Rocket UniData

Using the UniBasic Debugger

Version 8.2.1

July 2017
UDT-821-UDEB-1

Notices

Edition

Publication date: July 2017
Book number: UDT-821-UDEB-1
Product version: Version 8.2.1

Copyright

Trademarks

Examples

License agreement

Corporate information

Rocket Software, Inc. develops enterprise infrastructure products in four key areas: storage, networks, and compliance; database servers and tools; business information and analytics; and application development, integration, and modernization.

Website: www.rocketsoftware.com

Rocket Global Headquarters
77 4th Avenue, Suite 100
Waltham, MA 02451-1468
USA

To contact Rocket Software by telephone for any reason, including obtaining pre-sales information and technical support, use one of the following telephone numbers.

Country	Toll-free telephone number
United States	1-855-577-4323
Australia	1-800-823-405
Belgium	0800-266-65
Canada	1-855-577-4323
China	400-120-9242
France	08-05-08-05-62
Germany	0800-180-0882
Italy	800-878-295
Japan	0800-170-5464
Netherlands	0-800-022-2961
New Zealand	0800-003210
South Africa	0-800-980-818
United Kingdom	0800-520-0439

Contacting Technical Support

The Rocket Community is the primary method of obtaining support. If you have current support and maintenance agreements with Rocket Software, you can access the Rocket Community and report a problem, download an update, or read answers to FAQs. To log in to the Rocket Community or to request a Rocket Community account, go to www.rocketsoftware.com/support.

In addition to using the Rocket Community to obtain support, you can use one of the telephone numbers that are listed above or send an email to support@rocketsoftware.com.

Contents

Chapter 1: Using the debugger

You can use the UniBasic debugger to interactively examine the source code and the value of variables. This chapter introduces the terms and concepts you need to begin using the debugger.

For the syntax of debugger commands, including descriptions and examples, see Debugger commands reference, on page 25.

Getting started

This section explains how to prepare a program for debugging and get started with the debugging session. It includes the following subsections:

- Locating the source code, on page 7 – The source code must be in the same directory as the executable to be accessible to the debugger. Use the debugger `S` command to load the source code into the local directory.
- Linking the symbol table, on page 8 – Make available the symbol table that is used by the debugger.
  - Using the ECL `BASIC` command with debugging options.
  - Using the debugger `EXEC` command.
- Loading the symbol table, on page 9 – Load the table with data from the program by using the debugger `Z` command.
- Displaying the symbol table, on page 9 – List the symbol table to see all variables used in the program.
- Invoking the debugger, on page 10 – Start up the debugger by executing:
  - The `ECL RUN` command.
  - The program from the ECL prompt.
- Getting debugger help, on page 11 – Enter `H` at the debugger prompt to get online help.
- Exiting from the debugger, on page 11 – Enter `ABORT` or `END` to get out of the debugger. You can program the ON.ABORT clause to control where users are returned after exiting.

Locating the source code

The debugger requires that the source code be located in the same directory as the executable file. If it is not, you will need to use the debugger `S` command to load source code into the directory after you enter the debugger.

Syntax

S {*directory.name prog.name* | *pathname*}

You can use either of the following types of command lines to execute the debugger `S` command, assuming the program is located in the BP directory:

- `!S /`*`path`*`/BP/TEST` (for UNIX), or `!S C:\`*`path`*`\BP\TEST` (for Windows platforms)
- `!S BP TEST`

Linking the symbol table

The symbol table lists all program elements and their types. These elements include literals, constants, variables, labels, and arrays. The debugger will not run until a symbol table is linked for the program being analyzed. You can use the symbol table to learn about the program, and you can use the index numbers provided in the table in some debugger commands. You can link the symbol table in the following two ways:

- ECL `BASIC` command using a debugger option.
- Debugger `EXEC` command followed by the ECL `BASIC` command with a debugger option.

ECL BASIC command

Execute the ECL `BASIC` command with a debugger option to compile and build the symbol table.

Syntax

```
BASIC filename [TO filename] prog.name1 [progname2...] [options]
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
filename	UniData DIR-type file containing the source code to be compiled.
TO *filename*	UniData DIR-type file to receive the object code record, if different from the location of the source code record.
program	Source code to be compiled. You can compile more than one program by separating the names with a space.
options	See the following table..

Options

The following table lists the `BASIC` command options.

Option	Description
-D	Creates a cross-reference table for use with the UniBasic debugger.
-G	Generates a program that you can run with profiling.
-L -LIST	Generates a list of the program.
-X -XREF	Generates a cross reference table of statement labels and variable names used in the program.
-Z*n*	Creates a symbol table for use with the UniBasic debugger. UniData does not recompile the program or expand $INCLUDE statements. Use one of the following options: ▪ Z1—for programs compiled on a UniData release earlier than release 3.1. ▪ Z2—for programs compiled on UniData release 3.1 or later.
-I	When you compile a program with the -I option, all reserved words in UniBasic are case insensitive.

Debugger EXEC command

You also can generate the symbol table from the debugger. Use the debugger `EXEC` command to execute the `BASIC` command. The debugger -Z option directs UniBasic to build the symbol table.

Syntax

`EXEC BASIC` *`directory.name prog.name`* `[-Z1 | -Z2]`

Loading the symbol table

After you generate the symbol table, load the table with data from the program by using the debugger `Z` command.

Syntax

`Z` `{`*`directory.file prog.name`* `|` *`cat.name`*`}`

Example

You might execute a program and break to the debugger only to find that the program was not compiled with a debugger option. In this case, you can generate and load the symbol table, as shown in the following example:

```
!EXEC BASIC BP locate.array -Z2
***Executing UNIDATA command: BASIC BP locate.array -Z2
Compiling Unibasic: BP/locate.array in mode 'u'.
compilation finished
!Z BP locate.array
```

Displaying the symbol table

The debugger `\*` command displays the symbol table for the program you specify.

Syntax

`\*` `[`*`directory.file prog.name`* `|` *`cat.name`*`]`

Examples

The following example shows the symbol table with the names of the program elements and their types for the program DEBUG.TEST:

```
!\*
Program Name: BP/_DEBUG.TEST
                Name                                Type
                I                                       variable
                A                                       one dimension array
                L123                                label
```

In the following sample session, a programmer uses debugger option –Z2 to run a program for which no symbol table exists, and then generates and loads the symbol table from the debugger by using the ECL BASIC and debugger Z commands:

```
:RUN BP DEBUG.TEST -Z2
DEBUG.TEST
***DEBUGGER called at line 4 of program BP/_DEBUG.TEST
!\*
***There is not symbol table in BP/_DEBUG.TEST
!EXEC BASIC BP DEBUG.TEST -Z2
***Executing UNIDATA command: BASIC BP DEBUG.TEST -Z2

Compiling Unibasic: BP/DEBUG.TEST in mode 'u'.
compilation finished
!Z BP DEBUG.TEST
!\*
Program Name : BP/_DEBUG.TEST
                Name                            Type
                VAR1                            variable
                VAR2                            variable
```

Invoking the debugger

After you prepare the program with one of the debugger options, invoke the debugger using one of the following methods:

- Execute a program that has UniBasic DEBUG statements in the source code.
- While running the program, press the break or interrupt key. You must know what key is defined for this purpose on your system.
- Run or execute the program with a debugger execution option.

When you invoke the debugger, UniBasic displays the debugger prompt (!).

Program DEBUG statement

A UniBasic DEBUG command invokes the debugger. The debug statement operates in conjunction with the ECL DEBUG.FLAG command. When DEBUG.FLAG is ON, UniBasic executes the DEBUG command and invokes the debugger. When DEBUG.FLAG is OFF, UniBasic ignores the DEBUG command.

For more information, see DEBUG, on page 42. For more information about the ECL DEBUG.FLAG command, see the *UniData Commands Reference*.

The interrupt key

You can stop program execution and enter the debugger by pressing the interrupt key, which is usually defined as Ctrl+C. To learn which key is defined for break or interrupt on your system, see your system administrator.

Note: UDT.OPTIONS 38 determines where UniData positions you after you interrupt a program. When UDT.OPTIONS 38 is ON, you are returned to ECL. When UDT.OPTIONS 38 is OFF, you are returned to the debugger. For more information about UDT.OPTIONS, see the *UDT.OPTIONS Commands Reference*.

Run or execute with debugger option

When you run or execute the program, you can use the following options to invoke the debugger:

Syntax

```
RUN directory.name prog.name [-D | -E | -F | -D -E | -D -F]
```

or

```
directory.name prog.name [-D | -E | -F | -D -E | -D -F]
```

Options

The following table describes the debugger options.

Option	Description
-D	Immediately enters the debugger before the program executes.
-E	Enters the debugger whenever a warning or runtime error occurs.
-F	Enters the debugger only when a fatal error occurs.
-D -E	Immediately enters the debugger. You execute the program from the debugger, and then, if UniData encounters a warning or runtime error, it returns to the debugger.
-D -F	Immediately enters the debugger. You execute the program from the debugger, and then, if UniData encounters a fatal error, it returns to the debugger.

For more information about debugger commands, including syntax and usage, see Debugger commands reference, on page 25.

Getting debugger help

The UniBasic debugger provides online help. When you enter `H` at the debugger prompt, UniBasic displays an alphabetic summary of the debugger commands. You also can enter a command name on the line with `H`. In the following example, the programmer is requesting information on the debugger `ABORT` command:

