ARM® DS-5™

Version 5.15

Debugger Command Reference

ARM DUI 0452O (ID061213)

ARM DS-5

Debugger Command Reference

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
June 2010	A	Non-Confidential	First release for DS-5
September 2010	B	Non-Confidential	Update for DS-5 version 5.2
November 2010	C	Non-Confidential	Update for DS-5 version 5.3
January 2011	D	Non-Confidential	Update for DS-5 version 5.4
May 2011	E	Non-Confidential	Update for DS-5 version 5.5
July 2011	F	Non-Confidential	Update for DS-5 version 5.6
September 2011	G	Non-Confidential	Update for DS-5 version 5.7
November 2011	H	Non-Confidential	Update for DS-5 version 5.8
February 2012	I	Non-Confidential	Update for DS-5 version 5.9
May 2012	J	Non-Confidential	Update for DS-5 version 5.10
July 2012	K	Non-Confidential	Update for DS-5 version 5.11
October 2012	L	Non-Confidential	Update for DS-5 version 5.12
December 2012	M	Non-Confidential	Update for DS-5 version 5.13
March 2013	N	Non-Confidential	Update for DS-5 version 5.14
June 2013	O	Non-Confidential	Update for DS-5 version 5.15

Proprietary Notice

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents
ARM DS-5 Debugger Command Reference

Chapter 1
Conventions and feedback

The following describes the typographical conventions and how to give feedback:

Typographical conventions

The following typographical conventions are used:

`monospace`	Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
`monospace` (with `mono` underlined)	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
`monospace italic`	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM® processor signal names.

Feedback on this product

If you have any comments and suggestions about this product, contact your supplier and give:

- your name and company

- the serial number of the product
- details of the release you are using
- details of the platform you are using, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tools, including the version number and build numbers.

Feedback on content

If you have comments on content then send an e-mail to `errata@arm.com`. Give:

- the title
- the number, ARM DUI 0452O
- if viewing online, the topic names to which your comments apply
- if viewing a PDF version of a document, the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

ARM periodically provides updates and corrections to its documentation on the ARM Information Center, together with knowledge articles and *Frequently Asked Questions* (FAQs).

Other information

- *ARM Information Center*, `http://infocenter.arm.com/help/index.jsp`
- *ARM Technical Support Knowledge Articles*, http://infocenter.arm.com/help/topic/com.arm.doc.faqs
- *Support and Maintenance*, http://www.arm.com/support/services/support-maintenance.php
- *ARM Glossary*, http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html.

Chapter 2
DS-5 Debugger commands

The following topics describe the DS-5 Debugger commands:

2.1 General syntax and usage of DS-5 Debugger commands

DS-5 Debugger commands are a comprehensive set of commands to debug embedded applications.

Syntax of DS-5 Debugger commands

Many commands accept arguments and flags using the following syntax:

`command [argument] [/flag]…`

A flag acts as an optional switch and is introduced with a forward slash character. Where a command supports flags, the flags are described as part of the command syntax.

Note

Commands are not case sensitive. Abbreviations are underlined.

Usage of DS-5 Debugger commands

The commands you submit to the debugger must conform to the following rules:

- Each command line can contain only one debugger command.
- When referring to symbols, you must use the same case as the source code.

You can execute the commands by entering them in the debugger command-line console or by running debugger script files. Alternatively in Eclipse, you can open the DS-5 Debug perspective where you can use the menus, icons, and toolbars provided or you can enter DS-5 Debugger commands in the Commands view.

You can enter many debugger commands in an abbreviated form. The debugger requires enough letters to uniquely identify the command you enter. Many commands have alternative names, or aliases, that you might find easier to remember. For example, `backtrace` and `where` are aliases for the `info stack` command. Command names and aliases can be abbreviated. For example, `info stack` can be abbreviated to `i s`. The syntax definition for each command shows how it can be abbreviated by underlining it for example, <u>i</u>nfo <u>s</u>tack.

In the syntax definition of each command:

- square brackets [...] enclose optional parameters
- braces {...} enclose required parameters
- a vertical pipe | indicates alternatives from which you must choose one
- parameters that can be repeated are followed by an ellipsis (...).

