Wibug

Programmer's Reference Manual

188-190-203 A

May 1985

WICATsystems

first printing May 1985

Information about this Manual

Review the following items before you read this publication.

The subject of this manual

The manual describes how to use WIBUG, an assembly-language level, symbolic debugger.

The audience for whom this publication was written

This manual is written for programmers.

Table of Contents

Typographical Conventions Used in this Publication

Bold facing indicates what you should type.

Square brackets, [], indicate a function key, the name of which appears in uppercase within the brackets. For example, [RETRN], [CTRL], etc.

Underlining is used for emphasis.

Functional Description

Use WIBUG to debug an executable file. WIBUG is an assembly-language level, symbolic debugger. It has a built-in, 68000-based assembler and disassembler that uses Motorola mnemonics.

Command Line Syntax

Mnemonic	wibug
Optional parameter	file name
Optional parameter	file parameters

Parameters

filename
Function: This specifies the executable file you want to debug.
Default: WIBUG provides you with a 4K page of NOP's to experiment with.
Syntax: Type a single, standard file designation. Wildcard symbols or WMCS search paths cannot be used. (In other words, if you are not in the directory that contains the file, you must type the complete file designation.) You can also specify a logical name.

file parameters
ction: This specifies the parameters for the executable file.
Default: No parameters are passed to the executable file.

Examples

This command debugs an executable file named A.EXE and passes a parameter with the value 1000 to the executable:

```
> wibug a 1000
```

This command debugs an executable file named TEST.EXE but does not pass any parameters:

```
> wibug test
```

This command executes WIBUG and gives you a 4K page of nop's to experiment with:

```
> wibug
```

Section 1
General Information

a. Requirements for debugging programs

WIBUG cannot be used with executable files that exceed its size requirements, require the stack pointer to be in a specific location, or use certain traps.

The program to be debugged must fit into the 2 megabyte logical address space along with its data, its stack, and WIBUG. WIBUG uses about 128K of this space and also changes the location of the stack pointer. Figure 1 shows how WIBUG uses its share of logical address space.

Also, WIBUG cannot be used to debug a program that defines a trap handler for traps 13 (breakpoint), 19 (single step), or 22 (set exit handler). These traps are used by WIBUG.

NOTE: You can debug programs that define exit handlers, as long as they do not set an exit handler with trap 22.

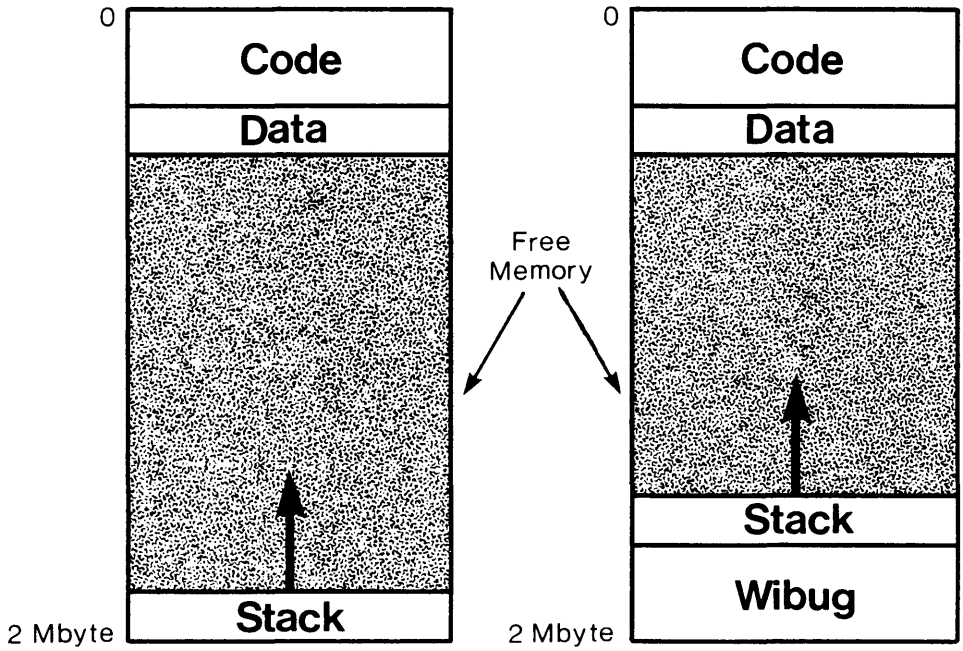


Fig 1 WIBUG uses some logical address space and moves the stack pointer

b. Executing and Exiting WIBUG

The syntax for executing WIBUG is explained at the beginning of the WIBUG description. It is just like typing **wibug** in front of the normal command line for your program.

WIBUG automatically tries to load the symbol table for your program. It does this by looking for a .mcr or .out file with the same name as the executable file. If WIBUG cannot find a symbol table it displays an error message.

After you execute WIBUG, its prompt appears:

```
->
```

At this point you can type any expression or WIBUG command.

