-level code
- Viewing virtual/linear/physical memory
- Viewing advanced 386 processor data, such as the GDT, LDT, IDT, and all of the PMODE registers
- Tracing hardware interrupt handlers
- Tracing terminate-and-stay-resident (TSR) programs, or MS-DOS device drivers
- Displaying the status of virtual machines (VMs)
- Monitoring all interrupts and exceptions
- Developing and debugging virtual devices (VxDs)

This topic gives information about how to use WDEB386 commands and features.



## Breaking into the Debugger

To execute debugging commands, you need to break into the debugger and have it display its command prompt.

You can automatically break into the debugger as Windows starts by specifying **/B** option on the WDEB386 command line or in a **y** command with the **DebugCmd** setting in the SYSTEM.INI file. The debugger stops execution just after loading VxDs and just prior to initialization.

To break into the debugger at any time interrupts are not disabled, press the CTRL+C key combination on the debugging terminal. Alternately, press the CTRL+ALT+SYSRQ key combination on the computer running the debugger. This stops execution at the next convenient location in ring 0 or ring 3 code.

To break into the debugger when interrupts are disabled, you use hardware to generate a nonmaskable interrupt (NMI). This usually means having an external "STOP" button connected to a debugging card installed in computer running the debugger. Some machines may have the capability of connecting a front panel button to the NMI line on the machine bus. In any case, using NMI has the advantage of being able to break into a machine that has hung with interrupts disabled. (You can disable the breaking on nonmaskable interrupts by using the **v2** command.)

You can have your application, DLL, or VxD break into the debugger by adding an **int 1** or **int 3** instruction or a call to the **DebugBreak** function to your code. The **int 1** instruction produces an "Unexpected trace interrupt" message and stops on the instruction after the **int 1**. This message does not indicate an error condition and can be ignored. An **int 3** will break directly on the **int** instruction and not produce the message. The **int 3** is used in system components to stop execution on an error.

Once an **int** instruction is hit, you can remove it by using the **z** (Zap) command. This command replaces the **int** instruction with a **nop** instruction. For programmers developing virtual device drivers (VxDs), the **Debug\_Out** macro is available to send an ASCII string to the debug terminal and execute an **int 1**, which will break to the debugger.

Once you have broken into the debugger, you can set additional breakpoints by using the **bp** or **br** command. For example, the following command sets a breakpoint at beginning of the function `_MyEntryPoint`:

```
bp _MyEntryPoint
```

The system also breaks into the debugger if an application or DLL causes a general protection fault (GPF) by attempting to read or write memory with a bad selector, beyond a selector limit, or with a selector set to 0. The debugger receives control immediately if it traps interrupt vector 0Dh (the default setting). If you disable this trap (by using the **vs** command), Windows first displays a dialog box notifying the user of a problem. The user can click the Debug button to pass control to the debugger at the instruction that caused the fault.



## Determining the State of the Processor

Once control has been given to the debugger, the prompt character used will provide the protected mode status of the processor. The following list shows what prompt characters may be displayed and the meaning of each:

- > The processor is in real mode
- # The processor is in protected mode
- The processor is in virtual 8086 mode

The mode the processor is in will be a good indication of what code is being executed. For example, if the prompt is a “#” (number sign), protected mode code is running. This can be a Windows-based application, DLL, or even the system itself. Stopping in ring 0 code may or may not be desirable. It is useful for VxD developers who need to examine and control execution of their VxDs, but it is not particularly useful for application or DLL developers.

Regardless of the state of the processor, you can use the following command keys and the debugger prompt:

Key	Action
CTRL+A	Repeats the previous command.
CTRL+C	Cancels the current command.
CTRL+S	Freezes an System Debugger display.
CTRL+Q	Restarts the display.

If the target system is executing code, CTRL+S and CTRL+Q are ignored.



## Setting Breakpoints

You can set breakpoints using the **bp** and **br** commands. With each breakpoint, you can specify breakpoint commands, a string of debugger commands that are executed when a breakpoint is hit. For example, the following command sets a breakpoint that stops execution in the function `_MyEntryPoint` and displays the registers and stack:

```
bp _MyEntryPoint+346 "r;k"
```

Semicolons (;) separate commands from one another. All text is converted to uppercase except for text surrounded by single quotation marks ('). Two single quotation marks (") or two double quotation marks (") in a row act as an escape character and add one single quotation mark or one double quotation mark to the string. The maximum length of a breakpoint command is 80 characters. If the breakpoint has no breakpoint command string, WDEB386 executes the default (**zd**) command.

The conditional execution command (**j**) is very useful in breakpoint commands. This command executes the command list if the expression evaluates to TRUE (nonzero); otherwise, it continues to the next command in the command line (not including the ones in the command list parameter). If the command list contains more than one command, it must be enclosed in single or double quotation marks. Use a semicolon (;) to separate commands. No quotation marks are required if the command list contains zero or one command. The conditional execution command can be used in breakpoint commands to halt execution when an expression becomes true or in the default command.

Any operator, number, or symbol value can be used in the conditional expression. Always put a zero in front of a hexadecimal number that begins with a nonnumeric character. Doing so will prevent the debugger from treating the number as a symbol and searching all the loaded symbol files. For example, using `0f000` is faster than `f000`.

For example, the following command stops execution and display registers only if the variable `_MyVar` is equal to 3 when control enters the `_MyEntryPoint` function. Otherwise, it displays the current value and continues:

```
bp _MyEntryPoint "j _MyVar == 3 r;'? "_MyVar=%x" _MyVar;g'"
```



## Setting the Default Command

The debugger runs the default command string when it reaches any breakpoint that you have set with the **g** (go), **bp** (Breakpoint), or **br** (Breakpoint Register) command. It also runs the default command when you run the **p** (program trace), **t** (trace), or **zd** command.

Initially, the default command string is set to the **r** (Register) command, but you can change it by using the **zs** (Set Default) command. If any errors occur (for example, if the command line is too long), the default command returns to the **r** command. The default command can be any sequence of debugger commands each separated by a semicolon (;). In the default command, **j** commands can be useful.

Default command needed to continue execution each time the application or test program encounters an Interrupt 3:

```
zs "j (by cs:eip) == 0cc 'g'"
```

Trace until the doubleword at 137:00001234h is equal to 0EEDh (a primitive watchpoint). This operation must be started with a **T**, a **P**, or a **ZD** command so that the default command can be executed. If this operation is started by the **G** command, the default command will not execute unless execution is stopped on a go breakpoint or on a sticky breakpoint with no breakpoint command.

```
zs "j (dw 137:00001234) == 0eed 'r';t"
zd
```

Perform a trace that displays each instruction until control is returned to code segment or selector 137h. Notice that **PN** displays only the disassembly line and not the register set, saving line space on the debugger's terminal screen.

```
zs "u cs:eip 11; j cs == 0137 'r';pn"
zd
```



## Reference



## Command Groups

The following is a list of System Debugger commands.

### Help Commands

?  
.?

### Breakpoint and Step Commands

bc  
bd  
be  
bl  
bp  
br  
g  
j  
p  
t  
z

### Register, Memory, and Port Commands

c  
d  
db  
dd  
dw  
e  
f  
i  
m  
o  
r  
s

### Disassemble and Stack Trace Commands

k  
ka  
kt  
u

### Symbol and Symbol Map Commands

la  
lg  
lm  
ln  
ls  
w  
wa  
wr

### Interrupt-Vector Commands

vc  
vl  
vo  
vs  
vt

### 80386 Data Structure Commands

dl



dp  
dt

#### Dot Commands

.b  
.df  
.dg  
.dh  
.dm  
.dq  
.du  
.reboot

#### Miscellaneous Commands

h  
v  
y  
zd  
zl  
zs



## Command Syntax

Commands consist of command names and parameters. Names and parameters are not case-sensitive.