```
!H ABORT
```

Exiting from the debugger

When you want to terminate a debugging session, enter `ABORT` or `END`. The following table identifies where UniData returns control.

Command	Condition	Action
`ABORT`	ON.ABORT paragraph is coded.	Proceeds to ON.ABORT paragraph in the program.
`ABORT`	ON.ABORT paragraph is not coded.	Returns to ECL.
`END`	UDT.OPTIONS 14 is ON and ON.ABORT paragraph is coded.	Proceeds to ON.ABORT paragraph in the program.

Command	Condition	Action
`END`	UDT.OPTIONS 14 is OFF, or ON.ABORT paragraph is not coded.	Returns to ECL.

Note: The `ON.ABORT`command lets you designate a paragraph to execute when a program aborts. You can code the paragraph to do whatever you want, such as execute a menu (which prohibits access to ECL), trap the abort condition and write to a file, or log off.

For more information about UDT.OPTIONS, see the *UDT.OPTIONS Commands Reference*. For information about UniData paragraphs and the ECL `ON.ABORT` and `CLEAR.ONABORT` commands, see the *UniData Commands Reference*.

Displaying program code and output

During the debugging process, you can direct the UniBasic debugger to display or suppress code as it executes, and display or suppress program output. You could find these commands especially useful in conjunction with the `E` command, which executes a specific number of lines of code.

Debugger command	Action
`V`	Displays or suppresses display of each line of code before it executes.
`P`	Displays or suppresses display of program output on the display terminal.

For more information about debugger commands, including syntax and usage, see Debugger commands reference, on page 25.

Querying the debugger

From the debugger prompt, you can list program status and variable values.

These interactive display commands provide the following capabilities.

Debugger command	Action
`$`	Prints the current line number and program name.
`?`	Lists all subroutine programs on the stack. The stack contains the calling program name and the number of the line after the call. This information provides a path through the program calls to the current location.
`D`	Lists all breakpoints and tracepoints. For more information about breakpoints and tracepoints, see Using breakpoints and tracepoints, on page 13.
`L`	Lists a specified number of lines of code surrounding the current line.
`DI`	Displays database system connection information for open servers, including user name, application name, and server name.
`DL`	Displays elements inside a select list. In BASICTYPE U, use a select list numbered from 0 through 9. In BASICTYPE P, use a variable name.
`FI`	Displays information about a file, including the type, name, path, index information, and privileges.
`LI`	Displays lock status and type (for locked records).

For more information about debugger commands, including syntax and usage, see Debugger commands reference, on page 25.

Printing and changing variables

The debugger provides commands that let you display and change the values in variables.

The following table describes these commands.

Debugger command	**Action**
`\`*`array`*	Prints the contents of all elements in an array as hexadecimal, octal, or ASCII (the default), and enters new values.
`\`*`variable`*	Prints the content of a variable, array, array element, or substring as hexadecimal, octal, or ASCII (the default), and enters a new value.
`SV`	Changes the value of a variable without displaying the original value of the variable.

For more information about debugger commands, including syntax and usage, see Debugger commands reference, on page 25.

Watching variables change

Use the watch commands to display the values in variables when they change without interrupting program execution. You can watch any number of variables with a single watch command. When you set watch on a variable, UniData displays breakpoints and tracepoints associated with that variable.

Watch commands provide the following capabilities.

Debugger command	**Action**
`W`	Sets watch on designated variables.
`WC`	Clears the watch on specified variables. The default is to clear the watch on all variables.
`WD`	Displays all variables that are being watched.

Tip: The breakpoint and tracepoint commands also display the values of variables. They offer more flexibility, but have more complex syntax. For more information, see Using breakpoints and tracepoints, on page 13.

For more information about debugger commands, including syntax and usage, see Debugger commands reference, on page 25.

Using breakpoints and tracepoints

Like the watch commands, you can use the breakpoint and tracepoint commands to view labels and variables during a program run. Unlike the watch commands, breakpoint and tracepoint commands control program execution so you can execute a portion of the program before reentering the debugger. You can also use these commands to monitor expression evaluation and break on subroutine and program calls.

Note: The debugger stores breakpoints and tracepoints in break and trace tables. The tables provide index numbers that are required by some debugger commands. The BD, TD, and D commands list the table(s).

What are breakpoints?

A breakpoint is a point at which UniBasic displays a variable value and enters the debugger. Use these commands to stop execution and examine the program and its variables at key points in program execution.

What are tracepoints?

A tracepoint is a point at which UniBasic displays variable values and continues to execute the program. You can elect to display variables at specific points in the program execution or when the values of the variables change.

What you can trace and break on

You can set breakpoints and tracepoints on any of the following:

- Successful expression evaluation, such as when the variable FLAG multiplied by the variable CODE is greater than 7(FLAG * CODE > 7).
- Source code line number.
- Label.
- Variable value, such as when the variable FLAG has a value of 7 (FLAG = 7).
- Subroutine call.

Note: You also can use the watch commands to display the value of variables. Watch commands are less flexible, but are much simpler to use than breakpoint and tracepoint commands. For more information, see Watching variables change, on page 13.

Breakpoint and tracepoint commands

You can use the following commands to create and use breakpoints and tracepoints. These commands reference the index numbers in the break or trace table. If you do not designate an index entry number, UniBasic acts on all breakpoints or tracepoints.

Command	Action
`B` or `T`	Creates new breakpoints or tracepoints associated with variable(s).
`BG` or `TG`	Creates and associates with a label.
`BL` or `TL`	Creates and associates with a line number.
`BP` or `TP`	Creates and associates with a program call.
`BD` or `TD`	Displays break and trace tables.

Command	Action
D	Lists break and trace tables.
BE or TE	Enables.
BU or TU	Disables.
BC or TC	Clears.

Examples

In the following example, the output displays the six breakpoints currently set. For each entry in the table, UniBasic lists:

- The index number required by breakpoint and tracepoint commands.
- The program for which the breakpoint is specified.
- The commands and arguments associated with the breakpoint.
- Whether the entry is currently enabled or disabled.

  Break table:

```
!D
Break table:
[1] /disk2/uddev/ud61/sys/CTLG/p/port: BP port Enable
[2] BP/_DBG.DEMO: BG LABEL.4 Enable
[3] BP/_DBG.DEMO: BL 38 Enable
[4] BP/_DBG.DEMO: B I? I>2 -U 3 Disable
[5] BP/_DBG.DEMO: B ARRAY(7)? ARRAY(7)>1 -D ARRAY(7) Enable
[6] BP/_DBG.DEMO: B I? I>2 -U 3 Enable
```

The following table describes the parameters in line [2] of the D command output.

Value in example	Description
[2]	Line number in the break table.
BP/_DBG.DEMO	Directory and name of the program being debugged.
BG	Command(s) that set the breakpoints and/or tracepoints on this line.
LABEL.4	Name of the variable, label, or other element associated with the breakpoint or tracepoint.
Enabled	Status of the breakpoint or tracepoint (enabled or disabled).

The following table explains additional examples of the B and T commands. For more information about the B and T command occurrence codes -A, -E, and -U, see B, on page 32 and T, on page 56.

Command	Description
T CODE -A 2	Traces the CODE variable and starts displaying its value after it changes twice.
T CODE -E 2	Traces the CODE variable and displays its value every second time the value changes.
T CODE -U 3	Traces the CODE variable and displays its value twice (until the value changes three times).
B CODE -A 2	Breaks when the value of the CODE variable changes twice.

Command	Description
T CODE -E 2 -D TYPE	Traces the CODE variable and displays its value every second time the value changes. In addition, displays the value of TYPE when the value of CODE displays.

Saving and loading the debug environment

The UniBasic debugger enables you to save information about debug environments to the `_DEBUG_` file. This file can contain debug information, including breakpoints, tracepoints, watches, and status information, for multiple debug environments identified by record ID. During a debug session, you can load debug environment information from the `_DEBUG_` file.

UniData creates the `_DEBUG_` file system file as an empty file, along with the associated dictionary file, under the account directory when you run the system-level `newacct` command.

Note: The current program must declare the variables included in the loading debug environment. Otherwise, the UniBasic debugger issues a syntax error message.

The following table describes the debugger commands available to save and load debug environments.

Command	Action
`EL`	Loads breakpoints, tracepoints, and watches, along with their status information, from the `_DEBUG_` file.
`ES`	Saves breakpoints, tracepoints, and watches, along with their status information, to the `_DEBUG_` file.

Accessing data in files

You can open UniData and operating system files from the debugger in read-write or read-only modes. With this capability, you can address file errors without stopping your process to correct the problem.

Debugger open file commands

The following table shows the commands you can use to open files from the debugger.

Debugger command	Action
`EL`	Loads breakpoints, tracepoints, and watches, along with their status information, from the `_DEBUG_` file.
`ES`	Saves breakpoints, tracepoints, and watches, along with their status information, to the `_DEBUG_` file.

Accessing a UniData file

This section explains how to open a UniData file and select record IDs.

1. Use the `SF` command to open the file *filename* to the file *variable*.

   The syntax is:

SF *variable* [-R] [*filename* | DICT *filename* | *absolute.pathname*]

The following table describes each parameter of the syntax.

Parameter	Description
variable	Indicates the name of the file variable.
-R	Sets the file to read-only mode. The default is read-write.
DICT	Indicates that the file being opened is a dictionary.
filename	Indicates the name of the dictionary file being opened.
absolute.pathname	Indicates the operating system path to the UniData file, including the file name.

In the following example, the debugger opens the file TEST.FILE to the file variable TEST in read-only mode and opens the file DICT TEST.FILE to the file variable DICT.TEST.

```
!SF TEST -R TEST.FILE
!SF DICT.TEST DICT TEST.FILE
```

2. Use the `SL` command to select record IDs in *filename* to the designated select list number.

   The syntax is:

   SL {*select.list.no* | *select.list.name*} [DICT] *filename*

   Note: *select.list.name* is provided for BASICTYPE P programs only.

   In the following example, the `SL` command selects record IDs in TEST.FILE to list 0, and selects record IDs in DICT TEST.FILE to list 1:

```
!SL 0 TEST.FILE
!SL 1 DICT TEST.FILEParameter
```

Executing programs from the debugger

With UniBasic debugger execution commands, you can execute a program until it reaches a specified line, label, or program. If UniData does not find the target line, label, or program, and if it does not execute a DEBUG statement, the program completes.

Select from the commands in the following table to execute a program.

Debugger command	Action
`E`	Executes to a breakpoint, to the number of lines specified, or to the end of the program if the number of lines specified is not reached. The `E` command counts lines in external subroutines it executes.
`G`	Executes, beginning with a specified line, through the end of the program. The default is the current line. This command honors all settings, such as breakpoints and tracepoints. Unlike the `E` command, the `G` command does not allow you to specify the number of lines to execute.
`N`	Executes to a breakpoint, to the number of lines specified, or to the end of the program if the number of lines specified is not reached. The `N` command counts an external subroutine it executes as one line.
`OUT`	Executes until control passes back to the calling program.
`PG`	Executes to the specified label.

Debugger command	**Action**
`PL`	Executes the specified line. If the line is not executable, display an error message.
`PP`	Executes until the specified program is called.

Accessing ECL and the operating system

To execute ECL commands, or operating system commands and editors from the debugger, you can enter one of the following:

- Colon (`:`) or `EXEC` followed by a ECL command.
- `!` (bang) command followed by an operating system command.

Colon

To execute an ECL command, use a colon (`:`) followed by the ECL command you want to execute.

Syntax

`:`*`ECL.command`*

Example

In the following example, a colon is followed by the `TIME` command:

```
***DEBUGGER called at line 53 of program BP/_TEST.4
!:TIME
***Executing UniData command: TIME
Thu Sep 10 17:48:09 CDT 1995
***DEBUGGER called at line 53 of program BP/_TEST.4
!
```

EXEC command

You can use the `EXEC` command at the debugger prompt (`!`) to execute an ECL command.

Syntax

`EXEC` *`ECL.command`*

Example

In the following example, UniData executes the `DATE` command:

```
!EXEC DATE
***Executing UniData command: DATE
Thu Sep 10 17:48:09 CDT 1995
***DEBUGGER called at line 53 of program BP/_TEST.4
```

```
!
```

! (Bang) command

You can use the `!` (bang) command at the debugger prompt to execute an operating system command.

Syntax

`!`*`oper.sys.command`*

Examples

In the following example, the `!` command executes the UNIX command `ls -ld *VOC`, which lists files ending with the string "VOC." Notice the two exclamation points. The first one is the debugger prompt, and the second one is the debugger command.

```
!!ls -ld *VOC
***Executing UNIX command: ls -ld *VOC
-rw-rw-rw           1 bobm      unisrc      4096 Aug 19 15:24 D_VOC
-rw-rw-rw           1 bobm      unisrc      111616 Oct 28 13:11 VOC
!
```

In the next example, the exclamation point command executes the Windows command `dir *VOC`, which again lists files ending with the string "VOC":

```
!!dir *VOC
***Executing NT command: dir *VOC
Volume in drive D has no label.