To exit WIBUG, type:

```
-> ex [RETRN]
```

c. Editing the WIBUG Command Line

WIBUG is a line-oriented debugger. It has the same command-line editing functions as the CIP. One command that is particularly useful in WIBUG is [CTRL] e. It executes the previous command, which is helpful when you want to repeatedly single step through a program.

d. The Help Display

WIBUG has a help display, which is a syntax line for each WIBUG command. To see the help display, type:

```
-> he [RETRN]
```

e. Interrupting Execution

Type [CTRL] c to interrupt the execution of your program in WIBUG. You can also use [CTRL] c to abort WIBUG commands (such as a long disassembly). It is normally best to type a single, deliberate [CTRL] c. As a last resort, you can try two quick [CTRL] c's, which WIBUG recognizes as a "panic" interrupt. If you wish, you can resume execution where the program stopped.

f. WIBUG Error Messages

For the most part, WIBUG generates the diagnostic messages WMCS would normally generate while your program is running. WIBUG also generates a few of its own diagnostic messages that pertain to operating the debugger, such as a syntax error in a command or reporting a full symbol table.

g. I/O Devices Used With WIBUG

WIBUG performs its input and output through the devices or files specified by the logical names WIBUGIN and WIBUGOUT (defined on the CIP command line using logical name assignments). If these names are not defined before you execute WIBUG, the devices specified by SYS$INPUT and SYS$OUTPUT are used. If the printer is activated by WIBUG's PR command, WIBUG outputs to the device or file specified by the logical name WIBUGPRT. If this name is not defined, WIBUG uses SYS$PRINT. You can also alter the input and output devices in WIBUG by using the PR and TERM commands. This is helpful for programs that are screen oriented, so you can run the program on one terminal and operate the debugger on another. To do this start your program with WIBUG on one terminal but define the logical names WIBUGIN and WIBUGOUT for another terminal. Then, when WIBUG begins execution, control is switched to the second terminal. You can change control to another terminal after executing WIBUG with the TERM command (see section 2).

h. WIBUG Expressions

Expressions are used in WIBUG to perform operations and to specify addresses. Expressions are typed on WIBUG's command line by themselves to display memory or registers. Expressions combined with the assignment operator, =, are used to modify memory. Section 2 explains how to do these operations.

Expressions also specify addresses for WIBUG commands and can be used anywhere a command requires an address. Address arithmetic is performed with expressions using the four standard arithmetic operators (+, -, *, /), plus the bitwise logical operators for AND and OR (& and |). The @ sign is used to signify address indirection. (Indirection accesses the value at that address. Double indirection uses the value at the specified address as a pointer to a second address, whose contents are then accessed.) Any value can be specified with an expression. For example, the following expression evaluates to the value found at the address specified by

adding the value of the symbol _main to the hexadecimal value le:

```
-> @(_main+$le)
```

Double indirection is indicated with two @ signs:

```
-> @@(_main+$le)
```

Additional indirection can be specified (if you desire) with additional @ signs.

i. Wildcarding

Wildcarding can be used with all applicable command parameters, which includes the specification of registers and symbols. The syntax for wildcarding is the same as WMCS wildcarding (* for multiple characters and = for single characters). The equal sign, =, is also used as an assignment operator. If the meaning of = is ambiguous, WIBUG assumes it is used as a wildcard symbol. For example, the following command displays all three letter symbols whose names start with te (the exclamation point means the expression refers to symbols, not registers):

```
-> !te=
```

And this command displays all symbols, of any length, beginning with re:

```
-> !re*
```

j. Accessing Symbols

If a symbol from a program is the same as a reserved word in the debugger, you must precede the name of the symbol with an exclamation point, !, when referring to it in an expression. Otherwise, WIBUG thinks the symbol refers to the reserved word. For example, to use the symbol pc (the same as WIBUG's reserved word for program counter), you must type !pc. Also, when there is a conflict of names between registers and symbols, WIBUG defaults to registers. For example, this command line displays all registers:

```
-> *
```

Whereas this command line displays all symbols:

```
-> !*
```

And this command line displays all symbols that begin with r:

```
-> !r*
```

k. Input Formats

The default input for expressions is hexadecimal. You cannot change the default input, however, you can specify the format of an input value with the following notations:

Format	Input
decimal	Precede the value with %
hexadecimal	Precede the value with $
octal	Precede the value with \
floating point	No symbol. Number must contain decimal point. Syntax: [-] digit... . [digit...] [e[+|-] digit...]
character	Enclose the value in ' '
string	Enclose the value in " " (WIBUG automatically terminates the string with a null)
assembly	Enclose the value in { }
relative	There is no format symbol. Input the value as an expression. For example, _main+$56f