### Parameters



### Binary and Unary Operators



### Regular Expressions

If a syntax error occurs in a debugger command, the debugger redisplay the command line and indicates the error with a caret (^) and the word Error, as in the following example:

```
A100
  ^ Error
```



## Parameters

You can separate command parameters with delimiters (spaces or commas), but a delimiter is required only between two consecutive hexadecimal values. The following commands are equivalent:

```
dCS:100 110
d CS:100 110
d,CS:100,110
```

Following are the parameters you can use with commands:

### *addr*

Represents an address parameter in one of four forms. For more information about the operators shown in the following address forms, see [Binary and Unary Operators](#).

#1f:02C0	Protected-mode address (selector:offset)
%31020	Linear address
%%31020	Physical address
&0100:02FF	Real-mode address (segment:offset)

Any of these specified address forms overrides the current address type.

### *byte*

Specifies a two-digit hexadecimal value.

### *cmds*

Specifies an optional set of debugger commands to be executed with the **bp** (Breakpoint) or **j** (Conditional) command.

### *count*

Specifies a count. Valid values depend on the command with which this parameter is being used.

### *dword*

Represents an eight-digit (4-byte) hexadecimal value. The **DWORD** data type is most commonly used as a physical address.

### *expr*

Represents a combination of parameters and operators that evaluates to an 8-bit, 16-bit, or 32-bit value. An *expr* parameter can be used as a value in any command. An *expr* parameter can combine any symbol, number, or address with any of the binary and unary operators.

### *flags*

Specifies one or more conditions. Valid conditions depend on the command with which this parameter is being used.

### *group-name*

Specifies the name of a group that contains the map symbols you want to display.

### *list*

Specifies a series of byte values or a string. The *list* parameter must be the last parameter on the command line. Following is an example of the **f** (Fill) command with a *list* parameter:

```
fCS:100 42 45 52 54 41
```

### *map-name*

Specifies the name of a symbol map file.

### *name-chars*

Specifies one or more characters.

### *number*

Specifies a numeric value. Valid values depend on the command with which this parameter is being used.

### *object*

Specifies a handle, a selector, or a heap address.

### *option*



Specifies an option. Valid options depend on the command with which this parameter is being used.

*range*

Specifies the block of memory on which the command should operate. The *range* parameter can be two addresses (*addr addr*); or it can be one address and a length (*addr L word*, where *word* is the number of items on which the command should operate; 80h is the default value). Following are three valid examples:

```
CS:100 110
CS:100 L 10
CS:100
```

The limit for *range* is 10000h. To specify a word of 10000h using only four digits, use 0000h or 0h.

*reg*

Specifies the name of a microprocessor register.

*string*

Represents any number of characters enclosed in single quotation marks (') or double quotation marks ("). For quotation marks that must appear within *string*, you must use two sets of quotation marks. For example, the following strings are valid:

```
'This 'string' is OK.'
"This \"string\" is OK.\"
```

However, the following strings are not valid:

```
\"This \"string\" is not OK.\"
\"This 'string' is not OK.\"
```

The ASCII values of the characters in the string are used as a list of byte values.

*word*

Specifies a four-digit (2-byte) hexadecimal value.



## Binary and Unary Operators

Following, in descending order of precedence, are the binary operators that can be used in commands:

Operator	Meaning
( )	Parentheses
:	Address binder
*	Multiplication
/	Integer division
MOD	Modulus (remainder)
+	Addition
–	Subtraction
>	Greater-than relational operator
<	Less-than relational operator
>=	Greater-than/equal-to relational operator
<=	Less-than/equal-to relational operator
==	Equal-to relational operator
!=	Not-equal-to relational operator
AND	Bitwise Boolean AND
XOR	Bitwise Boolean exclusive OR
OR	Bitwise Boolean OR
&&	Logical AND
	Logical OR

Following, in descending order of precedence, are the unary operators that can be used in commands:

Operator	Meaning
&(seg)	Address of segment value
#(sel)	Address of selector value
%%(phy)	Address as a physical value
%(lin)	Address as a linear value
–	Two's complement
!	Logical NOT operator
NOT	One's complement
SEG	Segment address of operand
OFF	Address offset of operand
BY	Low-order byte from given address
WO	Low-order word from given address
DW	Doubleword from given address
POI	Pointer (4 bytes) from given address — this operator works only with 16:16 addresses
PORT	1 byte from given port
WPORT	Word from given port



## Regular Expressions

The set of regular expressions that the debugger supports for matching symbols is similar to the set supported by UNIX grep. The debugger set includes a few enhancements.

Following are the wildcards:

Wildcard	Description
.	Matches any single character.
[ ]	Defines a character class; matches a set or range of characters.
^	Negates a character class.

Following are the postfix operators:

Operator	Description
*	Causes the previous wildcard or single character to match zero or more characters.
#	Matches zero or one.
+	Plus sign, matches one or more.

Anywhere a symbol is accepted, a regular expression can be used. If there is more than one match, a list of matching symbols is displayed and you must select the proper symbol. The symbol match is not case-sensitive.

The asterisk (\*), number sign (#), and plus sign (+) are already math expression operators. To be recognized as a regular expression operator, each of these characters must be immediately preceded by an escape character — the backslash (\). The period (.), opening bracket ([), and closing bracket (]) do not require escape characters. Anything inside the brackets of a character class does not have to be escaped. Following are valid character classes:

```
[a-z]
[; * + #]
```

Characters are escaped at two levels: in the expression evaluator and in the regular expression parser. A character special to the expression evaluator (\*, #, +, or \) must be escaped to make it to the regular expression parser. If a character special to the regular expression parser must be escaped (for example, to match symbols with \* or # in them), it must be escaped twice. If a backslash is needed in an expression, it must be double escaped.

Following are sample regular expressions:

Regular expression	Description
sym.\*	Matches any symbols beginning with the string sym.
sym\*	Matches sym alone and sym followed by any characters.
.\*sym.\*	Matches any symbols containing the string sym.
sym[0–9]	Matches sym0, sym1, sym2, and so on.
sym\\*	Matches sym*.
sym\\\\	Matches sym\.
sym\\\\.\*	Matches any symbols beginning with the string sym\.



## Commands

-  ? (Help)
-  ? (Evaluate Expression)
-  bc (Breakpoint Clear)
-  bd (Breakpoint Disable)
-  be (Breakpoint Enable)
-  bl (Breakpoint List)
-  bp (Breakpoint)
-  br (Breakpoint Register)
-  c (Compare)
-  d (Display)
-  da (Display ASCII)
-  db (Display Bytes)
-  dd (Display Doublewords)
-  dg (Display GDT)
-  di (Display IDT)
-  dl (Display LDT)
-  dp (Display Page Directory and Tables)
-  dt (Display Task State Segment)
-  dw (Display Words)
-  dx (Display Loadall Buffer)
-  e (Enter)
-  f (Fill)
-  g (Go)
-  h (Hex)
-  i (Input)
-  j (Conditional)
-  k (Stack Trace)
-  ka (Stack Trace Arguments)
-  kt (Stack Trace for Task)
-  la (List Absolute Symbols)
-  lg (List Groups)
-  lm (List Maps)
-  ln (List Nearest Symbol)
-  ls (List Symbols)
-  lse (List Symbols by Regular Expression)
-  m (Move)
-  o (Output)
-  p (Program Trace)
-  r (Register)
-  s (Search)
-  t (Trace)
-  u (Disassemble)
-  v (Version Number)
-  vc (Vector Clear)



-  vl (Vector List)
-  vo (Vector List New)
-  vs (Vector Trap Non-Supervisor)
-  vt (Vector Trap)
-  w (Change Map)
-  wa (Activate Map)
-  wr (Deactivate Map)
-  x (Dump Debug Report)
-  y (Debugger Options)
-  z (Zap)
-  zd (Execute Default)
-  zl (Display Default)
-  zs (Set Default)



## **? (Help)**

The ? command with no arguments displays a list of commands and syntax recognized by the debugger.



## ? (Evaluate Expression)

? [*option*.]*expr*

? "*string*", *expr*, ...

The ? command evaluates an expression and displays the result.