Volume Serial Number is 6478-7FC4

Directory of D:\UniData60\demo
10/08/99        03:02p      4,096 D_voc
10/28/99        10:04a      105,472 Voc
                        2 File(s) 109,568 bytes
                            2,483,828,224 bytes free
!
```

Using dual-terminal debugging

Dual-terminal debugging enhances the debugging process. You can use this feature to log in to two terminals or open two windows: one for the debugger, and the other for application output. This method is especially helpful when you debug full-screen or GUI applications.

You can set up a dual-terminal debugging session from the ECL prompt or from within the debugger.

Initiating dual-terminal debugging (UNIX)

Use the following procedure to initiate dual-terminal debugging from the ECL prompt on UniData for UNIX.

1. Log in to two terminals or open two windows on the same terminal. Note the device numbers. You can use the UNIX `tty` and `w` commands to determine device numbers.

   Note: For more information about these operating system commands, see your operating system manuals.

   The following example uses the UNIX `tty` command to determine a device number:

   ```
   %tty
   /dev/pty/ttyv5
   ```

   The following example uses the `w` command to determine the device number for the second terminal or window:

   ```
   % w carolw
   5:26pm up 67 days, 1:04, 6 users, load average: 0.08, 0.09, 0.07
   User           tty                   login@ idle JCPU PCPU what
   carolw      pty/ttyv5       5:24pm                          w carolw
   carolw      pty/ttyv6       5:24pm  1                       -csh
   ```
2. Decide which terminal/window to use to interact with the debugger and which one to use for application output.

   In this example, we assign the terminals/windows as follows:

   - Application terminal – ttyv5
   - Debugger terminal – ttyv6
3. Initiate UniData on the application terminal or window.

   The following example demonstrates the `udt` command, which initiates UniData:

   ```
   % udt
   UniData Release 7.3 Build: (6109)
   (c) Copyright Rocket Software, Inc. 1985-2012.
   All rights reserved.

   Current UniData home is /users/ud_33219/.
   Current working directory is /users/carolw/carol.
   :
   ```
4. Put the debugging terminal to sleep.

   You must put the debugging terminal to sleep with a long sleep period to prevent the command processor from attempting to interpret your debugging commands. Issue this command on the debugging terminal at the UNIX prompt. In the following example, we use a sleep period of 3200000:

   ```
   % sleep 3200000
   ```
5. On the application terminal (in this example, ttyv5), execute the ECL `SETDEBUGLINE` command to make the debug line attachable. Be sure to specify the full path to the debugging terminal.

   The syntax for this is:

   `SETDEBUGLINE` *`absolute.path`*

   In the following example, the `SETDEBUGLINE` command makes a debug line attachable (in this example, ttyv6):

   ```
   :SETDEBUGLINE /dev/pty/ttyv6
   :
   ```

   The cursor returns immediately to the ECL prompt. If an error message is displayed, the debug line is not set. In this case, check your device number and path, and try again.
6. On the application terminal, attach the debug line to the debugger using the `DEBUGLINE.ATT` command.

The syntax is:

```
DEBUGLINE.ATT
```

In the following example, the `DEBUGLINE.ATT` command attaches the debug line to the debugger:

```
:DEBUGLINE.ATT
:
```

If you attempt to attach to a line before making the line attachable, or if you attempt to attach the wrong line, an error message displays. Check the device number and path you are using for the debugging terminal, and try again.

7. Run the program. Now you are ready to invoke your program with the appropriate debug options from the application terminal:

   ```
   :RUN NEWDEMO GADGETS -D
   ```

8. Invoke the debugger by:

   - Running a program that contains a DEBUG statement.
   - Running a program that uses a debugger option.
   - Running a program, and then pressing the interrupt key.

   For instructions about initiating the debugger, see Invoking the debugger, on page 10.

   When you enter the debugger, notification is sent to the debugging terminal (in this case, ttyv6):

   ```
   ***DEBUGGER called at line 1 of program NEWDEMO/_GADGETS
   !
   ```

   If debugger output does not appear on the debugging terminal (in this example, ttyv6), you must terminate the program to try again. You also will need to detach the line and remove the pointer to the terminal that is currently and incorrectly displaying the debugger messages. For the ECL commands to do this, see Ending dual-terminal debugging, on page 23.

9. Conduct the debugging session. Interact with the debugger on the debugging terminal (in this example, ttyv6), and interact with the program on the application terminal (in this example, ttyv5).

Initiating dual-terminal debugging (Windows platforms)

To initiate dual-terminal debugging from the ECL prompt on UniData for Windows Platforms, perform the following steps:

1. Log in to two terminals or open two windows on the same terminal.
2. From the terminal or window on which you want to run the debugger, use the `dbgterm` command to initiate the dual-terminal debugging session.

   Note: Because the `dbgterm` command uses TCP/IP, you must have TCP/IP installed on both the debugger and application terminals.

   The `dbgterm` command is located in the `udtbin` directory.

   The syntax is `dbgterm`.

   After you execute the command, the screen displays the host address (with the format *hostname*:*portnumber*). The host name can be a symbolic name (in the following example, engine) or a literal name (for example, `hst1.rs.com`). The host address can be an IP address that includes a port number at the end (for example, 192.245.120.110:4294).

```
D:\>dbgterm
Host address: engine:4284
```

The `dbgterm` command automatically assigns a port number. If you want to assign a specific port number, use the following command.

`dbgterm [-p portnumber]`

where *portnumber* is the number of the port you want to assign.

3. Start UniData on the application terminal or window by executing the `udt` command. The following example shows how to use the command from the MS-DOS prompt:

```
D:\U2\ud73\bin>udt

UniData Release 7.3 Build: (6109)
(c) Copyright Rocket Software, Inc. 2005-2012.
All rights reserved.

Current UniData home is D:\U2\ud73\
Current working directory is D:\UniData73\demo.
:
```

4. Set the debugging terminal line. On the application terminal, execute the ECL `SETDEBUGLINE` command to make the debug line attachable.

The syntax is:

`SETDEBUGLINE hostname:portnumber`

In the following example, the `SETDEBUGLINE` command makes a debug line attachable:

```
:SETDEBUGLINE engine:4294
:
```

In the following example, the `SETDEBUGLINE` command specifies an IP address and port number, which must be enclosed with quotation marks:

```
:SETDEBUGLINE "192.245.120.110:4294"
:
```

After you execute the `SETDEBUGLINE` command, the cursor returns immediately to the ECL prompt. If an error message is displayed, the debug line is not set. In this case, check your host name and port number, and try again.

5. On the application terminal, attach the debug line to the debugger using the `DEBUGLINE.ATT` command.

The syntax is:

`DEBUGLINE.ATT`

In the following example, the `DEBUGLINE.ATT` command attaches the debug line to the debugger:

```
:DEBUGLINE.ATT
:
```

If you attempt to attach to a line before making the line attachable, or if you attempt to attach the wrong line, an error message displays. Check the device number and path you are using for the debugging terminal, and try again.

6. Run the program. Now you are ready to run your program with the appropriate debug options from the application terminal:

```
:RUN NEWDEMO GADGETS -D
```

7. Invoke the debugger by:
   - Running a program that contains a DEBUG statement.
   - Running a program that uses a debugger option.
   - Running a program, and then pressing the interrupt key.

   For instructions about initiating the debugger, see Invoking the debugger, on page 10.

   When you enter the debugger, notification is sent to the debugging terminal:
   ```
   ***DEBUGGER called at line 1 of program NEWDEMO\_GADGETS
   !
   ```

   If debugger output does not appear on the debugging terminal (in this example, ttyv6), you must terminate the program to try again. You also will need to detach the line and remove the pointer to the terminal that is currently and incorrectly displaying the debugger messages. For the ECL commands to do this, see Ending dual-terminal debugging, on page 23.
8. Conduct the debugging session. Interact with the debugger on the debugging terminal, and interact with the program on the application terminal.

Ending dual-terminal debugging

To end dual-terminal debugging, complete the following steps:

1. Exit from the debugger using one of the following methods:
   - Let the program run to termination.
   - Enter the `END` or `ABORT` commands from the debugging terminal. For more information about these commands, see Exiting from the debugger, on page 11.
2. Detach the debugging terminal using the debugger or ECL prompt.

   If using the debugger prompt:

   On the application terminal, enter the debugger `LD` command or the debugger colon (`:`) command, followed by the ECL `DEBUGLINE.DET` command, as shown in the following examples:

   ```
   ! LD
   !
   ```

   or

   ```
   !:DEBUGLINE.DET
   !
   ```

   If using the ECL prompt:

   On the application terminal, at the ECL prompt, enter the ECL `DEBUGLINE.DET` command as follows:

   ```
   : DEBUGLINE.DET
   :
   ```

   In both of the previous methods, the cursor returns to the prompt, and UniData does not display a confirmation message. If you run the program now with debugging options, debugger messages will be routed to the application terminal. However, you can attach the debugging terminal again and resume dual-terminal debugging.
3. Release the debugging terminal from the debugger prompt or from ECL.

Is using the debugger, enter the debugger `LU` command, as shown in the following example:

```
!LU
!
```

If using ECL, enter the ECL `UNSETDEBUGLINE` command, as shown in the following example:

```
:UNSETDEBUGLINE
:
```

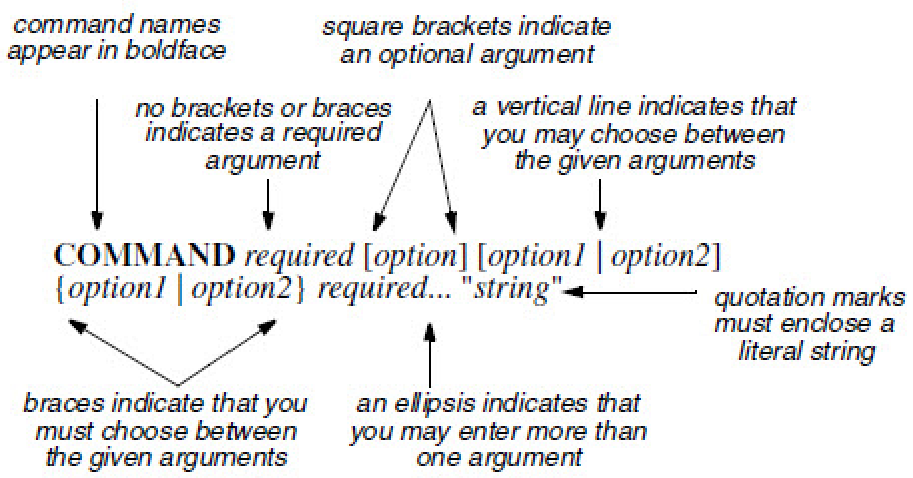
The cursor returns to the prompt again, and UniData does not display a confirmation message.

Chapter 2: Debugger commands reference

Elements of syntax statements

This reference manual uses a common method for stating syntax for UniData commands. The syntax statement includes the command name, required arguments, and options that you can use with the command. Italics represents a variable that you can replace with any valid option.

The following figure illustrates the elements of a syntax statement.


command names appear in boldface
square brackets indicate an optional argument
no brackets or braces indicates a required argument
a vertical line indicates that you may choose between the given arguments
COMMAND required [option] [option1 | option2]
{option1 | option2} required... "string"
quotation marks must enclose a literal string
braces indicate that you must choose between the given arguments
an ellipsis indicates that you may enter more than one argument

Summary of debugger commands

The following tables summarize the UniBasic debugger commands.

Note: All UniBasic debugger commands are case insensitive.

Display variables and break to debugger

Use the following commands to display the value of variables and/or exit to the debugger.

	Watch		Break		Trace
`W`	Watch (display variable when value changes)	`B`	Break on variable (exit to debugger).	`T`	Trace (display variable when value changes).
		`BG`	Break on label.	`TG`	Trace on label.
		`BL`	Break on line number.	`TL`	Trace on line number.
		`BP`	Break on program call.	`TP`	Trace on program call.
`WC`	Watch clear	`BC`	Break clear.	`TC`	Trace clear.
		`BU`	Break disable.	`TU`	Trace disable.
		`BE`	Break enable.	`TE`	Trace enable.

	Watch		Break		Trace
WD	Watch display (display variables being watched)	BD	Break display.	TD	Trace display.
		D	Display (all break and trace points).		

Display and change

Use the following commands to display information about variables and change their values.

Command	Action
`\`	Displays value of variable or array.
`\*`	Displays all variable names and types.
`SV`	String variable – Assigns string to variable.
`SZ`	Size – Displays size of variable.

Execute

Use the following commands to execute all or part of the program.

Command	Action
`G`	Go – Goes to line number and executes.
`GP`	Go Program – Begins executing at the program specified.
`E`	Execute – Executes specified number of lines (counts external subroutine lines).
`N`	Next – Executes specified number of lines (counts external subroutines as one line).
`OUT`	Back – Executes until RETURN.
`PG`	Proceed – Goes to the label specified, and then exits to the debugger.
`PL`	Line – Executes beginning at the line specified.
`PP`	Program – Executes until the specified program is called.

End

Use the following commands to terminate program execution and the debugging session.

Command	Action
`ABORT`	Ends by doing one of the following: Returns to ECL. Executes ON.ABORT paragraph.
`END`	Depending on setting of UDT.OPTIONS 14, ends by doing one of the following: Returns to ECL (when UDT.OPTIONS 14 is ON). Executes ON.ABORT paragraph (when UDT.OPTIONS 14 is OFF).

External execute

Use the following commands to execute ECL or operating system commands.

Command	Action
`!`	Executes operating system command.
`:`	Executes UniData command.
`EXEC`	Executes UniData command.

Open file

Use the following commands to open files.

Command	Action
`SF`	Opens a UniData file.
`SO`	Opens an operating system file.
`SS`	Opens a sequential file.

Dual-terminal debugging

Use the following commands to control dual-terminal debugging.

Command	Action
`LA`	Attaches line.
`LD`	Detaches line.
`LS`	Sets line.
`LU`	Unsets line.
`P`	Suppresses or displays program output to the display terminal.

Utilities

Use the following commands to obtain information about the program you are debugging.

Command	Action
`$`	Displays program name and line number.
`?`	Displays subroutine stack.
`DI`	Displays open server information.
`DL`	Displays select list elements.
`EL`	Loads breakpoint, tracepoint, and watch information, along with their status information, from the `_DEBUG_` file.
`ES`	Saves breakpoint, tracepoint, and watch information, along with their status information, to the `_DEBUG_` file.
`FI`	Displays file information.

Command	Action
L	Lists source code.
LI	Displays lock information.
P	Displays program output (toggle).
S	Loads source code.
SL	Selects record IDs to a variable.
V	Displays source lines before execution (toggle).
Z	Loads symbol table.

!

The debugger ! (bang) command lets you execute an operating system command from the debugger.

Syntax

!*oper.sys.command*

Examples

In the following example, UniData executes the UNIX command `ls -ld *VOC`, which lists files ending with the string "VOC." Notice that two exclamation points appear on the command line in this example. The first one is the debugger prompt, and the second one tells the debugger to execute the accompanying operating system command.

```
!!ls -ld *VOC
***Executing UNIX command: ls -ld *VOC
-rw-rw-rw       1 bobm      unisrc      4096 Aug 19 15:24 D_VOC
-rw-rw-rw       1 bobm      unisrc      111616 Oct 28 13:11 VOC
!
```

In the next example, the bang command executes the system-level command `dir *VOC`, which lists files ending with the string "VOC":

```
!!dir *VOC
***Executing NT command: dir *VOC
Volume in drive D has no label.
Volume Serial Number is 6478-7FC4

Directory of D:\UniData71\demo

10/08/99 03:02p         4,096 D_voc
10/28/99 10:04a         105,472 Voc
                    2 File(s) 109,568 bytes
                                2,483,828,224 bytes free
!
```

$

The debugger $ command prints the current line number and name of the program you are debugging.

Syntax

$

Example

The following example was executed at line 50 of the TEST program:

```
!$
***At line 50 of program BP/_TEST
```

:

The debugger **:** (colon) command executes an ECL command from the debugger.

Syntax

:*ECL.command*

Example

In the following example, the user entered a colon followed by the `TIME` command from the debugger:

```
***DEBUGGER called at line 53 of program BP/_TEST.4
!:TIME
***Executing UniData command: TIME
Thu Sep 10 17:48:09 CDT 1999
***DEBUGGER called at line 53 of program BP/_TEST.4
!
```

?

The debugger `?` command lists all subroutine programs on the stack. The stack contains the calling program name and the number of the line after the program call. This information provides a path through the program calls to the current line.

Syntax

?

Example

In the following example, the output lists two programs currently on the stack. The programs are GADGETS and MENU_DRIVER.

```
!?
[1] NEWDEMO/_GADGETS: 69
[2] /users/carolw/CTLG/MENU_DRIVER: 3
```

*

The debugger * command displays the debugger symbol table. The symbol table is required to run the debugger.

Syntax

\*

Example

The following example shows a partial listing of the symbol table for the sample program in *Developing UniBasic Applications*:

```
!\*
Program Name : BP/_UPDATE_ORDER
                Name                              Type
                ADDRESS                           variable
                CITY                              variable
                CLIENT.REC                    variable
                CLIENT_FILE                   variable
                CLIENT_NUMBER             variable
                COLOR                             variable
                COLOR_LINE                    variable
                COMMAND                           variable
                ENTRY                             variable
                FINISHED                      variable
                MESSAGE                           variable
                NEED.TO.WRITE             variable
                NEW.PRICE                     variable
                NEW_CLIENT                    variable
                NUM_ENTRIES                   variable
                OLD_CLIENT                 variable
...
```

\array

The debugger \*array* command prints the contents of the elements in an array as hexadecimal (X or x), octal (O or o), or ASCII (the default). The debugger displays each element individually, and follows each element with a prompt for a new value to replace that element.

Syntax

\array [X | x | O | o]

Parameters

The following table describes each parameter of the syntax.

Parameter	**Description**
array	Names the array to print.
X or x	Indicates that the array is to be displayed in hexadecimal.
O or o	Indicates that the array is to be displayed in octal.

Parameter	Description
No parameter	Indicates that the ASCII value of the array elements is to be displayed.

Example

In the following example, the elements in TEST.ARRAY are displayed one at a time, and the user is prompted for a replacement value:

```
!\TEST.ARRAY
TEST.ARRAY(1)=1^1;
Enter new value (hit <CR> for no change)=
Continue? (hit <CR> to continue)
TEST.ARRAY(2)=2^2;
Enter new value (hit <CR> for no change)=
Continue? (hit <CR> to continue)
.
.
.
TEST.ARRAY(10)=10^10;
Enter new value (hit <CR> for no change)=
```

\variable

The debugger `\variable`v command prints the contents of a variable as hexadecimal, octal, or ASCII (the default), and prompts for a replacement value.

Syntax

```
\variable [(i1 [,i2])] [<a,[v,[s]]>] [[d,] p, l]] [X | x | O | o]
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
variable	Specifies the variable to print and possibly change.
(*i1, i2*)	Specifies the element of a dimensioned array to be displayed.
<*a*, *v*, *s*>	Specifies the attribute, value, and subvalue of a dynamic array to be printed and possibly changed.
d, *p*, *l*	Specifies the delimiter, starting position, and length of a substring to be printed and possibly changed.
X or x	Indicates that the array is to be displayed in hexadecimal.
O or o	Indicates that the array is to be displayed in octal.
no parameter	Indicates that the array is to be displayed in ASCII.

Example

In the following example, the ASCII value of TEST.ARRY elements is displayed:

```
!\TEST.ARRAY(1)
TEST.ARRAY(1)=1^1;
Enter new value (hit <CR> for no change)=
!\TEST.ARRAY(1)<1>
TEST.ARRAY(1)<1>=1
```

```
Enter new value (hit <CR> for no change)=
!\TEST.ARRAY(1)<2>
TEST.ARRAY(1)<2>=1
```

The following example displays the value of the variable TITLE1 in the program GADGETS:

```
!\TITLE1
TITLE1=UniData Gadgets'
```

ABORT

The debugger `ABORT` command ends the debugging session. UniBasic executes the ON.ABORT paragraph if it is included in the UniBasic program. Otherwise, the cursor returns to ECL.

Syntax

ABORT

Related commands

Command	Description
`END`	The debugger `END` command ends the debugging session. UniBasic executes the ON.ABORT paragraph if it is included in the UniBasic program. Otherwise, the cursor returns to the ECL prompt. By turning UDT.OPTIONS 14 off, you can make UniBasic ignore the ON.ABORT paragraph and return the cursor to the ECL prompt.

B

The debugger `B` command creates a new breakpoint. You can include optional qualifiers that must be satisfied before the break is executed.

Note: At a breakpoint, UniData stops execution and enters the debugger. For more information about breakpoints, see Using breakpoints and tracepoints, on page 13.

Syntax

B *variable* [?*condition*] [-*occurrence.option*] *count* [-D *variable1* [,*variable2*]...

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
variable	The variable for which you want to create a breakpoint.
?condition	A condition that must be met to create a breakpoint. For example, if you specify ?X=7, UniBasic will not create a breakpoint until this condition is met.

Parameter	Description
-occurrence.option	A second condition that affects when the value of the variable will be displayed: ▪ -A – After the value changes *count* times. ▪ -E – Each time the value changes *count* times. For example, if *count* is set to 10, the value displays at the tenth, twentieth, thirtieth, and every tenth time the value changes. ▪ -U – Until the value changes the number of times specified in *count*. **Note:** Occurrence options are mutually exclusive. You can use only one for any breakpoint.
count	A number of variable changes. Used in *occurrence.option*.
-D	One or more variables are associated with this breakpoint. UniData displays the value of the associated variable(s) when the primary variable (and, if applicable, the breakpoint condition) is reached.
variable1...	Specifies the associated variable(s) to display when the primary variable (and, if applicable, the breakpoint condition) is reached.

Examples

In the following example, a tracepoint is associated with the variable ACTIVE.LINE program MENU_DRIVER. The contents of this variable are displayed every other time it changes. Notice that the command line overlays a portion of the program output showing the main menu.

```
            >> Client Information <<
!B ACTIVE.LINE -E 2 Information
Order Information
```

When the value in ACTIVE.LINE changes twice, the following message appears:

```
***Broke on command B ACTIVE.LINE -E 2
!
```

The following execution uses the sample program in *Developing UniBasic Applications*. First, the user creates a breakpoint associated with:

- Variable COMMAND
- Condition COMMAND="A"
- Occurrence -E2 (indicating every two times COMMAND changes)

Then the user executes the program with the debugger N command:

```
!B COMMAND ?COMMAND="A" -E2
!N
```

Finally, the debugger breaks after COMMAND changes twice and contains the value A:

```
ORDER MAINTENANCE
(Enter Q to quit)
Order #: 941
Date: 01/14/1996                York Software
Time: 03:00PM               2589 Celtic St.
                        Abbotsford
Client #10009               VIC, 3067 Australia
Product 50000
```

```
Color: Gray
Qty: 10
Price: $1,399.99


Enter A)lter, D)elete, or Q)uit: A
***Broke on command B COMMAND ?COMMAND="A"
***DEBUGGER called at line 209 of program BP/_UPDATE_ORDER
```

Related commands

Command	Description
T	The debugger `T` command creates a new tracepoint. You can display variables at specific points in program execution or when the values of the variables change. You also can include additional conditions that must be satisfied before the tracepoint is executed.
BC	The debugger `BC` command removes one or all breakpoints. To clear a single breakpoint, include the index entry for that breakpoint in the break table. If you do not specify an index number, the debugger deletes all breakpoints. The debugger does not confirm deletion of the breakpoint.
Breakpoints	For more information about breakpoints, see Using breakpoints and tracepoints, on page 13.