For example, the following is a decimal expression:

```
-> %16499 - %7500
```

You don't need to include a dollar sign, $, with a hex string since that is the default. For example, the following expression is the same as $lef + $2ed:

```
-> lef + 2ed
```

l. Memory Access Size

The size of memory accesses is specified by the switches :1 (byte), :2 (word), or :4 (longword). The initial default is longword. To specify a size other than the default for an

individual command, include the size specification anywhere on the input line. To change the default size for the remainder of the WIBUG session, type the size specification on a line by itself. For example, the following command changes the default size to word:

```
-> :2
```

The size of memory access can affect changes to memory locations and certain expressions. For example, if you assign 0 to a location and the access size is longword, 4 bytes are altered. However, if the access size is word, only two bytes are altered. Also, the size of memory derived from expressions can vary. For example, the following expression causes WIBUG to display 40 bytes of memory beginning at location 1000, assuming the default is longword:

```
-> 1000 for %10
```

But this expression displays only 10 locations (the access size is byte):

```
-> 1000 for %10 :1
```

m. Output Formats

You can specify the format of an output value with the following notations:

Format	Output
decimal	:decimal
hexadecimal	:hexadecimal
octal	:octal
single precision	:sp (used only with display and modify memory)
double precision	:dp (used only with display and modify memory)
character	:character
string	:string
assembly	:assembly
relative	:r followed by format symbol desired for the offset from the label. For example, :rd means relative with a decimal offset.

The default format is initially hexadecimal. Formats may be specified by a substring of the format name. For example, ":he" or ":h" are both valid specifications for hexadecimal format. To override the default format for an individual command, include the format specification anywhere on the input line. To change the default output format for the remainder of the WIBUG session, type the format specification on a line by itself. For example, the following command changes the default output format to assembly:

```
-> assembly
```

n. Address Ranges

You can indicate address ranges by specifying the starting address and the number of times to increment the starting address. Or you can specify a starting address and an ending address. For example, the following command displays the contents of eight memory locations of the default size beginning at location 1000 (hex 1000):

```
-> 1000 for 8
```

And this command displays the contents of addresses 1000 through 18e5 in default format:

```
-> $1000 to $18e5
```

Section 2
Dictionary of WIBUG Commands

Command descriptions appear in the following order:

br	Display and edit breakpoints
cp	Spawn a CIP
cs	Clear the screen
dh	Display history
(no mnemonic)	Display memory
(no mnemonic)	Display registers
ex	Exit WIBUG
he	Help
(no mnemonic)	Modify memory
pr	Turn on/off printing
rs	Read a symbol table
sb	Do Pascal- or C-style backtrace
sj	Step jsr, rts
sl	Step local
ss	Step single
term	Set WIBUG's terminal
xr	Execute realtime
xs	Execute silent
xt	Execute trace mode

Command Line Syntax

Display breakpoints	-> br
Set a breakpoint	-> br
Delete a breakpoint	-> br # *value*
Delete all breakpoints	-> br #*

Parameters

value	This specifies the address where a breakpoint is to be set or deleted.
#	This indicates the breakpoint is to be deleted.
*	This is a wildcard, meaning all breakpoints.

Operation

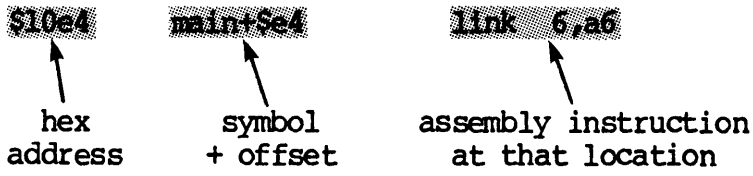
This command is used to display, set, or delete breakpoints. Breakpoints set with this command remain during the entire WIBUG session, unless you delete them. Breakpoints are referenced by the address at which the breakpoint is set. The XR, XS, and XT commands allow you to set temporary (one use only) breakpoints.

Examples

This command displays all breakpoints:

```
-> br
```

Here is a sample display of a breakpoint:



This command sets a breakpoint at location main+$lef (WIBUG does not generate a message to show the breakpoint was set):

```
-> br main+$lef
```

This command deletes a breakpoint at location $2e4fa:

```
-> br # $2e4fa
```

This command deletes all breakpoints:

```
-> br #*
```

Command Line Syntax

Spawn a CIP -> cp

Parameters

none

Operation

This command spawns a CIP. When you want to return to WIBUG by logging out of the CIP, you return to location in the program you were at before the CIP was spawned.

Examples

This command spawns a CIP:

```
-> cp
```

Command Line Syntax

clear screen -> cs

Parameters

none

Operation

This command clears the screen and places the WIBUG prompt, ->, at the bottom of a blank screen.

Examples

This command clears the screen:

```
-> cs
```

Command Line Syntax

display last 20 pc values -> dh

display a number of last pc values -> dh

Parameters

number This specifies the number of pc values to display. The default is 20 decimal. The maximum value of this parameter is 100 decimal.

Operation

This command displays a history of program execution. During tracing the last 100 pc (program counter) values (addresses) are stored. Each value displayed also includes the corresponding assembly instruction for that location.