*expr*

Expression to evaluate. Can be a combination of numbers, addresses, and operators. Numbers are assumed to be hexadecimal. Addresses can be 32-bit physical addresses or protected-mode addresses (selector:offset). The number sign (#) operator overrides the current address type. Operators can be any listed in Binary and Unary Operators.

*option*

Format in which to display the expression. Can be one of these:

- h.** Hexadecimal
- d.** Decimal
- t.** Decimal
- o.** Octal
- q.** Octal
- y.** Binary

By default, the command displays all formats: decimal, hexadecimal, octal, binary, ASCII, and Boolean.

*string*

Formatting string. Can be a combination of text and zero or more of the following formatting descriptors and escape sequences:

- %% Displays a percent sign (%).
- %A Displays matching *expr* as an address.
- %b Displays matching *expr* in binary format.
- %c Displays matching *expr* as a character.
- %d Displays matching *expr* in decimal format.
- %G Evaluates matching *expr* as an address and displays the group and symbol associated with the address in group:symbol format.
- %M Evaluates matching *expr* as an address and displays the map file, group and symbol associated with the address in map:group:symbol format.
- %o Displays matching *expr* in octal format.
- %S Evaluates matching *expr* as an address and displays the map file, group and symbol associated with the address.
- %s Evaluates matching *expr* as an address and displays the string at that address.
- %u Displays matching *expr* in unsigned decimal format.
- %X Displays matching *expr* in hexadecimal format.
- %x Displays matching *expr* in hexadecimal format
- \a Inserts a bell (alert) character.
- \b Inserts a backspace character.
- \n Inserts a new line character.
- \r Inserts a carriage return character.
- \t Inserts a horizontal tab character.

One expression must be given for each formatting descriptor in the string. Multiple expressions can be separated with commas (,) or spaces.

Formatting descriptors can have these optional prefixes:



A	<code>[-][width][.precision][a][p][n][L][H][N]</code>
b	<code>[-][0][width][.precision][p][n]</code>
d	<code>[-][+][ ][0][width][.precision][p][n]</code>
G	<code>[-][width][.precision][a][p][n][L][H][N]</code>
M	<code>[-][width][.precision][a][p][n][L][H][N]</code>
o	<code>[-][0][width][.precision][p][n]</code>
s	<code>[-][width][.precision][a]</code>
S	<code>[-][width][.precision][a][p][n][L][H][N]</code>
u	<code>[-][0][width][.precision][p][n]</code>
X	<code>[-][#][0][width][.precision][p][n]</code>
x	<code>[-][#][0][width][.precision][p][n]</code>

Specifying an asterisk (\*) for the *width* or *precision* parameter causes the field width or precision, respectively, to be picked up from the next parameter. Decimal values can also be specified for the *width* and *precision* parameters. The prefix letters have these meanings:

a	Address argument size
H	Display 16-bit offset
L	Display 32-bit offset
N	Display offset only
p	Get the previous symbol, symbol address, or offset
n	Get the next symbol, symbol address, or offset

The following examples show simple commands and corresponding output:

```
? ds:esi
013f:000001B3 %00098953 %%00098953
```

```
// display the value of the arithmetic expression 3*4:
? 3*4
0Ch 12T 14Q 00001100Y '.' TRUE
```



## **bc (Breakpoint Clear)**

**bc** *list* | \*

The **bc** command removes one or more defined breakpoints.

*list*

Specifies any combination of integer values in the range 0 through 9. If you specify *list*, the debugger removes the specified breakpoints.

\*

Clears all breakpoints.

This example removes breakpoints 0, 4, and 8:

```
bc 0 4 8
```



## **bd (Breakpoint Disable)**

**bd** *list* | \*

The **bd** command temporarily disables one or more breakpoints. To restore breakpoints disabled by the **bd** command, use the **be** (Breakpoint Enable) command.

*list*

Specifies any combination of integer values in the range 0 through 9. If you specify *list*, the debugger disables the specified breakpoints.

\*

Disables all breakpoints.

This example disables breakpoints 0, 4, and 8:

```
bd 0 4 8
```



## **be (Breakpoint Enable)**

**be** *list* | \*

The **be** command restores (enables) one or more breakpoints that have been temporarily disabled by a **bd** (Breakpoint Disable) command.

*list*

Specifies any combination of integer values in the range 0 through 9. If you specify *list*, the debugger enables the specified breakpoints.

\*

Enables all breakpoints.

This example enables breakpoints 0, 4, and 8:

```
be 0 4 8
```



## **bl (Breakpoint List)**

**bl**

The **bl** command lists current information about all breakpoints created by the **bp** (Breakpoint) command. For each existing breakpoint, the command displays the breakpoint number, the enabled status ('e' for enabled, 'd' for disabled, 'i' for invalid), the breakpoint address, the number of passes remaining (if any), the initial number of passes in parentheses (if any), and debugger commands to be executed when the breakpoint is reached (if any). The following example shows a typical list:

```
0 d %004010b5 [_MyTest] 4 (10) "db ds:edi"
1 eI %0040110f [_MyWndProc@16 + 5a]
```



## bp (Breakpoint)

**bp**[*number*]*addr* [*count*] ["*cmds*"]

The **bp** command creates a software breakpoint at an address. When the application is running, software breakpoints stop execution and force the debugger to execute the default or optional command string. Unlike breakpoints created by the **g** (Go) command, software breakpoints remain in memory until you remove them with the **bc** (Breakpoint Clear) command or temporarily disable them with the **bd** (Breakpoint Disable) command.

### *number*

Specifies which breakpoint is being created. No space is allowed between the **bp** and *number*. If *number* is omitted, the first available breakpoint number is used. The debugger allows up to 10 software breakpoints (0 through 9). If you specify more than 10 breakpoints, the debugger returns the message: "Too Many Breakpoints."

### *addr*

Specifies any valid instruction address — the first byte of an operation code (opcode). The *addr* parameter is required for all new breakpoints.

### *count*

Specifies the number of times the breakpoint is to be ignored before being executed. It can be any 16-bit value.

### *cmds*

Specifies an optional list of debugger commands to be executed in place of the default command when the breakpoint is reached. You must enclose optional commands in quotation marks and separate optional commands with semicolons (;).

This example creates a breakpoint at address CS:401000:

```
bp 401000
```

This example creates breakpoint 8 at address given by the symbol `_MyTest`. When the breakpoint occurs, the debugger displays bytes at DS:SI:

```
bp8 _MyTest "db DS:SI"
```



## **br (Breakpoint Register)**

**br**[number] flags [count] ["cmds"]

The **br** command sets a debug register breakpoint. Debug registers can be used to break on data reads and writes and instruction execution. Up to four debug registers can be set and enabled at one time.

### *number*

Specifies which breakpoint is being created. No space is allowed between the **br** command and the *number* parameter. If *number* is omitted, the first available breakpoint number is used.

### *flags*

Specifies the length and break conditions for the breakpoint. This parameter can be some combination of the following values:

- |          |   |
|----------|---|
| <b>1</b> | Set 1-byte length (default value).                        |
| <b>2</b> | Set word length on word boundary.                         |
| <b>4</b> | Set doubleword length on doubleword boundary.             |
| <b>E</b> | Break on instruction execution only (1-byte length only). |
| <b>W</b> | Break on writes only.                                     |
| <b>R</b> | Break on reads and writes.                                |

### *count*

Specifies the number of times the breakpoint is to be ignored before being executed. It can be any 16-bit value.

### *cmds*

Specifies an optional list of debugger commands to be executed in place of the default command when the breakpoint is reached. You must enclose the group of optional commands in quotation marks and separate optional commands with semicolons (;).



## **c (Compare)**

**c** *range addr*

The **c** command compares one memory location with another memory location. If the two memory areas are identical, the debugger displays nothing and returns the debugger prompt. Differences, when they exist, are displayed in this form: *addr1 byte1 byte2 addr2*.

*range*

Specifies the block of memory that is to be compared with a block of memory starting at *addr*.

*addr*

Specifies the starting address of the second block of memory.