BASIC

The ECL `BASIC` command compiles UniBasic source code into interpretive code to be used with the UniBasic interpreter. UniData names the resulting object code record *_prog.name* where *prog.name* is the name of the source code record.

You can create a select list, and then execute `BASIC` to compile all programs in the select list. For example, to select and compile all UniBasic source files in the BP directory, enter `SELECT BP WITH @ID UNLIKE "_..."` and then enter `BASIC BP`.

Syntax

`BASIC` `*filename* [TO *filename*] *prog.name1* [*prog.name2*...] [*options*]`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
filename	The UniData DIR-type file containing the source code to be compiled.
TO *filename*	The UniData DIR-type file to receive the object code record if different from the location of the source code record.
prog.name	The source code to be compiled. You can compile more than one program by separating the names with a space.
options	See the following table.

Options

The following table lists the `BASIC` command options.

Option	Description
-D	Creates a cross-reference table for use with the UniBasic debugger. UniData will produce an include stack trace which will allow both the include name and line number to be reported if an error is encountered in the execution of the included code. The trace produced with the -D option will also allow the debugger to step through the included code if the `V` command is used with the debugger to turn on visual mode.
-G	Generates a program that you can run with profiling.
-L -LIST	Generates a list of the program.
-X -L -XREF -L	Generates a cross reference table of statement labels and variable names used in the program.
-Z*n*	Creates a symbol table for use with the UniBasic debugger. UniData does not recompile the program or expand $INCLUDE statements. Use one of the following options: ▪ Z1 – For programs compiled on a UniData release before release 3.1. ▪ Z2 – For programs compiled on UniData release 3.1 or later.
-I	When you compile a program with the -I option, all reserved words in UniBasic are case insensitive.

Examples

In the following example, the `BASIC` command compiles the program TEST, which is found in the BP file, and stores the resulting object code as _TEST:

```
:BASIC BP TEST -D

Compiling Unibasic: BP/TEST in mode 'u'.
compilation finished
```

In the next example, the `SELECT` command saves in select list 0 the names of all programs in the BP file with names (record IDs) beginning with T. Then, the `BASIC` command compiles the selected program.

In the next example, a `SELECT` statement is used to select all programs that start with the letter "T", and then they are compiled:

```
:SELECT BP WITH @ID LIKE "T..."
1 records selected to list 0.
>BASIC BP
Compiling Unibasic: BP/TEST in mode 'u'.
compilation finished
```

The next example saves the executable in a DIR-type file different than the one that contains the source code. In the first line, the program named test, which resides in BP, is compiled. The executable is placed in PROGRAMS. Finally, the program is executed from PROGRAMS, and it prints "Hello."

```
:BASIC BP TO PROGRAMS test
Compiling Unibasic: BP/test in mode 'u'.
compilation finished
:RUN PROGRAMS test
Hello
```

BC

The debugger BC command removes one or all breakpoints. To clear a single breakpoint, include the index entry for that breakpoint in the break table. If you do not specify an index number, the debugger deletes all breakpoints. The debugger does not confirm deletion of the breakpoint.

Note: At a breakpoint, UniData stops execution and enters the debugger.

Syntax

BC [*index_num*]

Example

The following statement clears the breakpoint associated with the variable TITLE2, identified by *index_num* 1:

```
!BC 1
```

To confirm deletion of the breakpoint, execute the BD command to display the break table:

```
!BD
***No breakpoints are set
!
```

BD

The debugger BD command displays the break table, which lists all breakpoints for the current program. The break table lists the item number for each variable, which is required by other debugger commands.

Note: At a breakpoint, UniData stops execution and enters the debugger.

Syntax

BD [*index_num*]

Example

In the following example, the debugger BD command displays the break table. Notice that the ACTIVE.LINE breakpoint is disabled.

```
!BD
[1]  /users/carolw/CTLG/ORDERFORM: BP ORDERFORM  Enable
[2]  /users/carolw/CTLG/MENU_DRIVER: B ACTIVE.LINE  Disable
```

Related commands

Command	Description
D	The debugger D command lists all breakpoints and tracepoints currently set for this debugging session.

Command	Description
TD	The debugger TD command displays the trace table, which lists all tracepoints, their status, and their index numbers. The trace table also provides the index number that is used in other debugger commands.

BE

The debugger BE command enables previously disabled breakpoints. *index_num* is an entry in the break table. If you do not specify an index entry, UniData enables all breakpoints.

Note: At a breakpoint, UniData stops execution and enters the debugger. For more information about breakpoints, see Using breakpoints and tracepoints, on page 13.

Syntax

BE [*index_num*]

Example

In the following example, the debugger BD command displays the break table. Then the statement BE 2 disables the breakpoint identified by *item_num* 2. Finally, BD is used again to redisplay the break table, demonstrating that the ACTIVE.LINE breakpoint has been enabled.

```
!BD
[1] /users/carolw/CTLG/ORDERFORM: BP ORDERFORM Enable
[2] /users/carolw/CTLG/MENU_DRIVER: B ACTIVE.LINE Disable
!BE 2
!BD
[1] /users/carolw/CTLG/ORDERFORM: BP ORDERFORM Enable
[2] /users/carolw/CTLG/MENU_DRIVER: B ACTIVE.LINE Enable
```

Related commands

Command	Description
BD	The debugger BD command displays the break table, which lists all breakpoints for the current program. The break table lists the item number for each variable, which is required by other debugger commands.
TE	The debugger TE command enables one or all previously disabled tracepoints. To enable a particular tracepoint, include *index_num*, an entry for a tracepoint in the trace table. If you do not specify *index_num*, UniData enables all tracepoints.
Breakpoints	For more information about breakpoints, see Using breakpoints and tracepoints, on page 13.

BG

The debugger BG command creates a breakpoint associated with a label.

Syntax

```
BG label [?condition[-occurrence.option] count [-D variable1
\[,variable2]...
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
label	The label for which you want to create a breakpoint.
?condition	A condition that must be met to create a breakpoint. For example, if you specify ?X=7, UniBasic will not create a breakpoint until this condition is met.
-occurrence.option	A second condition that affects when the break will execute: ▪ -A – After the value changes *count* times. ▪ -E – Each time the value changes *count* times. For example, if *count* is set to 10, the value displays at the tenth, twentieth, thirtieth, and every tenth time the value changes. ▪ -U – Until the value changes the number of times specified in *count*. Occurrence options are mutually exclusive. You can use only one for each breakpoint.
count	The number of times the variable changes. Used by *occurrence.option*.
-D	One or more variables are associated with this breakpoint. UniData displays the value of the associated variable when the label (and, if applicable, the breakpoint condition) is reached.
variable1...	An associated variable(s) to display when the label (and, if applicable, the breakpoint condition) is reached.

Example

The following example uses the sample program in *Developing UniBasic Applications*. First, the user creates a breakpoint associated with:

- Label DISPLAY_DATA
- Condition ORDER_NUMBER="941"

Then the user executes the program with debugger N command:

```
!BG DISPLAY_DATA ?ORDER_NUMBER="941"
!N
```

Finally, the debugger breaks after displaying the screen and receiving a value of 941 for the ORDER_NUMBER variable:

```
                                 ORDER MAINTENANCE
                                 (Enter Q to quit)
***Broke on command BG DISPLAY_DATA ?ORDER_NUMBER="941"
***DEBUGGER called at line 114 of program BP/_UPDATE_ORDER
```

BL

The debugger BL command creates a breakpoint associated with a line.

Note: At a breakpoint, UniData stops execution and enters the debugger.

Syntax

```
BL line [?condition[occurrence.option] count [-D variable1
\[,variable2]...
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
line	The line for which you want to create a breakpoint.
?condition	A condition that must be met to create a breakpoint. For example, if you specify ?X=7, UniBasic will not create a breakpoint until this condition is met.
-occurrence.option	Specifies a second condition that affects when the value of the variable is displayed: ▪ -A – After the value changes *count* times. ▪ -E – Each time the value changes *count* times. For example, if *count* is set to 10, the value displays at the tenth, twentieth, thirtieth, and every tenth time the value changes. ▪ -U – Until the value changes the number of times specified in *count*. Occurrence options are mutually exclusive. You can use only one in any breakpoint.
count	Provides a count used in *occurrence.option*.
-D	Indicates that one or more variables are associated with this breakpoint. UniData displays the value of the associated variable when the line (and, if applicable, the breakpoint condition) is reached.
variable1...	Specifies the associated variable(s) to display when the line (and, if applicable, the breakpoint condition) is reached.

Example

In the following example, the cataloged program ORDER_UPDATE is executed from the UniData prompt. The -D option initiates the debugger immediately. In this example, the user creates a breakpoint on program line 183 by entering the debugger BL183 command from the debugger prompt, and then executes 200 lines of the program with the debugger E200 command. The program breaks to the debugger at line 183 immediately after displaying the ORDER MAINTENANCE screen.

```
:UPDATE_ORDER -D
!BL183
!E200
                ORDER MAINTENANCE
                (Enter Q to quit)
Order #:
Date:
Time:

Client #:
Product #:
```

```
Color:
Qty:
Price:

***Broke on command BL183
!
```

BP

The debugger `BP` command defines a breakpoint associated with a program call.

Note: At a breakpoint, UniData stops execution and enters the debugger.

Syntax

BP {*directory.name prog.name* | *cat.name*}

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
directory.name	Name of the directory file where the noncataloged program is stored.
prog.name	Name of the noncataloged program.
cat.name	Name of the cataloged program.

Example

The following example sets a breakpoint on the program call to program MENU_DRIVER:

```
!BP MENU_DRIVER
***Set break point at program '/users/CTLG/MENU_DRIVER'.
!
```

When execution reaches a call to MENU_DRIVER, the following appears on the screen:

```
***Broke on command BP MENU_DRIVER
***DEBUGGER called at line 3 of program /users/CTLG/MENU_DRIVER
```

BU

The debugger `BU` command disables the breakpoints identified by an index entry in the break table. If you do not specify the index entry number, the debugger disables all index entries in the table. Disabled breakpoints remain in the break table and can be enabled with the `BE` command.

Syntax

BU [*index_num*]

Example

In the following example, the first line of a BU statement disables the breakpoint for the variable ACTIVE.LINE. The user executes the BD command to display the break table and the status of breakpoints.

```
!BU 2
!BD
[1] /users/carolw/CTLG/ORDERFORM: BP ORDERFORM Enable
[2] /users/carolw/CTLG/MENU_DRIVER: B ACTIVE.LINE Disable
!
```

D

The debugger D command lists all breakpoints and tracepoints currently set for this debugging session.

Syntax

D

Example

The following example displays the four breakpoints currently set. For each entry in the table, UniData lists:

- An index number that you can reference in breakpoint and tracepoint commands (*index_num*).
- The program for which the breakpoint is specified.
- The commands and arguments used with the breakpoint.
- Whether the entry is currently enabled or disabled.

```
!D
[1] /users/CTLG/MENU_DRIVER: B ACTIVE.LINE -A 2 Enable
[2] /users/CTLG/MENU_DRIVER: B PREV.LINE -A 2 Disable
[3] /users/CTLG/MENU_DRIVER: B MAX.MENU.AMC -A 2 Enable
[4] /users/CTLG/MENU_DRIVER: B MENU.ID Enable
!
```

The following table describes the values in line [2] of the D command output.