NOTE: You cannot keep track of the execution if you use the XR (execute real time) command to execute the program. The DH command displays four question marks, ????, instead of a pc value to indicate that an XR command was executed.

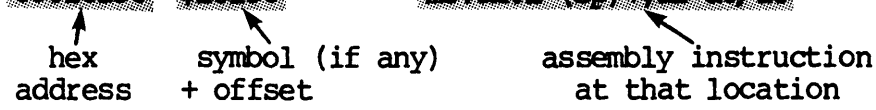
Examples

This command displays the 50 (decimal) instructions that were last executed and their addresses:

```
-> dh %50
```

Here is a sample five-line display (produced by the command dh 5):

```
00010148  $10148          tst.b    (a1)
0001014a  $1014a          *bne.s   $10152
00010152  $10152          cmpm.b   (a0)+,(a1)+
00010154  $10154          *dbne    d0,$1013a
00010158  $10158          movem.l  (sp)+,a1-a0/d0
```



Command Line Syntax

display one location	@
display a number of locations	for *number*
display a range of locations	*value* to *value*

Parameters

value	Specify one location in memory.
value for *number*	Specify a beginning location for the value. Specify a number of locations for the number.
value to *value*	Specify the beginning location for the left value and the ending location for the right value.
:a	Add this to the command line if you want the values in the specified locations disassembled. (:a is a substring of :assembly)
:sp	Specify this with @value to display the 4 bytes, beginning at value, as a single-precision number.
:dp	Specify this with @value to display the 8 bytes, beginning at value, as a double-precision number.

Operation

There is no command mnemonic for displaying memory. You display memory by simply listing an address or a range of locations (see section 1-n). The number of actual bytes displayed depends on the memory access size (section 1-1). If you display a single location, the at sign, @, must precede the value. If you display a range, you should not include the at sign.

Examples

This command displays the contents at the address hex $1000 and disassembles value:

```
-> @$1000 :a
```

Here is a sample display the previous command would produce (assuming the memory access size is longword, :4):

```
00001000  $1000 jmp     $3b30
```

This command displays 2e hex locations (in bytes) beginning at location 45fb:

```
-> $45fb for $2e :1
```

This command displays the contents of locations 4500 through 4600 hex (the default input is hex):

```
-> 4500 to 4600
```

This command displays the contents of locations 4500 through 4600 This command displays the contents of 4 bytes beginning at location 5fa (hex) as a single-precision number:

```
-> @$5fa :sp
```

This command displays the contents of 8 bytes beginning at hex location 5680 (default is hex) as a double-precision number:

```
-> @5680 :dp
```

Command Line Syntax

display all registers *

display a register

Parameters

register This specifies the register whose contents you want to display. The asterisk means all registers. Data registers are specified by d0 through d7, address registers by a0 through a7 (d* displays all data registers, a* all address registers). The program counter is specified by pc, the current stack pointer by sp and the user stack pointer by usp. Register a7 also contains the value of the stack pointer in use at the time.

Operation

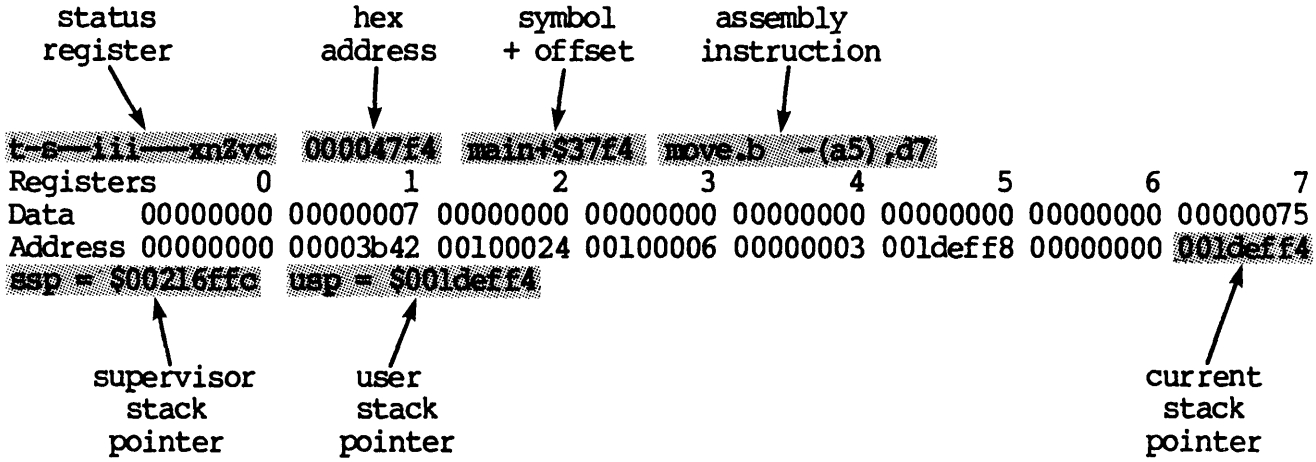
In WIBUG, you display the contents of a register by specifying its name. Wildcarding applies (with * and =).