This example compares the bytes at addresses in the range 100h to 1FFh with the corresponding bytes at address from 300h to 3FFh:

```
c100 1FF 300
```

This example compares the same block of memory as the previous example but specifies the *range* by using the **L** (length) option.

```
c100 L 100 300
```



## **d (Display)**

**d** [*range*]

The **d** command displays the contents of memory at a given address or in a range of addresses. The **d** command displays one or more lines, depending on the *range* given. Each line displays the address of the first item displayed. The command always displays at least one value. The memory display is in the format defined by a previously executed **da**, **db**, **dd**, or **dw** command. Each subsequent **d** (typed without parameters) displays the bytes immediately following those last displayed.

*range*

Specifies the block of memory to display. If you omit *range*, the **d** command displays the next byte of memory after the last one displayed. The **d** command must be separated by at least one space from any *range* value.



## **da (Display ASCII)**

**da** [*range*]

The **da** command displays as ASCII characters the values of the bytes at a given address or in a given range. The display includes all bytes up to the first zero byte. Nonprinting characters and characters having values greater than 127 are denoted by a period (.).



## **db (Display Bytes)**

**db** [*range*]

The **db** command displays the values of the bytes at a given address or in a given range.

The display is in two portions: a hexadecimal display (each byte is shown in hexadecimal format) and an ASCII display (the bytes are shown as ASCII characters). A nonprinting character is denoted by a period (.) in the ASCII portion of the display. Each display line shows 16 bytes, with a hyphen between the eighth and ninth bytes. Each displayed line begins on a 16-byte boundary.

*range*

Specifies the block of memory to display. If you omit *range*, 128 bytes are displayed beginning at the first address after the address displayed by the previous **db** command.

The following example displays 0Ah bytes of memory, beginning at the specified address:

```
db CS:100 0A
```

This example displays lines in a format similar to the following:

```
04BA:00000100 54 4F 4D 20 53 . . . 45 52 TOM SAWYER
```

Each line of the display begins with an address, incremented by 10h from the address on the previous line.



## **dd (Display Doublewords)**

**dd** [*range*]

The **dd** command displays the hexadecimal values of the doublewords at the address specified or in the specified range of addresses. The **dd** command displays one or more lines, depending on the *range* given. Each line displays the address of the first doubleword in the line, followed by up to four hexadecimal doubleword values. The hexadecimal values are separated by spaces. The **dd** command displays values up to the end of the *range* or until the first 32 doublewords have been displayed.

*range*

Specifies the block of memory to display. If you omit *range*, 32 doubleword values are displayed beginning at the first address after the address displayed by the previous **dd** command.

The following example displays the doubleword values from CS:100 to CS:110:

```
dd CS:100 110
04BA:0100 7473:2041 676E:6972 5405:0104 0A0D:7865
04BA:0110 0000:002E
```

No more than four values per line are displayed.



## **dg (Display GDT)**

**dg[a]** [*range*]

The **dg** command displays the specified range of entries in the global descriptor table (GDT).

**a**

Displays all entries in the table, not just the valid ones. By default, only the valid GDT entries are displayed. If *range* specifies a local descriptor table (LDT) selector, the command displays the appropriate LDT entry.

*range*

Specifies the range of entries to be displayed. If you omit *range*, the entire table is displayed.

The command displays the selector, descriptor type, base address, limit, privilege, and other descriptor flags.



## **di (Display IDT)**

**di**[**a**] [*range*]

The **di** command displays the specified range of entries in the interrupt descriptor table (IDT).

**a**

Displays all entries in the table, not just the valid ones.

*range*

Specifies the range of entries to be displayed. If you omit *range*, the entire table is displayed.

The command displays the interrupt number, the interrupt type, the interrupt address, privilege, and descriptor flags.



## **dl (Display LDT)**

**dl**[a | p | s | h] [*range*]

The **dl** command displays the specified range of entries in the local descriptor table (LDT).

### **a**

Displays all entries in the table, not just the valid ones. By default, only the valid LDT entries are displayed. If *range* specifies a global descriptor table (GDT) selector, it displays the appropriate GDT entry.

### **p**

Displays private segment selectors.

### **s**

Displays shared segment selectors.

### **h**

Displays huge segment selectors. To display the huge segment selectors, give the shadow selector followed by the maximum number of selectors reserved for that segment plus 1.

### *range*

Specifies the range of entries to be displayed. If you omit *range*, the entire table is displayed.

The command displays the selector, descriptor type, base address, limit, privilege, and other descriptor flags.



## dp (Display Page Directory and Tables)

**dp[a|d] [range]**

The **dp** command displays the page directory and page tables. Page tables are always skipped if the corresponding page directory entry is not present. Page directory entries appear with an asterisk next to the page frame.

**a**

Displays all present page directory and page table entries; by default, page directory and page table entries that are zero are skipped.

**d**

Displays only page directory entries. If a count is given as part of the optional range, it will be interpreted as a page directory entry count.

*range*

Specifies the range of linear addresses for page tables.

The command displays the entry address, frame, and entry flags. Page directory entries are marked with an asterisk (\*). Entry flags have the following meanings:

A	Accessed
c	clean
CD	Cache Disable
D	Dirty
n	Not-present
P	Present
r	Read-only
s	Supervisor
U	User
u	Unaccessed
W	Writable
WT	Write Through



## **dt (Display Task State Segment)**

**dt** [*addr*]

The **dt** command displays the current task state segment (TSS) or the selected TSS if you specify the optional address.

*addr*

Specifies the address of the TSS to display. If no *addr* is given, **dt** displays the current TSS pointed to by the TR register.



## **dw (Display Words)**

**dw** [*range*]

The **dw** command displays the hexadecimal values of the words at a given address or in a given range of addresses. The command displays one or more lines, depending on the *range* given. Each line displays the address of the first word in the line, followed by up to eight hexadecimal word values. The hexadecimal values are separated by spaces. The command displays values until the end of the *range* or until the first 64 words have been displayed.

*range*

Specifies the range of addresses to display. If you omit *range*, 64 words are displayed beginning at the first address after the address displayed by the previous **dw** command.

The following example displays the word values from CS:100 to CS:110:

```
dw CS:100 110
04BA:0100 2041 7473 6972 676E 0104 5404 7865 0A0D
04BA:0110 002E
```



## **dx (Display Loadall Buffer)**

**dx**

Displays the contents of the loadall buffer.



## **e (Enter)**

**e** *addr* [*list*]

The **e** command enters byte values into memory at a specified address. You can specify the new values on the command line or let the debugger prompt you for values.

*addr*

Specifies the address of the first byte to be entered.

*list*

Specifies the byte values used for replacement. These values are inserted automatically. If an error occurs when you are using the list form of the command, no byte values are changed.

If the debugger prompts you, it displays the address and its contents and then waits for you to perform one of the following actions:

- Replace a byte value with a value you type. Type the value after the current value. If the byte you type is an invalid hexadecimal value or contains more than two digits, the system does not echo the illegal or extra character.
- Press the SPACEBAR to advance to the next byte. To change the value, type the new value after the current value. If, when you press the SPACEBAR, you move beyond an 8-byte boundary, the debugger starts a new display line with the address displayed at the beginning.
- Type a hyphen (-) to return to the preceding byte. If you decide to change a byte before the current position, typing the hyphen returns the current position to the previous byte. When you type the hyphen, a new line is started with its address and byte value displayed.
- Press ENTER to terminate the **e** command. You can press ENTER at any byte position.

The following example prompts you to change the value EB at CS:100:

```
eCS:100
04BA:0100 EB.
```

To step through the subsequent bytes without changing values, press the SPACEBAR. In the following example, the SPACEBAR is pressed three times:

```
04BA:0100 EB.41 10. 00. BC.
```

To return to a value at a previous address, type a hyphen, as shown in the following example:

```
04BA:0100 EB.41 10. 00. BC.-
04BA:0102 00.-
04BA:0101 10.
```

This example returns to the address CS:101.



## **f (Fill)**

**f** *range list*

The **f** command fills the addresses in a specified range with the values in the specified list.