Value	Description
2	An index number that identifies this breakpoint. It is used in other debugger commands (*index_num*).
/users/CTLG/ MENU_DRIVER	The cataloged program is MENU_DRIVER.
B	A breakpoint is set.
PREV.LINE	The breakpoint is associated with the variable PREV.LINE.
-A 2	The contents of the variable will be displayed every two times it changes.
Disabled	This breakpoint has been disabled.

Related commands

Command	Description
`BD`	The debugger `BD` command displays the break table, which lists all breakpoints for the current program. The break table lists the item number for each variable, which is required by other debugger commands.
`BT`	The debugger `BT` command displays tracepoints and their status.

DEBUG

The UniBasic `DEBUG` command stops program execution and turns control over to the interactive UniBasic debugger. The debugger prompt (`!`) is displayed. Pressing BREAK also gives control to the debugger.

To use the `DEBUG` command to display the contents of variables, you must compile the program with the D option.

> **Note:** The setting of UDT.OPTIONS 14 determines where to return control after exiting a UniBasic program when you are using the UniBasic debugger and enter `ABORT` or `END`. For information about UDT.OPTIONS, see the *UDT.OPTIONS Commands Reference*.

When you enter the debugger, a message similar to the following displays, and the cursor is placed at the debugger prompt:

```
DEBUGGER called before line 1 of program BP/TEST
!
```

Syntax

`DEBUG`

Related commands

Command	Description
`ABORT`	The debugger `ABORT` command ends the debugging session. UniBasic executes the ON.ABORT paragraph if it is included in the UniBasic program. Otherwise, the cursor returns to ECL.

DI

The debugger `DI` command displays database system connection information for open servers, including user name, application name, and server name. *variable* is the name assigned to the server in the program being debugged.

Syntax

`DI` *`variable`*

DL

The debugger `DL` command displays the specified select list.

Note: In BASICTYPE U, use a select list numbered from 0 through 9. In BASICTYPE P, use the list name.

Syntax

```
DL [var | select.list]
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
var	The name of the stack to display (P mode only).
select.list	The select list number (U mode only).

E

The debugger `E` command executes source code lines based on the number of lines and beginning line specified. It executes the specified number of lines unless it encounters a breakpoint or the program ends. The `E` command functions like the debugger `N` command except that it counts lines in external subroutines it executes.

Note: At a breakpoint, UniData stops execution and the cursor returns to the debugger prompt.

Syntax

```
E [n] [R] [m]
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
n	The number of lines to execute before the cursor returns to the debugger prompt. The UniBasic debugger does not count comment lines.
R	Indicates that, after the `E` command executes *n* lines, an additional *n* lines will execute when the user presses Enter. This occurs each time the user presses Enter. If the user does not press Enter, but issues a different command, the `E` command stops executing.
m	The line at which to begin execution. The default is the current line.

Note: The blank space between the `E` command and the *n* parameter is optional. You must use spaces between the *n*, R, and *m* parameters.

Examples

The following example uses the sample program in *Developing UniBasic Applications*. The debugger `E` command executes 100 lines beginning with line 20 (GOSUB INITIALIZE).

```
!E 100 20
                    ORDER MAINTENANCE
```

```
 (Enter Q to quit)
Order #:
Date:
Time:

Client #:
Product #:

Color:
Qty:
Price:
```

In the following example, UniBasic executes 30 lines of the MENU_DRIVER program beginning at the current line. Notice that the debugger prints the last line of the main menu and the prompt for user input. The debugger is called after the user presses Enter.

```
!E30                          Order Information
                              Down,Up,Help or ESC to Exit, RETURN to Select Process!
```

EL

The debugger `EL` command loads breakpoints, tracepoints, and watches, along with their status information, from the `_DEBUG_` file. Use the *debugname* parameter to load information for a debug environment you previously saved by using the debugger `ES` command.

The name you specify for *debug_name* must match a record ID in the `_DEBUG_` file. If you did not specify *debug_name* when you saved the environment you now want to load, the record ID in `_DEBUG_` is an empty string. In this case, do not specify *debug_name* in the `EL` command.

Note: The current program must declare the variables included in the loading debug environment. Otherwise, the `EL` command issues a syntax error message.

Syntax

`EL` `[`*`debug-name`*`]`

Related commands

Command	Description
`ES`	The debugger `ES` command saves breakpoints, tracepoints, and watches, along with their status information, to the `_DEBUG_` file.

END

The debugger `END` command ends the debugging session. UniBasic executes the ON.ABORT paragraph if it is included in the UniBasic program. Otherwise, the cursor returns to the ECL prompt. By turning UDT.OPTIONS 14 off, you can make UniBasic ignore the ON.ABORT paragraph and return the cursor to the ECL prompt.

For more information about UDT.OPTIONS, see the *UDT.OPTIONS Commands Reference*. For information about paragraphs, ON.ABORT, and CLEAR.ONABORT, see the *UniData Commands Reference*. For instructions about using the UniBasic debugger, see Using the debugger, on page 7.

Syntax

END

Related commands

Command	Description
ABORT	The debugger ABORT command ends the debugging session. UniBasic executes the ON.ABORT paragraph if it is included in the UniBasic program. Otherwise, the cursor returns to ECL.

ES

The debugger ES command saves breakpoints, tracepoints, and watches, along with their status information, to the _DEBUG_ file. To save information for the current debug environment, use the *debug-name* parameter. The name you specify for this parameter becomes the record ID in the _DEBUG_ file for the current debug environment. If you do not specify *debug_name*, the record ID is set to an empty string by default.

Syntax

ES [*debug-name*]

Related commands

Command	Description
EL	The debugger EL command loads breakpoints, tracepoints, and watches, along with their status information, from the _DEBUG_ file.

EXEC

The debugger EXEC command executes an ECL command from the debugger.

Syntax

EXEC *ECL.command*

Example

In the following example, UniData executes the ECL DATE command:

```
!EXEC DATE
***Executing UniData command: DATE
Thu Sep 10 17:48:09 CDT 199
***DEBUGGER called at line 53 of program BP/_TEST.4
!
```

FI

The debugger FI command displays file information, including the type, name, path, index, and permissions for the file cited in the variable.

Syntax

FI *variable*

Example

In the following example, a user requests file information about the demo database file CLIENTS, which is opened to the variable CLIENT_FILE:

```
!FI CLIENT_FILE
'CLIENT_FILE' is a static hash file.
    file_name=CLIENTS
    path_name=/users/carolw/demo/CLIENTS
    hash_algorithm=0
    modulo=19
    group_size=0
    block_size=1024
    index=no
    privilege=read:write
!
```

G

The debugger `G` command begins execution with the specified line and continues until the end of the program. The default beginning is the current line. This command executes all breakpoints and tracepoints. You cannot specify the number of lines to process.

Syntax

G [*line*]

H

The debugger `H` command displays help for the debugger. The default (no parameters) displays an alphabetic summary of the debugger commands.

Syntax

H [*topic* | *debugger.command*]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
topic	A debugger topic for which you need help.
debugger.command	A debugger command or topic for which you need help.

Example

The following example demonstrates getting help on the debugger `ABORT` command:

```
!H ABORT
ABORT Ends the program and returns to ECL level.
```

```
!
```

L

The debugger `L` command lists source code lines surrounding the current line.

Syntax

L [*s*[,*e*]]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
s	The first line to list. If you do not specify *s*, the debugger lists the ten lines surrounding the current line.
e	The last line to list. If you do not specify *s*, but not *e*, the debugger lists 10 lines following *s*.

Example

In the following example, the programmer executes the `$` command to confirm the current line and program:

```
!$
***At line 3 of program /users/carolw/CTLG/MENU_DRIVER
```

Then the programmer executes the debugger `L` command to list lines 1 through 10 in program MENU_DRIVER:

```
!L1,10
[1]        $BASICTYPE 'U'
[2] *
(3)            SENTENCE = TRIM(UPCASE(@SENTENCE))
[4]            VERBOSE.FLAG = (FIELD(SENTENCE, ' ', 3) = "VERBOSE")
[5]        PROMPT ""
[6] *
[7]        OPEN "UCONV_MENUS" TO MENU.FD ELSE STOP "Can't open UCONV_MENUS file"
[8]        OPEN "UCONV_MENUS_HELP" TO MENU.HELP.FD ELSE STOP "Can't open
UCONV_MENUS_HELP file"
[9]        EQU MENU.TEXT TO 1
[10]       EQU MENU.ACTION TO 2
```

LA

The debugger `LA` command attaches a terminal line for debug input/output. The debugger attaches the line set by the ECL `SETDEBUGLINE` command or by the debugger `LS` command.

This command is used in dual-terminal debugging. For instructions about this process, see Using the debugger, on page 7.

Note: This debugger command is the same as the ECL `DEBUGLINE.ATT` command.

Syntax

`LA`

LD

The debugger `LD` command detaches the debug terminal line. The debug terminal line is the one set by the ECL `SETDEBUGLINE` command or the debugger `LS` command.

After detaching the debug terminal line, use the `LU` command to release it.

This command is used in dual-terminal debugging. For instructions about this process, see Using the debugger, on page 7.

Note: This debugger command is the same as the ECL `DEBUGLINE.ATT` command.

Syntax

`LD`

LI

The debugger `LI` command displays record lock status and type.

Syntax

`LI {`*`file.name`*` [`*`recordID`*`] | `*`resource.num`*`}`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
file.name	Name of the file to check. (Not the file variable name in the program.)
recordID	ID of the record to check (optional).
resource.num	Resource number. For more information about the UniBasic `LOCK` command, see the *UniBasic Commands Reference*. For more information about assigning resource numbers, see *Developing UniBasic Applications*.

Example

The following example checks the file CLIENTS for locks:

```
!LI CLIENTS
not locked
```

LS

The debugger `LS` command sets the debug terminal line.

This command is used in dual-terminal debugging. For instructions about this process, see Using the debugger, on page 7.

Syntax

`LS` *`absolute.pathname`* (for UNIX only)

`LS` *`hostname:portnumber`* (for Windows platforms only)

Parameters

For UNIX, *absolute.pathname* is the full path to the debugging terminal, including the device name (for example, `/dev/pty/ttyv6`).

For Windows NT or Windows 2000, *hostname*:*portnumber* is the host address (host name and port number separated by a colon) of the debugging terminal. The host name can be a symbolic name (for example, engine) or a literal name (for example, claireg.rs.com). The host address can be an IP address that includes a port number at the end, all of which must be enclosed in quotation marks (for example, "192.245.120.110:4294").

After setting the line, attach it with the debugger `LA` command or the ECL `DEBUGLINE.ATT` command.

LU

The debugger `LU` command releases the debug terminal line. The line released is the one set by the ECL `SETDEBUGLINE` command or by the debugger `LS` command.

This command is used in dual-terminal debugging. For instructions about this process, see Using the debugger, on page 7.

Syntax

```
LU
```

N

The debugger `N` command executes source code lines based on the number of lines and beginning line you specify. It executes the specified number of lines unless it encounters a breakpoint, or the program ends. The `N` command functions like the debugger `E` command, except it does not count lines in external subroutines it executes. In this case, the entire subroutine counts as one line.

Note: At a breakpoint, UniData stops execution and the cursor returns to the debugger prompt.