Examples

This expression displays the contents of all registers:

```
-> *
```

A display of all registers looks like this:



This expression displays the contents of all data registers (d* also works):

```
-> d=
```

This expression displays the contents of the program counter:

```
-> pc
```

Command Line Syntax

exit WIBUG -> ex

Parameters

none

Operation

This command allows you to exit WIBUG. Any changes you made to your program during the WIBUG session are lost.

Command Line Syntax

help display -> he

Parameters

none

Operation

The help command displays a page of one-line syntax summaries for each WIBUG command.

Command Line Syntax

modify memory	@	=
modify a register		=
modify or create a symbol	*symbol*	

Parameters

@ *value* = *value*

The value on the left specifies a location. The value on the right specifies the value assigned to that location. Use the input format explained in section 1-k. You can use the address ranges explained in section 1-n for either of this parameters. If you use a range for the left parameter, do not use the at sign, @. For example, assuming the default is hexadecimal, this command copies 16 locations, in the default size, from 2000 to 1000 (i.e., locations 2000 - 203f are copied to 1000 - 103f if the default size is :4):

```
-> @1000 = 2000 for 10
```

And, assuming the default is hexadecimal, this command initializes 16 words with 4e71 beginning at location 2000:

```
-> 2000 for 10 = 4e71 :2
```

register =

The value is assigned to the register you specify.

symbol value

If you specify a symbol that already exists, the value is assigned to the symbol. If you specify a symbol that does not exist, a symbol is created with the value you specified. Use an exclamation mark in front of symbols with the same name as WIBUG reserved words or those that could be

confused with registers.

NOTE: There must be a space on each side of the equal sign, =, or else a wildcard function is assumed.

:sp — Specify this with @value = fp_value to store the floating-point value in the 4 bytes beginning with the address specified by value. The floating-point value must be in the correct format specified in section 1.k.

:dp — Specify this with @value = fp_value to store the floating-point value in the 8 bytes beginning with the address specified by value. The floating-point value must be in the correct format specified in section 1.k. Double-precision is the default value, so if you specify without :sp or :dp, WIBUG assumes it is double precision.

Operation

You modify memory by using the assignment operator. For more than one location, use the address ranges explained in section 1-n.

Examples

This expression assigns the assembly command tst.b (a4)+ to location 105e:

```
-> @105e = {tst.b (a4)+}
```

This expression assigns zero to locations 1000 to 1020:

```
-> 1000 to 1020 = 0
```

And this expression assigns decimal 20 for 20 bytes (10 accesses of word) beginning at location 1200:

```
-> 2000 for %10 = %20 :2
```

This expression assigns the hex value 15ee4 to data register 3:

```
-> d3 = 15ee4
```

And this expression creates a symbol called ROUTINE1 and assigns it the hex value 1000:

```
-> routine1 = 1000
```

This expression stores the single-precision value zero to the 4 bytes beginning at location 54f:

```
-> @54f = 0. :sp
```

This expression stores the double-precision value 3.14159 to the 8 bytes beginning at location 1000 (hex):

```
-> @$1000 = 3.14159 :dp
```

This expression stores the double-precision value -123.456e-44 to the 8 bytes beginning at location 2000 (hex):

```
-> @$2000 = -123.456e-44
```

Command Line Syntax

toggle on/off	-> pr
turn on printing	-> pr on
turn off printing	-> pr off
change print terminal	-> pr *terminal name*

Parameters

terminal name This specifies the port of a terminal or printer where you want WIBUG's output echoed.

Operation

This command allows you to turn on or turn off printing, or to specify a new terminal or printer where the printing goes. When print is on, WIBUG echoes everything that appears on your screen. In other words, the output appears in both places.

Examples

This command turns printing on:

```
-> pr on
```

This command specifies _TT3 as the printer port:

```
-> pr _tt3
```

Command Line Syntax

Read from a file -> rs *filename*

Read from file with same name as .exe file -> rs

Parameters

ename

This is the name of a file with a .mcr or .out extension. You don't need to specify the extension unless there is file with with the same name for both extensions. If you don't specify a filename, the .mcr or .out file with the same name as the current .exe file is used.

Operation

This command loads a symbol table from a .mcr or .out file into WIBUG's symbol table. If symbols already exist in the symbol table, this command adds to the list of symbols. If a symbol of the same name exists in both the symbol table and the file, the value in the file overwrites the value currently in the symbol table.