*range*

Specifies the block of memory to be filled. If *range* contains more bytes than the number of values in *list*, the debugger uses *list* repeatedly until all bytes in *range* are filled. If any of the memory in *range* is not valid (bad or nonexistent), an error occurs in all succeeding locations.

*list*

Specifies the list of values to fill the given *range*. If *list* contains more values than the number of bytes in *range*, the debugger ignores the extra values in *list*.

The following example fills memory locations 04BA:100 through 04BA:1FF with the bytes specified, repeating the five values until it has filled all 100h bytes:

```
f04BA:100 L 100 42 45 52 54 41
```



## g (Go)

**g[s|h|t|z] [=addr [addr[...]] ]**

The **g** command executes the application currently in memory. If you type the **g** command by itself, the current application runs as if it had been run outside the debugger. If you specify **=addr**, execution begins at the specified address.

### s

Shows the time, in microseconds, from when the system is started with **gs** until the next entry to the debugger. No attempt is made to calculate and remove debugger overhead from the measurement. Requires a timing card.

### h

Displays the approximate debugger overhead in the **s** option. Requires a timing card.

### t or z

Allows trapped exceptions to resume at the original trap handler address without having to unhook the exception. Use these options instead of the **vcp d; t; vsp d** command.

### =addr

Specifies the address at which execution is to begin. The equal sign (=) is needed to distinguish the starting address from the breakpoint address.

### addr

Specifies one or more breakpoint addresses where execution is to halt. You can specify up to 10 breakpoints, but only at addresses containing the first byte of an operation code (opcode). If you attempt to set more than 10 breakpoints, an error message is displayed.

Specifying an optional breakpoint address causes execution to halt at the first address encountered, regardless of the position of the address in the list of addresses that halts execution or application branching. When execution of the application reaches a breakpoint, the default command string is executed.

The stack (SS:SP) must be valid and have 6 bytes available for this command. The **g** command uses an **iret** instruction to cause a jump to the application being tested. The stack is set, and the user flags, CS register, and IP register are pushed on the user stack. (If the user stack is not valid or is too small, the operating system may crash.) An interrupt code (0CCh) is placed at the specified breakpoint addresses.

When the debugger encounters an instruction with the breakpoint code, it restores all breakpoint addresses listed with the **g** command to their original instructions. If you do not halt execution at one of the breakpoints, the interrupt codes are not replaced with the original instructions.

The following example executes the application currently in memory until address 7550 in the CS selector is executed. The debugger then executes the default command string, removes the **int 3** trap from this address, and restores the original instruction. When you resume execution, the original instruction is executed.

```
gCS:7550
```



## **h (Hex)**

**h** word word

The **h** command performs hexadecimal arithmetic on the two specified parameters. The debugger adds, subtracts, and multiplies the two parameters; divides the second parameter by the first; and then displays the results on one line. The debugger does 32-bit multiplication and displays the result as doublewords. The debugger displays the result of division as a 16-bit quotient and a 16-bit remainder.

*word*

Specifies a 16-bit word parameter.



## **i (Input)**

**i** word

The **i** command accepts and displays 1 byte from a specified port.

*word*

Specifies the 16-bit port address.

The following example displays the byte at port address 2F8h:

```
i2F8
```



## j (Conditional)

**j** *expr* [*cmds-if-true*][:*cmds-if-false*]

The **j** command executes selected commands based on whether the specified expression is TRUE or FALSE. The **j** command is useful in breakpoint commands to conditionally break execution when an expression becomes TRUE.

*expr*

Evaluates to a Boolean TRUE or FALSE.

*cmds-if-true*

Specifies a list of debugger commands to be executed when *expr* is TRUE. More than one command can be given, but the commands must be separated by semicolons and the complete list must be enclosed in single or double quotation marks.

*cmds-if-false*

Specifies a list of debugger commands to be executed when *expr* is FALSE. The preceding semicolon is required. More than one command can be given, but the commands must be separated by semicolons and the complete list must be enclosed in single or double quotation marks.

The following example causes execution to continue if EAX equals zero when the breakpoint is reached:

```
bp 167:1454 "J EAX == 0 G"
```

The following example displays the registers and continues execution when the byte pointed to by DS:SI +3 is equal to 40h; otherwise, it displays the descriptor table:

```
bp 167:1462 "J BY (DS:SI+3) == 40 'R;G';DG DS"
```



## **k (Stack Trace)**

**k**[**b**|**s**|**v**] [*addr*] [*addr*]

This command displays the current stack frame. Each line shows the name of a procedure, its arguments, and the address of the statement that called it. The command displays four 2-byte arguments by default. The **ka** command changes the number of arguments displayed by this command.

### **b**

Indicates the stack frame is 32 bits wide.

### **s**

Indicates the stack frame is 16 bits wide.

### **v**

Displays the verbose version of stack information — that is, information about stack location and frame pointer values for each frame.

### *addr*

Specifies an optional stack-frame address (SS:EBP) or an optional code address (CS:EIP).

Using the **k** command at the beginning of a function (before the function prolog has been executed) gives incorrect results. The command uses the BP register to compute the current backtrace, and this register is not correctly set for a function until its prolog has been executed.



## **ka (Stack Trace Arguments)**

**ka** *count*

The **ka** command sets the number of arguments displayed for all subsequent stack trace commands. The initial default value is 4.

*count*

Specifies the number of arguments to be displayed. The *count* parameter must be in the range 0 through 1Fh.



## kt (Stack Trace for Task)

**k**[**b**|**s**|**v**]**t** [*addr*]

This command displays the stack frame of the current task or the task specified by the *addr* parameter. Each line shows the name of a procedure, its arguments, and the address of the statement that called it. The command displays four 2-byte arguments by default. The **ka** command changes the number of arguments displayed by this command.

**b**

Indicates the stack frame is 32 bits wide.

**s**

Indicates the stack frame is 16 bits wide.

**v**

Displays the verbose version of stack information — that is, information about stack location and frame pointer values for each frame.

*addr*

Specifies the segment address of the process descriptor block (PDB) for the task to be traced. To obtain the *addr* value, use the **.dq** (Dump Task Queue) command. If *addr* is not supplied, the **kt** command displays the stack frame of the current task.



## **la (List Absolute Symbols)**

### **la**

The **la** command lists the absolute symbols in the active map.



## **lg (List Groups)**

### **lg**

The **lg** command lists the address and the name of each group in the active map.



## **lm (List Maps)**

### **lm**

The **lm** command lists the symbol files currently loaded and indicates which one is active. The last symbol file loaded is made active by default. Use the **w** (Change Map) command to change the active file.



## **In (List Nearest Symbol)**

**In** [*addr*]

The **In** command lists the symbol nearest the specified address. The command lists the nearest symbol before and after the specified *addr* parameter. This command also shows line-number information if it is available in the symbol file.

*addr*

Specifies any valid instruction address. The default value is the current disassembly address.

The **In** command without the *addr* parameter displays the nearest symbols before and after the current disassembly address.



## **Is (List Symbols)**

**Is** *group-name* | *name-chars* | \*

The **Is** command lists the symbols in the specified group or lists names that match the search specification in all groups. The only valid wildcard is a single asterisk (\*) as the last character on the command line; all other characters are ignored.

*group-name*

Names the group that contains the symbols you want to list.

*name-chars*

Specifies the beginning characters of the symbols you want to list.



## **Ise (List Symbols by Regular Expression)**

**Ise** *regular-expression*

The **Ise** command lists the symbols specified by the given regular expression.

*regular-expression*

Can be a regular expression as described in [Command Syntax](#).



## **m (Move)**

**m** *range addr*

The **m** command moves a block of memory from one memory location to another. If part of the destination block overlaps some of the source block, the move is always performed without loss of data. Addresses that could be overwritten are moved first.

*range*

Specifies the block of memory to be moved.

*addr*

Specifies the starting address at which the memory is to be relocated.