Syntax

`N` `[`*`n`*`] [R] [`*`m`*`]`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
n	The number of lines to execute before the cursor returns to the debugger prompt. The UniBasic debugger does not count comment lines.
R	Indicates that, after the `E` command executes *n* lines, an additional *n* lines will execute when the user presses Enter. This occurs each time the user presses Enter. If the user does not press Enter, but issues a different command, the `E` command stops executing.
m	The line at which to begin execution. The default is the current line.

Note: The blank space between the `N` command and the *n* parameter is optional. You must use spaces between the *n*, R, and *m* parameters.

Example

In the following example, the debugger `N` command executes 100 lines of code beginning with line 48:

```
!N 100 48
```

OUT

The debugger `OUT` command executes the subroutine you are debugging until control passes back to a calling program.

Syntax

`OUT`

Example

In the following example, a program calls the CALLED.PGM subroutine:

```
PRINT "Enter operation to perform: (1)add, (2)delete, (3)update : ";INPUT operation
CALL CALLED.PGM(operation,ret.val)
PRINT "Operation completed: ":ret.val
```

The following example shows the CALLED.PGM subroutine:

```
SUBROUTINE CALLED.PGM(operation,ret.val)
BEGIN CASE
        CASE operation = 1
                ret.val = "Record added."
        CASE operation = 2
                ret.val = "Record deleted."
        CASE operation = 3
                ret.val = "Record updated."
END CASE
RETURN
```

In the following example, the programmer executes the `PP` command to exit to the debugger when the external subroutine is called:

```
!PP CALLED.PGM
Enter operation to perform: (1)add, (2)delete, (3)update :
?1
***Broke in /users/ud_71/sys/CTLG/c/CALLED.PGM at line 2
```

```
!
```

Then the programmer executes the `OUT` command to return to the debugger when control returns to the calling program:

```
!OUT
***DEBUGGER called at line 2 of program BP/_sub.call
!
```

P

The debugger `P` command suppresses program output to the display terminal.

Syntax

`P`

This command is used in dual-terminal debugging. For instructions about this process, see Using the debugger, on page 7.

PG

The debugger `PG` command causes the program to proceed to the label you specify.

Syntax

`PG` *`label`*

Example

In the following example, the programmer uses the `PG` command to make the program break out of executing the program and proceed to the OPEN_FILES label. The debugger `L` command is executed to show the program lines surrounding the label.

```
!PG OPEN_FILES
***Broke in BP/_UPDATE_ORDER at line 253
!L
[249] END
[250] RETURN
[251]
[252]
(253) OPEN_FILES:
[254]       OPEN "CLIENTS" TO CLIENT_FILE ELSE
[255]        MESSAGE = "The CLIENT file could not be opened."
[256]        CALL DISPLAY_MESSAGE(MESSAGE)
[257]        STOP
[258] END
!
```

PL

The debugger `PL` command executes to the line you designate. When the program arrives at that line, execution stops. *line* cannot be a comment line.

Syntax

PL *line*

Example

In the following example, the `PL` command is executed from line 253 (the program label OPEN_FILES). Program execution continues until control is returned to Main Logic, line 20. (The `L` command shows the line surrounding line 20).

```
!PL20
***Broke in BP/_UPDATE_ORDER at line 20
!L
[16] GOSUB OPEN_FILES
[17]
[18] *-------------- Main Logic -----------------------------
[19]
(20) GOSUB INITIALIZE
[21]
[22] LOOP
[23]        GOSUB DISPLAY_SCREEN
[24]        GOSUB GET_ORDER_NUMBER
[25] UNTIL ORDER_NUMBER[1,1] = 'Q'
!
```

PP

The debugger `PP` command executes until the program calls another program or catalog.

Syntax

PP [*file.name prog.name* | *cat.name*]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
file.name	The name of the directory file where the noncataloged program is stored.
prog.name	The name of the noncataloged program.
cat.name	The name of the cataloged program.

Example

In the following example, a program calls the CALLED.PGM subroutine:

```
PRINT "Enter operation to perform: (1)add, (2)delete, (3)update : ";INPUT operation
CALL CALLED.PGM(operation,ret.val)
PRINT "Operation completed: ":ret.val
END
```

The following example shows the CALLED.PGM subroutine:

```
SUBROUTINE CALLED.PGM(operation,ret.val)
BEGIN CASE
        CASE operation = 1
```

```
                ret.val = "Record added."
        CASE operation = 2
                ret.val = "Record deleted."
        CASE operation = 3
            ret.val = "Record updated."
END CASE
RETURN
```

The programmer executes the -D option, which causes the program to break to the debugger immediately. Then the programmer executes the `PP` command, which makes the program return to the debugger when the external subroutine is called.

```
!PP CALLED.PGM
Enter operation to perform: (1)add, (2)delete, (3)update :
?1
***Broke in /users/ud_71/sys/CTLG/c/CALLED.PGM at line 2
!
```

In the following example, the programmer executes the `PP` command to make the program stop when it calls TEST.SUB.2:

```
!PP TEST.SUB.2
```

SF

The debugger `SF` command opens the file *file.name* to *variable*.

You must use a variable name that is used in the compiled program. If you do not, the following error message results because the variable name is not listed in the symbol table:

```
Variable 'variable' is not found in the current program/subroutine.
```

Syntax

SF *variable* [-R][*filename* | DICT *file.name* | *absolutepathname*]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
variable	The name of the file variable.
-R	Sets the file to read-only mode. The default is read-write.
DICT	Indicates that the file being opened is a UniData dictionary file.
file.name	Can be one of the following: The non-UniData file being opened. The dictionary file name if a UniData dictionary file is being opened.
absolutepathname	The operating system path name to the non-UniData file, including the file name.

Examples

In the following example, the user attempts to open the INVENTORY file in the demo database to the file variable *inventory.carol*. The subsequent error message results because this variable is not in the symbol table for the program being debugged.

```
!SF inventory.sam INVENTORY
```

```
 ***Variable 'inventory.sam' is not found in the current program/subroutine.
```

The following command statement is successful because the variable used in the program is used:

```
 !SF INVENTORY.FILE INVENTORY
 !
```

SL

The debugger `SL` command selects record IDs in file *file.name* to the designated SELECT list number.

Note: The debugger `SL` command is equivalent to the `OSOPEN` statement in UniBasic.

Syntax

SL *select.list.no* [DICT | OS] *file.name*

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
select.list.no	The number of the select list to contain the record IDs.
DICT	Indicator that the file being accessed is a UniData hashed file defined by a UniData dictionary file.
OS	Indicator that the file being accessed is not a UniData hashed file.
file.name	The name of the file from which to read. This must be the actual filename, not the file variable. The file does not need to be accessed in the current program.

Examples

In the following example, the `SL` command selects record IDs in the demonstration database file INVENTORY to select list 1:

```
!SL 1 INVENTORY
```

In the next example, the `SL` command selects record IDs in TEST.FILE to SELECT LIST 0 and selects record IDs in DICT TEST.FILE to SELECT LIST 1:

```
 !SL 0 TEST.FILE
 !SL 1 DICT TEST.FILE
```

SO

The debugger `SO` command opens the operating system *os.file.path* to *file.variable*.

Syntax

SO *file.variable* [-R] *os.file.path*

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
file.variable	The file variable used in this program for the file to be opened.
-R	Indicator that the file is to be opened in read-only mode.
os.file.path	The operating system path to the file to be opened.

Example

In the following example, the `SO` commands open Mail/mymail to OS.FILE in read-only mode:

```
!SO OS.FILE -R Mail/mymail
```

SS

The debugger `SS` command opens the sequential file *file.name recordID* to *file.variable*.

This debugger command is equivalent to the `OPENSEQ` statement in UniBasic.

Syntax

```
SS file.variable [-R] file.name recordID
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
file.variable	The variable to store the record in this program.
-R	Indicator that the file is to be opened in read-only mode.
file.name	The name of the file being opened.
recordID	The primary key of the record saved to *file.variable*.

Example

In the following example, UniBasic opens the sequential file with *recordID* TEST.SUB.1 in *file.name* BP to *file.variable* SEQ.FILE in read-only mode:

```
!SS SEQ.FILE -R BP TEST.SUB.1
```

SV

The debugger `SV` command changes the value of a *variable* to the new value provided in "*string.*" UniData does not display the original value of *variable*.

Syntax

```
SV variable "string"
```

Parameters

The following table describes each parameter of the syntax.

Table 1: SV Parameters

Parameter	Description
variable	The name of the variable to be changed.
"*string*"	The value to assign to the variable.

Example

In the following example, the variable ACTIVE.LINE is changed to 1:

```
!SV ACTIVE.LINE "1"
!
```

SZ

The debugger `SZ` command displays the size of a *variable*.

Syntax

`SZ` *`variable`*

Example

In the following example, the `SL` command shows that the size of the variable MENU.ID is nine characters:

```
!SZ MENU.ID
size of 'MENU.ID' = 9
```

T

The debugger `T` command creates a new tracepoint. You can display variables at specific points in program execution or when the values of the variables change. You also can include additional conditions that must be satisfied before the tracepoint is executed.

Note: At a tracepoint, the debugger displays variable values without interrupting program execution.

Syntax

`T` *`variable`* `[?`*`condition`*`]` `[`*`occurrence.option`*`]` *`count`* `[-D` *`variable1`* `[,`*`variable2`*`]...`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
variable	The variable for which you want to create a tracepoint.

Parameter	Description
?condition	A condition that must be met to create a tracepoint. For example, if you specify ?X=7, UniBasic will not create a tracepoint until this condition is met.
-*occurrence.option*	A second condition that affects when the value of the variable will be displayed: ▪ -A – After the value changes *count* times. ▪ -E – Each time the value changes *count* times. For example, if *count* is set to 10, the value displays at the tenth, twentieth, thirtieth, and every tenth time the value changes. ▪ -U – Until the value changes the number of times specified in *count*. Occurrence options are mutually exclusive. You can use only one in any tracepoint.
count	Number of times the variable changes. Used in *occurrence.option*.
-D	Indicates that one or more variables are associated with this tracepoint. The system displays the value of the associated variable when the primary variable (and, if applicable, the tracepoint condition) is reached.
variable1...	Associated variable(s) that will display when the primary variable (and, if applicable, the tracepoint condition) is reached.

Example

In the following example, a trace is placed on the variable ACTIVE.LINE:

```
!T ACTIVE.LINE
```

The display of the value in the variable interrupts display of program output (if dual-terminal debugging is not used).

TC

The debugger `TC` command clears one or all tracepoints. To clear a particular tracepoint, include *index_num*, an entry for a tracepoint in the trace table. If you do not specify *index_num*, UniData enables all tracepoints.

Use the debugger `D` command to list breakpoints, tracepoints, and their index numbers.

Note: At a tracepoint, the debugger displays variable values without interrupting program execution. For more information about tracepoints, see Using breakpoints and tracepoints, on page 13.

Syntax

TC [*index_num*]

Example

In the following example, the debugger `TD` command displays the trace table, and then the `TC` command deletes the tracepoint by using the index number listed in the trace table. Finally, the `TD` command displays the trace table, which does not exist.