Examples

This command causes WIBUG to look for a symbol table in a file named SUM.MCR or SUM.OUT, and read the table into WIBUG if it is found:

```
-> rs sum
```

Command Line Syntax

stack backtrace -> sb

Parameters

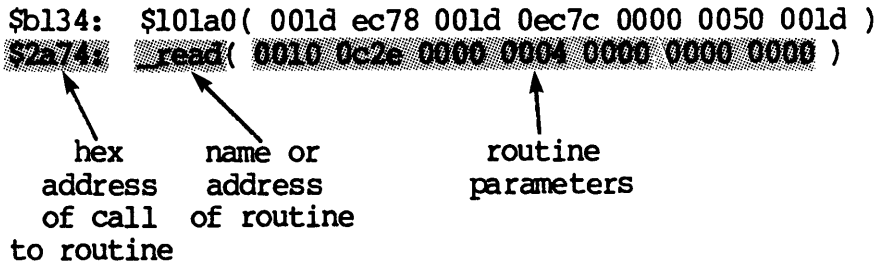
none

Operation

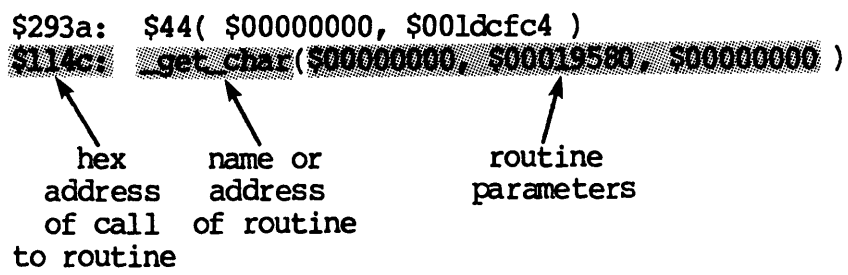
This command prints out a backtrace of the calling sequence, starting with the current stack frame and continuing back until the end of the stack is reached. The name of each routine called is printed, followed by the parameters. For C programs, the parameter list is in the proper order and values of the parameters are displayed as long words. For Pascal programs, the parameter list is in reverse order of the declared parameters. Also, for Pascal programs, WIBUG displays the value of the parameters as words. Pascal passes parameters as words and longwords, but WIBUG does not make judgments about the length of the parameters. Thus, the value of a long word parameter is found in two adjoining words.

Examples

Here is a sample display of a stack backtrace for a Pascal routine:



Here is a sample display of a stack backtrace for a C routine:



Command Line Syntax

```
step jsr, rts        -> sj
```

Parameters

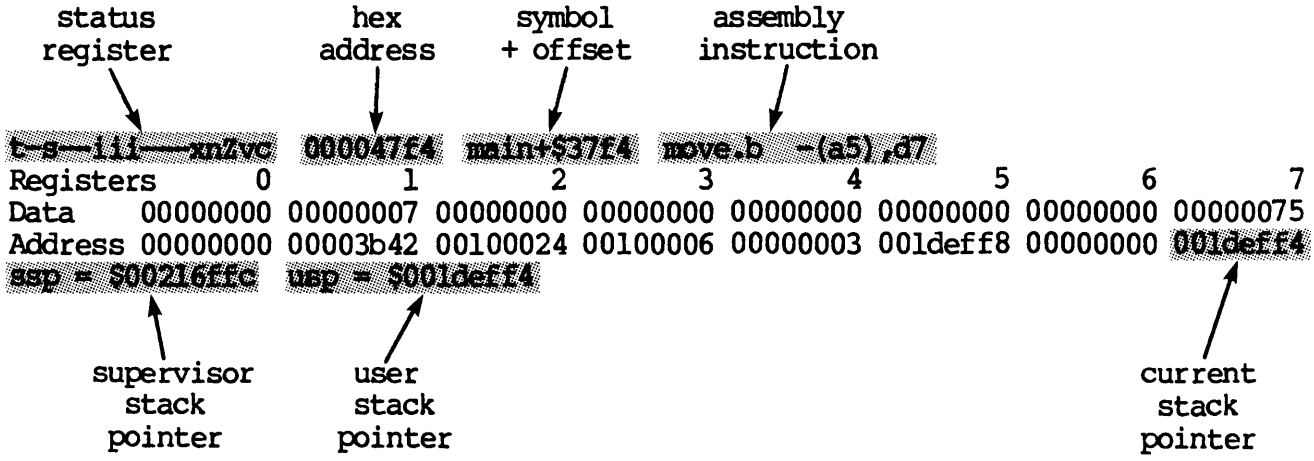
none

Operation

This command causes your program to execute until it encounters a jsr, bsr, rts, rtr, or rte instruction and then displays the contents of the registers. In other words, SJ is the same as the SS command, except it only stops on jsr, bsr, rts, rtr, and rte instructions.