For moves from higher to lower addresses, the sequence of events is first to move the data at the block's lowest address and then to work toward the highest. For moves from lower to higher addresses, the sequence is first to move the data at the block's highest address and then to work toward the lowest.

The following example first moves the data at address CS:110 to CS:510 and then moves the data at CS:10F to CS:50F, and so on, until the data at CS:100 is moved to CS:500:

```
mCS:100 110 CS:500
```



## **o (Output)**

**o** word byte

The **o** command writes a byte to a 16-bit port address.

*word*

Specifies the 16-bit port address to be written to.

*byte*

Specifies the 8-bit value to be written to the port.

The following example writes the byte value 4Fh to output port 2F8h:

o 2F8 4F



## **p (Program Trace)**

**p**[*n|t|z*] [=*addr*][*count*]

The **p** command executes the instruction at a specified address and displays the current values of all the registers and flags (whatever the **zd** command has been set to). It then executes the default command string, if any. The **p** command is identical to the **t** (Trace Instructions) command, except that it automatically executes and returns from any calls or software interrupts it encounters. The **t** command always stops after executing into the call or interrupt, leaving execution control inside the called routine.

### **n**

Suppresses the register display so just the assembly line is displayed. The suppression results only if the default command, **zd**, is set to a normal setting, **r**.

### **t** or **z**

Allows trapped exceptions to resume at the original trap handler address without having to unhook the exception.

### *addr*

Specifies the starting address at which to begin execution. If you omit the optional *addr* parameter, execution begins at the instruction pointed to by the CS and IP registers. Use the equal sign (=) only if you specify *addr*.

### *count*

Specifies the number of instructions to execute before stopping and executing the default command string. The command executes the default command string for each instruction before executing the next.

The following example executes the instruction pointed to by the current CS and IP register values before it executes the default command string:

**p**

The following example executes the instruction at address CS:120 before it executes the default command string:

**p=120**



## r (Register)

**r[t][reg[=value]]**

The **r** command displays the contents of one or more central processing unit (CPU) registers and allows the contents to be changed to new values. If you specify the *reg* parameter with the **r** command, the command displays the value of that register in hexadecimal format and prompts for a new value. If you specify a *reg* and a *value*, the command sets the register to the given value.

**t**

Displays registers in terse format.

*reg*

Specifies the register to be displayed. If you specify **f** or **msw** for *reg*, the debugger displays the flags in a row at the beginning of a new line and prompts for one or more flag values. If you omit *reg*, the debugger displays the contents of all registers and flags along with the next executable instruction.

*value*

Specifies the new value for the register. Can be a number or a combination of these flag values:

OV	Overflow set
NV	Overflow clear
DN	Direction decrement
UP	Direction increment
EI	Interrupt enabled
DI	Interrupt disabled
NG	Sign negative
PL	Sign positive
ZR	Zero set
NZ	Zero clear
AC	Auxiliary carry set
NA	Auxiliary carry clear
PE	Parity even
PO	Parity odd
CY	Carry set
NC	Carry clear
NT	Nested task switch (on and off)
TS	Sets the task switch bit. (MSW only)
EM	Sets the emulation processor extension bit. (MSW only)
MP	Sets the monitor processor extension bit. (MSW only)
PM	Sets the protected-mode bit. (MSW only)

Flag values can be in any order. You do not have to leave spaces between these values.

If you type more than one value for a flag or enter an invalid flag name, the flags up to the error in the list are changed and those flags at and after the error are not changed. In addition, the debugger display the message: "Bad Flag."

Setting the protected-mode bit from within the debugger does *not* set the target system to run in protected mode. The debugger simulates the setting. To configure the target system to run in protected mode, you would have to set the PM bit in the MSW register and reset the target system to restart in protected mode.



## **s (Search)**

**s** *range list* | "*string*"

The **s** command searches an address range for a specified list of bytes or an ASCII character string. You can include one or more bytes in *list*, but multiple bytes must be separated by a space or comma. When you search for more than one byte, the command returns the address of only the first byte in the string. When *list* contains only one byte, the debugger displays the addresses of all occurrences of the byte in *range*.

*range*

Specifies the block of memory to be searched.

*list*

Specifies one or more byte values to search for.

*string*

Specifies an ASCII character string to be searched for. The string must be enclosed in quotation marks.

The following example searches for byte 41h in the address range CS:100 to CS:110:

```
sCS:100 110 41
04BA:0104
04BA:010D
```



## **t (Trace)**

**t[a|c|n|s|x|z][=start\_addr][count][addr]**

The **t** command executes one or more instructions along with the default command string and then displays the decoded instruction. If you include the *start\_addr* parameter, tracing starts at the specified address. Otherwise, the command steps through the next machine instruction and then executes the default command string. The **t** command uses the hardware trace mode of the Intel microprocessor. Consequently, you can also trace instructions stored in read-only memory (ROM).

### **a**

Indicates that an ending address is specified for the trace. Instructions are traced until the address in *addr* is reached.

### **c**

Suppresses all output and counts instructions traced. An ending address is required for this command. Instructions are traced until the address in *addr* is reached.

### **n**

Suppresses the register display so just the assembly line is displayed. This works only if the default command, **zd**, is set to **r** (the normal setting).

### **s**

Suppresses output; the instruction and count are displayed for each **call** and the return from that call.

### **x**

Forces the debugger to trace regions of code known to be untraceable (`_PGSwitchContext`, for example).

### **z**

Allows original trap handler address to be traced into without having to unhook the exception. Use this option instead of **vcp d; t; vsp d**.

*start\_addr*

Specifies the instruction address at which to start tracing. The equal sign (=) is required.

*count*

Specifies the number of instructions to execute and trace.

*addr*

Specifies the instruction address at which to stop tracing.

The following command causes the debugger to execute 16 (10h) instructions beginning at 011A in the current selector:

```
t=011A 10
```

The debugger executes and displays the results of the default command string for each instruction. The display is scrolled until the last instruction is executed. Press the CTRL+S key combination to stop the scrolling and CTRL+Q to resume.



## **u (Disassemble)**

**u** [*range*]

The **u** command disassembles bytes and displays the source statements, with addresses and byte values, that correspond to them. The display of disassembled code looks similar to a code listing for an assembled file. If you type the **u** command by itself, 20h bytes are disassembled at the first address after the one displayed by the previous **u** command.

*range*

Specifies the block of memory in which instructions are to be disassembled. If no *range* is given, the command disassembles the next 20h bytes.



## **v (Version Number)**

**v**

The **v** command displays the current debugger version number and date.



## **vc (Vector Clear)**

**vc**[**n** | **p** | **r** | **v**] *number*[,*number* [...]]

The **vc** command clears the specified interrupt vector and reinstalls the previous interrupt vector.

**n**

Removes the beep from traps that beep when encountered; does not clear the traps.

**p**

Clears protected-mode vectors only.

**r**

Clears real-mode vectors only.

**v**

Clears virtual 8086 (V86) mode vectors only.

*number*

Specifies the interrupt vector to clear.



## **vl (Vector List)**

**vl[n | p | r | v]**

Lists the interrupt vectors that the debugger intercepts. Vectors that have been set with the **vt** command (as opposed to **vs**) are listed with an asterisk (\*) following the vector number.

**n**

Lists the traps that beep when encountered.

**p**

Lists the protected-mode vectors only.

**r**

Lists the real-mode vectors only.

**v**

Lists the virtual 8086 (V86) mode vectors only.



## **vo (Vector List New)**

**vo**[n | p | r | v]

The **vo** command lists interrupt vectors in the display format based on the **newvec** option. For details, see the **y** command.

**n**

Lists the traps that beep when encountered.

**p**

Lists the protected-mode vectors only.

**r**

Lists the real-mode vectors only.

**v**

Lists the virtual 8086 (V86) mode vectors only.



## **vs (Vector Trap Non-Supervisor)**

**vs**[**n** | **p** | **r** | **v**] *number*[,*number*[,...]]

The **vs** command adds a new interrupt vector to the list of intercepted vectors. Vectors set by this command do not intercept interrupts that occur at ring 0.

**n**

Beep when trap encountered.