```
!TD
[1] BP/_UPDATE_ORDER: T COMMAND Enable
```

```
!TC 1
!TD
***No tracepoints are set
!
```

Related commands

Command	Description
BC	The debugger BC command removes one or all breakpoints. To clear a single breakpoint, include the index entry for that breakpoint in the break table. If you do not specify an index number, the debugger deletes all breakpoints. The debugger does not confirm deletion of the breakpoint.
TD	The debugger TD command displays the trace table, which lists all tracepoints, their status, and their index numbers. The trace table also provides the index number that is used in other debugger commands.

TD

The debugger TD command displays the trace table, which lists all tracepoints, their status, and their index numbers. The trace table also provides the index number that is used in other debugger commands.

Note: At a tracepoint, the debugger displays variable values without interrupting program execution. For more information about tracepoints, see Using breakpoints and tracepoints, on page 13.

For a complete description of the trace table, see D, on page 41.

Syntax

TD

Example

The following example demonstrates the TD command. In this example, two tracepoints are set for the program GADGETS, which resides in the directory NEWDEMO. The first was created by the debugger T command, and, therefore, is associated with a variable (INTROPROMPT). The second was created with the debugger TG command, and, therefore, is associated with a label (CHECKINPUT).

```
!TD
[1]  NEWDEMO/_GADGETS: T INTROPROMPT  Enable
[2]  NEWDEMO/_GADGETS: TG CHECKINPUT  Enable
```

Related commands

Command	Description
BD	The debugger BD command displays the break table, which lists all breakpoints for the current program. The break table lists the item number for each variable, which is required by other debugger commands.

TE

The debugger `TE` command enables one or all previously disabled tracepoints. To enable a particular tracepoint, include *index_num*, an entry for a tracepoint in the trace table. If you do not specify *index_num*, UniData enables all tracepoints.

Use the debugger `D` command to list breakpoints, tracepoints, and their index numbers.

Note: At a tracepoint, the debugger displays variable values without interrupting program execution. For more information about tracepoints, see Using breakpoints and tracepoints, on page 13.

Syntax

`TE` `[`*`index_num`*`]`

Example

In the following example, the debugger `TD` command displays the trace table, showing a disabled tracepoint on the COMMAND variable. `TE` enables the tracepoint, using the index number from the trace table. Finally, `TD` again displays the trace table, this time showing an enabled tracepoint for the COMMAND variable.

```
!TD
[1] BP/_UPDATE_ORDER: T COMMAND Disabled
!TE 1
!TD
[1] BP/_UPDATE_ORDER: T COMMAND Enable
```

Related commands

Command	Description
`BE`	The debugger `BE` command enables previously disabled breakpoints. *index_num* is an entry in the break table. If you do not specify an index entry, UniData enables all breakpoints.
`TD`	The debugger `TD` command displays the trace table, which lists all tracepoints, their status, and their index numbers. The trace table also provides the index number that is used in other debugger commands.

TG

The debugger `TG` command creates a tracepoint associated with a label.

At a tracepoint, the debugger displays variable values without interrupting program execution. For more information about tracepoints, see Using breakpoints and tracepoints, on page 13.

Syntax

```
TG label [?condition[occurrence.option] count [-D variable1
\[,variable2]...
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
label	The label for which you want to create a tracepoint.
?condition	A condition that must be met to create a tracepoint. For example, if you specify ?X=7, UniBasic will not create a tracepoint until this condition is met.
-occurrence.option	Specifies a second condition that affects when the value of the variable will be displayed: ▪ -A – After the value changes *count* times. ▪ -E – Each time the value changes *count* times. For example, if *count* is set to 10, the value displays at the tenth, twentieth, thirtieth, and every tenth time the value changes. ▪ -U – Until the value changes the number of times specified in *count*. Occurrence options are mutually exclusive. You can use only one in any tracepoint.
count	Provides a count used in *occurrence.option*.
-D	Indicates that one or more variables are associated with this tracepoint. The system displays the value of the associated variable when the primary variable (and, if applicable, the tracepoint condition) is reached.
variable1...	Specifies the associated variable(s) to display when the primary variable (and, if applicable, the tracepoint condition) is reached.

Example

In the following example, a tracepoint is associated with the label DISPLAY_DATA. First, the ORDER MAINTENANCE screen is displayed. The user responds with an order number. Then the debugger prints a message when the UPDATE_ORDER label is reached.

```
!TG DISPLAY_DATA
                                ORDER MAINTENANCE
                                    (Enter Q to quit)
Order #: 912
***In BP/_UPDATE_ORDER at line 114 label 'DISPLAY_DATA' reached
    Time:

    Client #:
    Product #:

    Color:
    Qty:
    Price:

**(Record Does Not Exist)**
Press the (Return) key.
```

Related command

Command	Description
BG	The debugger `BG` command creates a breakpoint associated with a label.

TL

The debugger `TL` command creates a tracepoint associated with a line.

Note: At a tracepoint, the debugger displays variable values without interrupting program execution. For more information about tracepoints, see Using breakpoints and tracepoints, on page 13.

Syntax

```
TL line [?condition[occurrence.option] count [-D variable1
\[,variable2]...
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
line	The line for which you want to create a tracepoint.
?condition	A condition that must be met to create a tracepoint. For example, if you specify ?X=7, UniBasic will not create a tracepoint until this condition is met.
-occurrence.option	Specifies a second condition that affects when the value of the variable will be displayed: ▪ -A – After the value changes *count* times. ▪ -E – Each time the value changes *count* times. For example, if *count* is set to 10, the value displays at the tenth, twentieth, thirtieth, and every tenth time the value changes. ▪ -U – Until the value changes the number of times specified in *count*. Occurrence options are mutually exclusive. You can use only one in any tracepoint.
count	Provides a count used in *occurrence.option*.
-D	Indicates that one or more variables are associated with this tracepoint. UniData displays the value of the associated variable when the primary variable (and, if applicable, the tracepoint condition) is reached.
variable1...	Specifies the associated variable(s) to display when the primary variable (and, if applicable, the tracepoint condition) is reached.

Example

In the following example, a tracepoint is associated with line 173 (the line that contains the RETURN from the subroutine DISPLAY_SCREEN).

```
!TL 183
```

When this line is executed, a message displays on the screen:

```
                                ORDER MAINTENANCE
                             (Enter Q to quit)
    Order #:
    Date:
    Time:

    Client #:
    Product #:

    Color:
    Qty:
```

```
    Price:
***In BP/_UPDATE_ORDER at line 183count
```

Related command

Command	Description
`BL`	The debugger `BL` command creates a breakpoint associated with a line.

TP

The debugger `TP` command defines a tracepoint associated with a program call.

Note: At a tracepoint, the debugger displays variable values without interrupting program execution.

Syntax

`TP` `{`*`file.name prog.name`*` | `*`cat.name`*`}`

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
file.name	Name of the directory file where the noncataloged program is stored.
prog.name	Name of the noncataloged program.
cat.name	Name of the cataloged program.

Example

In the following example, a program calls the CALLED.PGM subroutine:

```
PRINT "Enter operation to perform: (1)add, (2)delete, (3)update : ";INPUT operation
CALL CALLED.PGM(operation,ret.val)
PRINT "Operation completed: ":ret.val
END
```

The following example shows the CALLED.PGM subroutine:

```
SUBROUTINE CALLED.PGM(operation,ret.val)
BEGIN CASE
        CASE operation = 1
                ret.val = "Record added."
        CASE operation = 2
                ret.val = "Record deleted."
        CASE operation = 3
                ret.val = "Record updated."
END CASE
RETURN
```

In the following example, the programmer executes the `TP` command to establish a breakpoint when the CALLED.PGM subroutine is called:

```
!TP CALLED.PGM
```

```
***Trace program '/users/ud_71/sys/CTLG/c/CALLED.PGM'.
!
```

TU

The debugger `TU` command disables the tracepoint identified by an index entry in the break table. If you do not designate the index entry number, the debugger disables all index entries in the table. Disabled tracepoints remain in the trace table and can be enabled with the `TE` command.

For more information about tracepoints, see Using breakpoints and tracepoints, on page 13.

Syntax

TU [*index_num*]

Example

In the following example, the first line, a `TD` command, displays the trace table. The next debugger command, `TU`, disables the breakpoint associated with the CHECKINPUT label. The user in this example then executes the `TD` command to display the trace table, confirming that the tracepoint is disabled.

```
!TD
[1] NEWDEMO/_GADGETS: T INTROPROMPT Enable
[2] NEWDEMO/_GADGETS: TG CHECKINPUT Enable
!TU 2
!TD
[1] NEWDEMO/_GADGETS: T INTROPROMPT Enable
[2] NEWDEMO/_GADGETS: TG CHECKINPUT Disable
```

V

The debugger `V` command enables or disables visual mode. In visual mode, the debugger displays each line of code before it executes. You could find this command especially helpful when using the `E` command to execute a specific number of lines.

For more information about tracepoints, see Using breakpoints and tracepoints, on page 13.

Note: This command is most useful with dual-terminal debugging.

Syntax

V

W

The debugger `W` command sets watch on one or more variables. You can place an unlimited number of variables under a single watch command.

Tip: The debugger displays the value of watched variables whenever their values change without interrupting program execution. Setting watch on a variable has no effect on breakpoints or tracepoints associated with that or any other variable.

Syntax

W *variable1* [,*variable2*]...

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
variable1	The variable to display when its value changes.
variable2	Another variable to display when its value changes.

Example

In the following example, a program calls the CALLED.PGM subroutine:

```
PRINT "Enter operation to perform: (1)add, (2)delete, (3)update : ";INPUT operation
CALL CALLED.PGM(operation,ret.val)
PRINT "Operation completed: ":ret.val
END
```

The following example shows the CALLED.PGM subroutine:

```
SUBROUTINE CALLED.PGM(operation,ret.val)
BEGIN CASE
        CASE operation = 1
            ret.val = "Record added."
        CASE operation = 2
            ret.val = "Record deleted."
        CASE operation = 3
            ret.val = "Record updated."
END CASE
RETURN
```

The following example shows the symbol table for the preceding program:

```
!\*
Program Name : BP/_sub.call
Name                                    Type
operation                           variable
ret.val                                 variable
```

Watch is set on the two variables, and the program is executed with the debugger `N` command. The variables are displayed as the program executes.

```
!W operation,ret.val
!N
The following displays:
Enter operation to perform: (1)add, (2)delete, (3)update :
?1
***In BP/_sub.call at line 1 variable(s) changed:
operation = 1
***CALLED.PGM (/users/ud_71/sys/CTLG/c/CALLED.PGM) called
```

```
***In BP/_sub.call at line 2 variable(s) changed:
ret.val = Record added.
Operation completed: Record added.
:
```

WC

The debugger `WC` command clears the watch on specified variables. The default for this command is to clear all variables being watched.

Tip: The debugger displays the value of watched variables whenever their values change without interrupting program execution. Setting watch on a variable has no effect on breakpoints or tracepoints associated with that or any other variable.

Syntax

WC [*variable1* [,*variable2*]...]

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
variable1	The variable for which to clear the watch.
variable2	Another variable for which to clear the watch.

WD

The debugger `WD` command displays all variables under watch.

Tip: The debugger displays the value of watched variables whenever their values change without interrupting program execution. Setting watch on a variable has no effect on breakpoints or tracepoints associated with that or any other variable.

Syntax

WD

Example

In the following example, the `WD` command displays all watched variables:

```
!WD
[1] BP/_TEST: I
[2] BP/_TEST: TEST.ARRAY(2)
```

Z

The debugger `Z` command loads the symbol table, which is required to run the debugger.

Syntax

```
Z {file.name prog.name | cat.name}
```

Parameters

The following table describes each parameter of the syntax.

Parameter	Description
file.name	The name of the directory file where the noncataloged program is stored.
prog.name	The name of the noncataloged program.
cat.name	The name of the cataloged program.