Do not type square brackets, braces, or the vertical pipe. Replace parameters in italics with the value you want. When you supply more than one parameter, use the separator as shown in the syntax definition for each command. If a parameter is a name that includes spaces, enclose it in double quotation marks.

Descriptive comments can be placed either at the end of a command or on a separate line. You can use the `#` character to identify a descriptive comment.

2.1.1 Using special characters and environment variables in paths

When specifying paths, you can use any of the following:

- a tilde character (~) at the start of a path to refer to your home directory
- an environment variable, for example:
  - `%LOG_DIRECTORY%`
  - `${LOG_DIRECTORY}`

  - `$LOG_DIRECTORY`
- a backslash (\) or forward slash (/) as a directory separator.

See also

- *set escapes-in-filenames* on page 2-159.

2.1.2 Using expressions

Some commands accept expressions. There are many types of expressions accepted by the debugger that enable you to extend the operation of a command. For example, binary mathematical expressions, references to module names, or calls to functions.

Using the $ character to access the content of registers and debugger variables

In an expression you can access the content of registers by using the $ character and the register name, for example:

```
print 4+$R0      # add 4 to the content of R0 register and print result
```

Results from the `print` commands are recorded in debugger variables. Other commands, such as breakpoint or watchpoint creating commands, the `start` command, and the `memory` command, also use debugger variables to record the ID of the new resource. Each of these debugger variables is assigned a number and can be used subsequently in expressions by using the $ character.

You can access print results or resource IDs using the debugger variables:

`$`	print result or ID in the last assigned debugger variable
`$$`	print result or ID in the second-to-last debugger variable
`$`*`n`*	print result or ID in the debugger variable with number *`n`*.

You can also use the following debugger variables:

`$cwd`	current working directory
`$cdir`	current compilation directory
`$entrypoint`	entry point of the current image
`$idir`	current image directory
`$sdir`	current script directory
`$datetime`	current date and time in string format
`$timems`	number of milliseconds since 1st Jan 1970.
`$pid`	current operating system process ID.
`$thread`	current thread ID for a multi-threaded application
`$core`	current processor ID for a *Symmetric MultiProcessing* (SMP) systems.
`$vmid`	current *Virtual Machine ID* (VMID) for systems that support hypervisor / virtual machine debugging.

Using built-in functions within expressions

In an expression you can use built-in functions to provide more functionality. The debugger supports the following:

`int strcmp(const char *str1, const char *str2);`

Compares two strings and returns an integer.

Return values are:

`<0`	Indicates that the second argument string value comes after the first argument string value in the machine collating sequences, `str1` < `str2`.
`0`	Indicates that the two strings are identical in content.
`>0`	Indicates that the first argument string value comes after the second argument string value in the machine collating sequences, `str2` < `str1`.

`int strncmp(const char *str1, const char *str2, size_t n);`

Compares at most `n` characters of two strings and returns an integer.

Return values are:

`<0`	Indicates that the second argument string value comes after the first argument string value in the machine collating sequences, `str1` < `str2`.
`0`	Indicates that the two strings are identical in content.
`>0`	Indicates that the first argument string value comes after the second argument string value in the machine collating sequences, `str2` < `str1`.

`char *strcpy(char *str1, const char *str2);`

Copies `str2` to `str1` including "`\0`" and returns `str1`.

`char *strncpy(char *str1, const char *str2, size_t n);`

Copies at most `n` characters of `str2` to `str1` including "`\0`" and returns `str1`. If `str2` has fewer than `n` characters then fill with "`\0`".

`void *memcpy(void *s, const void *cs, size_t n);`

Copies at most `n` characters from `cs` to `s` and returns `s`.

Example 2-1 Using a built-in `strcmp()` function with the break command

```
break main.c:45 if strcmp(myVar, "10") == 0     # Set conditional breakpoint that stops
                                                # when strings are identical
```

See also

- *break* on page 2-38
- *memory* on page 2-118
- *print, inspect* on page 2-134
- *set print* on page 2-163
- *show print* on page 2-195
- *start* on page 2-209
- *watch* on page 2-227.