**p**

Set trap for protected-mode vectors only.

**r**

Set trap for real-mode vectors only.

**v**

Set trap for virtual 8086 (V86) mode vectors only.

*number*

Specifies the interrupt vector to intercept.



## **vt (Vector Trap)**

**vt**[**n** | **p** | **r** | **v**] *number* [, *number* [, ...]]

The **vt** command adds a new interrupt vector to the list of intercepted vectors.

**n**

Beep when trap encountered.

**p**

Set trap for protected-mode vectors only.

**r**

Set trap for real-mode vectors only.

**v**

Set trap for virtual 8086 (V86) mode vectors only.

*number*

Specifies the interrupt vector to intercept.



## **w (Change Map)**

**w** [*map-name*]

The **w** command changes the active map file.

*map-name*

Specifies the name of the map file you want to make active. Use the **lm** (List Map) command to display a list of available map files.

If *map-name* is not specified, the loaded maps are displayed and the user is prompted to select a map by pressing its corresponding number.



## **wa (Activate Map)**

**wa** *map-name*

The **wa** command adds the specified map to the list of active maps.

*map-name*

Specifies the map to add to the list of active maps.



## **wr (Deactivate Map)**

**wr** *map-name*

The **wr** command removes the specified map from the list of active maps.

*map-name*

Specifies the map to remove from the list of active maps.



## **x (Dump Debug Report)**

**x**

Dumps a listing of the current execution environment.



## y (Debugger Options)

y[? | *option*]

The **y** command changes the debugger configuration. The following list describes the available configuration options. All settings are toggles.

**?**

Displays a list of supported options.

*option*

Following are the available configuration options:

**/a**

Controls automatic symbol loading. If this option is set, Windows will not load symbols automatically.

**/n**

Sets the following options:

**codebytes**  
**dislwr**  
**int3line**  
**newprompt**  
**newreg**  
**newvec**  
**symaddrs**

**/v**

Controls segment load notification messages. If this option is set, all segment load notifications will be displayed.

**386env**

Controls the size of addresses, registers, and so on when displayed. When this option is on, addresses, registers, and so on are shown in 32-bit format; otherwise, they are shown in 16-bit format.

**codebytes**

Causes code bytes to be displayed along with disassembled instructions.

**disaddr**

Causes addresses to be displayed with disassembled instructions.

**disline**

Causes filenames and line numbers to be displayed with disassembled instructions.

**dislwr**

Causes register and instruction names to be displayed in lowercase letters.

**int3line**

Causes the filename and line number to be displayed with **int 3** instructions.

**newprompt**

Causes a double prompt when paging is enabled and a nesting level if the debugger is reentered.

**newreg**

Controls the format of the register display.

**newvec**

Controls the display format for the intercepted interrupt vectors.

**regterse**

Controls the number of registers displayed by the **r** (Register) command. When **regterse** is on, only the first three lines are displayed (instead of the normal six lines plus disassembly line).

**scrncols**

Sets the number of screen columns in the debug display. The default is 79 columns.

**scrnlines**

Sets the number of screen lines in the debug display. The default is 24 lines.

**skipint3s**



Causes the debugger to ignore inline **int 3** instructions.

**symaddrs**

Causes symbol values to be displayed with the symbols.

**teftibase**

Sets the base port address for the timing card.



## **z (Zap)**

**z**

Replaces the instruction bytes of the current **int 3** instruction or the previous **int 1** instruction with **nop** instructions. This allows the user to avoid **int 1** or **int 3** instructions that were assembled into the executable file by breaking into the debugger more than once.



## **zd (Execute Default)**

zd

The **zd** command executes the default command string. The default command string is initially set to the **r** (Register) command by the debugger. The default command string is executed every time a breakpoint is encountered during execution of the application or whenever a **p** (Program Trace) or **t** (Trace) command is executed.

Use the **zl** command to display the default command string and the **zs** command to change the default command string.



## **zl (Display Default)**

zl

The **zl** command displays the default command string.



## **zs (Set Default)**

**zs** "*string*"

The **zs** command makes it possible for you to change the default command string.

*string*

Specifies the new default command string. The string must be enclosed in single or double quotation marks. You must separate the debugger commands within the string with semicolons.

The following example changes the current default command string to an **r** (Display Register) command followed by a **c** (Compare Memory) command:

```
zs "r;c100 L 100 300"
```

The following example begins execution whenever an **int 3** instruction is executed in your test application. This example executes a **g** (Go) command every time an **int 3** instruction is executed.

```
zs "j (by cs:ip) == cc 'g'"
```

You can use **zs** as follows to set up a watchpoint:

```
zs "j (wo 40:1234) == 0eeed;t"
```

This command traces until the word at 40:1234 is *not* equal to 0EEED. This does not work if you are tracing through the mode switching code in MS-DOS or other sections of code that cannot be traced.



## Dot Commands

-  .<dev\_name> (Dot Device)
-  .? (Help)
-  .b (Baud Rate)
-  .df (Display Free Memory)
-  .dg (Display Global Memory)
-  .dh (Display Local Heap)
-  .dm (Display Modules)
-  .dq (Display Task Queues)
-  .ds (Dump Stack)
-  .du (Display Least Recently Used)
-  .m (Memory)
-  .reboot (System Restart)
-  .v (Virtual Machine Commands)
-  .vmm (Display VMM Commands)
-  .w (Display Win32 Object)



## **.<dev\_name> (Dot Device)**

*.dev\_name*

Directs the VMM to send a Debug\_Query message to the given virtual device. If the virtual device supports a debugging interface, it displays information about its operating state.

*dev\_name*

Can be the name of any virtual device.

Because virtual devices are not required to support the Debug\_Query message, many do not produce debugging output. If a virtual device does generate output, it is typically device specific and not normally useful in a typical debugging situation.

Examine the sample virtual devices to determine how to use the **Trace\_Out** macro, and the **In\_Debug\_Chr** service to generate useful debugging information.



## **.? (Help)**

. ?

The .? command displays a list of external commands. These commands are part of debugger, but they are specific to the environment in which the debugger is running.



## **.b (Baud Rate)**

**.b** *number* [*addr*]

The **.b** command sets the baud rate for the debugging port (COM2).

*number*

Specifies the baud rate. It can be one of the following values: 150, 300, 600, 1200, 2400, 4800, 9600, or 19200. Because the default radix for the debugger is 16, you must type **t** after the number to indicate a decimal value.

*addr*

Specifies 1 for COM1 or 2 for COM2; anything else is taken as a base port address. If there is no COM2, the debugger checks for COM1 and then for any other COM port address in the read-only memory (ROM) data area to use as the console.

The following example sets the baud rate to 1200:

```
#.b 1200t
```



## **.df (Display Free Memory)**

### **.df**

The **.df** command displays a list of the free global memory objects in the global heap. The list has the following form:

*address: size owner [chain]*

#### *address*

Specifies physical and heap addresses.

#### *size*

Specifies the size of the object, in bytes.

#### *owner*

Always specifies that the module is free.

#### *chain*

Specifies the previous and next addresses in the list of least recently used (LRU) objects. The debugger displays the addresses only if the segment is movable and discardable.



## **.dg (Display Global Memory)**

### **.dg** [*object*]

The **.dg** command displays a list of the global memory objects in the global heap. The command displays information in much the same way as the HEAPWALK application.

#### *object*

Specifies the first object to be listed. The *object* parameter can be a handle, a selector, or a heap address.

The list has the following form:

*address: size segment-type owner [handle flags chain]*

#### *address*

Specifies physical and heap addresses.

#### *size*

Specifies the size of the object, in bytes.

#### *segment-type*

Specifies the type of object. The type can be any one of the following:

CODE	Segment contains application code.
DATA	Segment contains application data and possible stack and local heap data.
FREE	Segment belongs to pool of free memory objects ready for allocation by an application.
PRIV	Segment contains private data.
SENTINAL	Segment marks the beginning or end of the global heap.