Examples

A register display looks like this:



Command Line Syntax

step local -> sl

Parameters

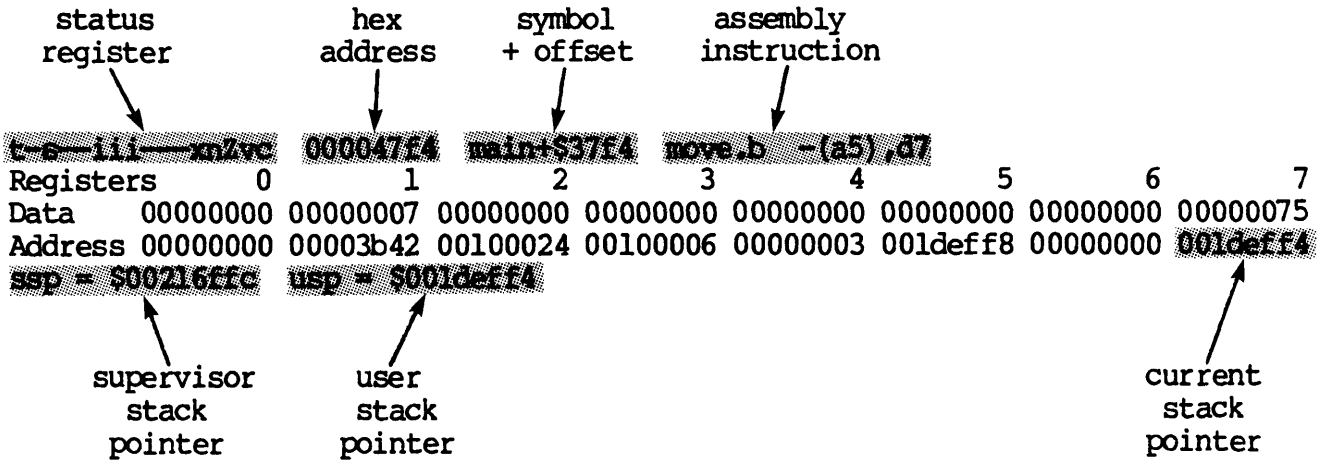
none

Operation

This command executes your program one step at a time, and displays the contents of the registers after each step, except for subroutines. It treats a subroutine as one step, and executes the entire routine and returns to the line after the routine call before displaying the registers. In other words, it is the same as the SS command, except a call to a subroutine is counted as one instruction and WIBUG does not show single steps through the routine.

Examples

A register display looks like this:



Command Line Syntax

step single -> ss

Parameters

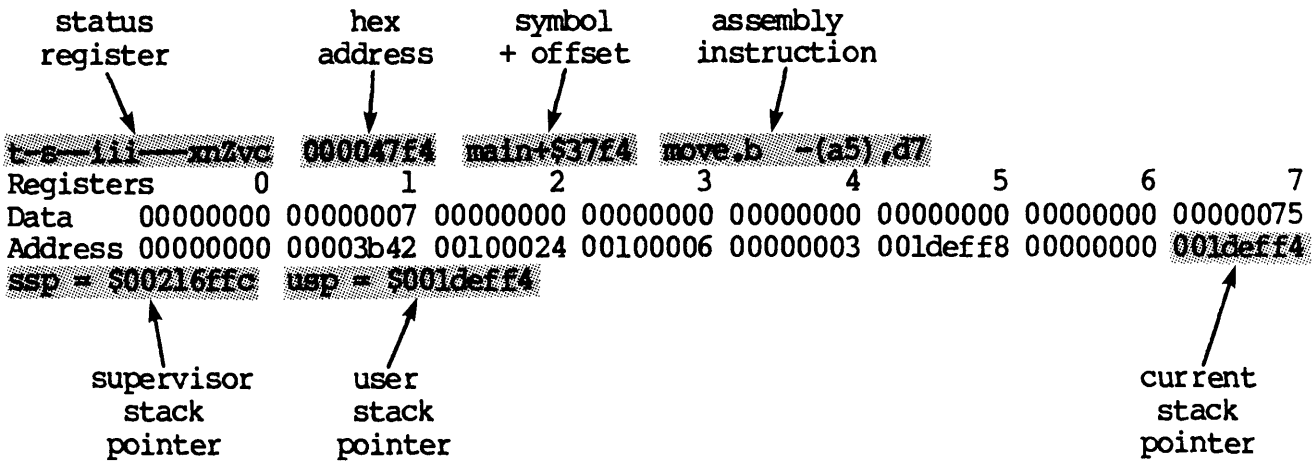
none

Operation

This command executes a single instruction and then dumps the current pc and the contents of all other registers.

Examples

A register display looks like this:



Command Line Syntax

set input terminal	-> term <	
set output terminal	-> term >	
set both input & output	-> term	*terminal name*

Parameters

terminal name This specifies a port that has been mounted on your system.

Operation

This command changes WIBUG's terminal to the specified device. You can change the input terminal, the output terminal, or both. If you change the output terminal, the output is displayed there but not on your terminal. In other words, the output is not echoed to both terminals. This command is helpful when you want to execute your program on one terminal and operate WIBUG on another terminal.

Examples

To operate WIBUG on another terminal once you have executed your program with WIBUG, you would change the input terminal. This command changes the input terminal to _TT5

```
-> term < _tt5
```

Command Line Syntax

execute to end or [CTRL] c -> xr

execute and specify temporary breakpoint -> xr

Parameters

temp breakpoint This specifies a temporary breakpoint that is set just before execution begins and is cleared when execution ends.

Operation

This command executes your program until a breakpoint is encountered, your program ends, or you type [CTRL] c. Your program executes at full speed with this command. Breakpoints are physically stored in your program as trap number 13. Execution begins at the current pc. WIBUG cannot keep a history of the pc with this command. The output your program generates or error messages are the only output generated with this command. This command, when used with the temporary breakpoint, allows you to execute quickly to a spot in your program you want to work with more closely.

Examples

This command causes your program to be executed until it encounters a breakpoint and sets a temporary breakpoint at location 4efe:

```
-> xr 4efe
```

Command Line Syntax

execute	-> xs
execute and specify temporary breakpoint	-> xs

Parameters

temp breakpoint This specifies a temporary breakpoint that is set just before execution begins and is cleared when execution ends.

Operation

This command executes your program until a breakpoint is encountered, your program ends, or you type [CTRL] c. The XS command is similar to the XR command but with one very important difference. While the XR command executes your program at full speed, the XS command silently single steps your program. The single stepping is very slow, but WIBUG is able to keep a history of the pc. The only display generated by this command is the display generated by your program and error messages.

Examples

This command causes your program to executed and keep track of the program counter history:

```
-> xs
```

Command Line Syntax

execute	-> xt
execute and specify temporary breakpoint	-> xt *temp breakpoint*

Parameters

temp breakpoint This specifies a temporary breakpoint that is set just before execution begins and is cleared when execution ends.

Operation

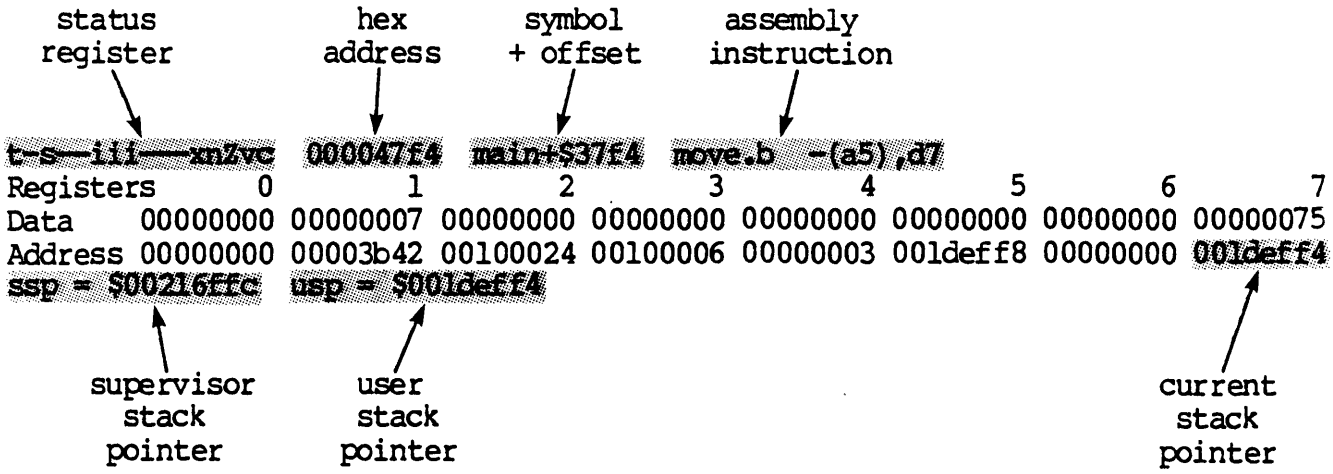
This command executes your program until a breakpoint is encountered, your program ends, or you type [CTRL] c. The XT command is just like the XS command except that XT displays the registers between each single step. There is a one second delay between each step to give you a chance to glance at the register display and decide if you want to interrupt execution with [CTRL] c.

Examples

This command causes your program to execute a step at a time and display the registers after each step. It executes until a breakpoint is encountered and the command also sets a temporary breakpoint at location 204e:

```
-> xt 204e
```

A register display looks like this:



WICAT Systems, Inc.
Product-documentation Comment Form

We are constantly improving our documentation, and we welcome specific comments on this manual.

Document Title: _____

Part Number: _____

Your Position:

- ☐ Novice user
- ☐ Experienced user
- ☐ Applications programmer
- ☐ System manager
- ☐ Systems analyst
- ☐ Hardware technician

Questions and Comments Page No.

Briefly describe examples, illustrations, or information that you think should be added to this manual.

_____ _____

_____ _____

_____ _____

What would you delete from the manual and why?

_____ _____

_____ _____

_____ _____

What areas need greater emphasis?

_____ _____

_____ _____

_____ _____

List any terms or symbols used incorrectly.

_____ _____

_____ _____

_____ _____

First Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 00028 OREM, UTAH

POSTAGE WILL BE PAID BE ADDRESSEE

WICAT Systems, Inc.
Attn: Corporate Communications
1875 S. State St.
Orem, UT 84058

Second Fold

Tape