#### *owner*

Specifies the module name of the application or library that allocated the memory object. The acronym PDB is used for memory objects that represent process descriptor blocks. These blocks contain execution information about applications.

#### *handle*

Specifies the handle of the global memory object. If the debugger displays no handle, the segment is fixed.

#### *flags*

Specifies either of the following:

D	The segment is movable and discardable.
L	The segment is locked. If the segment is locked, the lock count appears to the right of the flag.

If the debugger displays a handle but no flag, the segment is movable but not discardable.

#### *chain*

Specifies the previous and next addresses in the list of least recently used (LRU) objects. Addresses are displayed only if the segment is movable and discardable (specified by the D flag).



## **.dh (Display Local Heap)**

### **.dh**

The **.dh** command displays a list of the local memory objects in the local heap (if any) belonging to the current data segment. The command uses the current value of the DS register to locate the data segment and check for a local heap.

The list of memory objects has the following form:

*offset: size { **BUSY** | **FREE** }*

*offset*

Specifies the address offset from the beginning of the data segment to the local memory object.

*size*

Specifies the size of the object, in bytes.

If **BUSY** is displayed, the object has been allocated and is currently in use. If **FREE** is displayed, the object is in the pool of free objects ready to be allocated by the application. A special memory object, **SENTINAL**, may also be displayed.



## **.dm (Display Modules)**

.dm

The **.dm** command displays a list of the global modules in the global heap.

The list has the following form:

*module-handle module-type module-name filename*

*module-handle*

Specifies the handle of the module.

*module-type*

Specifies either a dynamic-link library (DLL) or the name of the application you are debugging.

*module-name*

Specifies the name of the module.

*filename*

Specifies the name of the file from which you loaded the application.



## **.dq (Display Task Queues)**

### **.dq**

The **.dq** command displays a list containing information about the various task queues supported by the system. The list has the following form:

*task-descriptor-block stack-segment:stack-pointer number-of-events*

*priority internal-messaging-information module*

*task-descriptor-block*

Specifies the selector or segment address.

The task descriptor block is identical to the process descriptor block (PDB).

*stack-segment:stack-pointer*

Specifies the stack segment and pointer.

*number-of-events*

Specifies the number of events waiting for the segment.

*priority*

Specifies the priority of the segment.

*internal-messaging-information*

Specifies information about internal messages.

*module*

Specifies the module name.



## **.ds (Dump Stack)**

### **.ds**

Dumps the protected-mode stack and displays near code-segment labels (if available) next to each doubleword value from the stack. One important distinction that should be made is the difference between the **k** and the **.ds** commands. The **k** command will walk the Windows stack as long as the debugger is stopped in Windows-based application or dynamic-link library (DLL) code. However, if the debugger is tracing through ring 0 code, the **k** command will not produce any useful output. For this reason, the **.ds** command has been provided to display the ring 0 stack.



## **.du (Display Least Recently Used)**

**.du**

The **.du** command displays a list of the least recently used (LRU) global memory objects in the global heap. The list has the following form:

*address: size segment-type owner [handle flags chain]*

*address*

Specifies physical and heap addresses.

*size*

Specifies the size of the object, in bytes.

*segment-type*

Specifies the type of object. The type can be any one of the following:

CODE	Segment contains application code.
DATA	Segment contains application data and possible stack and local heap data.
FREE	Segment belongs to pool of free memory objects ready for allocation by an application.
PRIV	Segment contains private data.
SENTINAL	Segment marks the beginning or end of the global heap.

*owner*

Specifies the module name of the application or library that allocated the memory object. The acronym PDB is used for memory objects that represent process descriptor blocks. These blocks contain execution information about applications.

*handle*

Specifies the handle of the global memory object.

*flags*

Specifies D, which means the segment is movable and discardable.

*chain*

Specifies the previous and next addresses in the LRU list.



## **.m (Memory)**

**.m**[*struct*][*link*] [*expression* [L *count*]]

Displays information about memory and memory objects.

### *struct*

Command option. Can be one of these letters.

- a** Display arena record.
- c** Display context descriptor
- d** Display page directory entry
- e** Toggle stopping in the debugger for memory manager errors
- f** With no arguments, dump page fault log. With arguments, set to log faults in a specific range (syntax: **.MF** <address> L<length in bytes>).
- fb** Toggle stopping for each logged page fault.
- ff** Toggle logging page faults to debug terminal.
- g** Display pager descriptor
- h** [*handle*] Displays heap information for the given *handle* or global information if no *handle* is given.
- hs** [*handle*] Dump statistics about a given heap.
- hw** Toggle paranoid VMM32 heap checking.
- hx** [*handle*] Check for corruption on given heap defaults to VMM32 fixed heap.
- i** Displays instanced data regions.
- m** [*address* [L *page-count*]] Force memory present. The *address* defaults to last page faulted upon (contents of cr2 register)
- n** Force all unlocked memory not present
- r** Display arena header
- s** Dump memory manager statistics
- t** Displays page table entry
- u** Toggles computing checksums for page-outs and page-ins
- w** Dumps allocators of all committed pages.
- x** Displays all memory structures for an address.
- y** Displays valid physical memory ranges.

If no letter is given, the command displays all the structures associated with a given linear address.

### *link*

Direction to walk structures if dumping more than one. Can be **b** to walk backward or **f** to walk forward. If *link* is not given, the default is to walk adjacent entries.

### *expression*

An expression identifying the structure to display. Can be a linear address, handle, array index, or other value as specified by the *struct* parameter.

### *count*

Number of structures to display if the specified structures are linked.

The default address for the **.m** and **.mx** commands is the contents of the cr2 register (the last page fault) and for the other commands it is the first structure in the list. For example, **.m** displays information about the page that most recently faulted.



A word of caution about the **.mm**, **.mhx**, and **.mhs** commands. If you invoke these commands at the wrong time, you can crash your system. The only time it can be guaranteed safe is if you are currently executing ring 3 code. At other times we may be in the middle of another file system or memory manager operation.



## **.reboot (System Restart)**

### **.reboot**

The **.reboot** command causes the target system to restart.



## **.v (Virtual Machine Commands)**

**.v[option]**

Displays information about the current virtual machine (VM).

*option*

Command option. Can be one of these option letters:

- |          |  |
|----------|--|
| <b>c</b> | Displays the standard fields of the current VM's control block.  |
| <b>h</b> | Displays the current VM's handle.  |
| <b>m</b> | Displays the status of the current VM. Status information includes re-entry count, VM handle, critical section state, client registers, and top entries from the client's stack. |
| <b>r</b> | Displays the current VM's client registers if the 80386 debugger is running in protected mode.   |
| <b>s</b> | Displays the current VM's virtual mode stack if the debugger is running in protected mode.   |
| <b>l</b> | Displays a list of all valid VM handles.   |



## **.vmm (Display VMM Commands)**

### **.vmm**

Displays a menu of subcommands. Pressing a single key causes the corresponding command to be carried out.



## **.w (Display Win32 Object)**

**.w**[*option*][*expression*]

Displays information about a Win32 object.

### *option*

Can be one of these option letters:

HE	Toggles stopping in debugger for Win32 memory manager error returns
HW	Toggles Win32 paranoid heap corruption checking
L	Displays LDT-related information, for example, selman lists, sel free lists.
M	Displays module table.
MP [ <i>ppdb</i> ]	Displays module table for process. If <i>ppdb</i> is omitted, the module table for the current process is displayed.
P [ <i>ppdb</i> ]	Displays process list.
C <i>context</i>	Displays context record.
E <i>exception</i>	Displays exception record.
D <i>dispatcherContext</i>	Displays dispatcher context.
S	Displays the status of all ring 3 system critical sections.
T [ <i>ppdb</i> ]	Displays the process handle table. If <i>ppdb</i> is omitted, the handle table for the current process is displayed.

### *expression*

Must be the address of a Win32 object of one of these types: thread, process, semaphore, event, mutex, or critical section. If *expression* is omitted, information on all threads is displayed.

If any data item being dumped resides in not-present memory, the address of that data item is displayed in brackets.