2.1.3 Using wildcards

You can use wildcards to enhance your pattern matching. The following types of wildcard pattern matching can be used:

- Globs. This is the default.
- Regular expressions.

You can use the DS-5 Debugger command `set wildcard-style` to change the default setting.

Globs

Globs are a mechanism for examining the contents of strings, and can be used to search variables for strings matching specific patterns. Commands that support wildcards can use globs with the following syntax:

`*`	Specifies zero or more characters

?	Specifies only one character
\	Specifies an escape character to match on strings containing either * or ?
[*character*]	Specifies a range of characters. You can use !*character* to match characters that are not listed in the range.

Example 2-2 Globs where a wildcard is expected

```
info functions m*                # List all functions starting with m
```

Regular expressions

Commands that support wildcards can use regular expressions. The exact regular expression syntax supported is described in a book called *Mastering Regular Expressions*.

Example 2-3 Regular expressions where a wildcard is expected

```
info functions m.*               # List all functions starting with m
```

See also

- *set wildcard-style* on page 2-175
- *show wildcard-style* on page 2-206
- Jeffrey E. F. Friedl, *Mastering Regular Expressions*. ISBN 0-596-52812-4, http://oreilly.com.

2.1.4 Using regular expressions in the C expression parser

The C expression parser in the debugger supports regular expressions. Regular expressions are a mechanism for examining the contents of strings, and can be used to search variables for strings matching specific patterns.

The debugger extends C expression syntax to support regular expressions using the =~ and !~ operators in the style of Perl, as shown in the following examples:

Example 2-4 Regular expressions using the =~ and !~ operators

This example evaluates to 1 if the regular expression matches anywhere in the string and 0 if it does not match:

```
expression =~ regular_expression
```

This example evaluates to 0 if the regular expression matches anywhere in the string and 1 if it does not match:

```
expression !~ regular_expression
```

where:

expression	is any expression of type `char *` or `char[]`. For example, a variable name.

regular_expression	is a regular expression in the form `/`*`regex`*`/`*`modifiers`* or `m/`*`regex`*`/`*`modifiers`*. For example, if `str` is a variable of type char*, the following are valid expressions:
	`str =~ /abc/`
	`((char *) void_pointer) !~ m/abc/i`

The exact regular expression syntax supported is described by the *Mastering Regular Expressions* book in the chapter discussing Java regex support. An exception to this is the parsing of the handling of modifiers. The following modifiers are supported by the debugger:

`i`	enable case insensitive matching
`m`	multiline mode (`^` and `$` match embedded newline)
`s`	dotall mode (`.` matches line terminators)
`x`	comments mode (permit whitespace and comments).

See also

- Jeffrey E. F. Friedl, *Mastering Regular Expressions*. ISBN 0-596-52812-4, http://oreilly.com.

2.1.5 Using the C++ scoping resolution operator

In C++, the `::` (scope resolution) operator is a global identifier for variable or function names that are out of scope.

The expression evaluator supports scoping operations using the scope resolution, member and member pointer operators. This can be used to reference variables and functions within files, namespaces or classes.

For example:

Example 2-5 demo.cpp

```
static int FILE_STATIC_VARIABLE = 20;
class OuterClass
{
  public:
  OuterClass(int i)
  {
    value = i;
  }

  class InnerClass
  {
    public:
    int demoFunction()
    {
      return 25;
    }
  };

  void increment()
  {
    value++;
  }
  int value;

};
```

```
namespace NAME_SPACE_OUTER
{
  const int TEST_VAR= 20;
  namespace NAME_SPACE_INNER
  {
    const int TEST_VAR= 19;
    int nameSpaceFoo ()
    {
      return 60;
    }
  };
};

int main()
{
  OuterClass oc(14);
  OuterClass *ptr_oc = &oc;

  ptr_oc->increment();
}
```

You can query this example by using any of the following expressions:

```
OuterClass::InnerClass::demoFunction
"demo.c"::FILE_STATIC_VARIABLE
NAME_SPACE_OUTER::TEST_VAR
NAME_SPACE_OUTER::NAME_SPACE_INNER::TEST_VAR
```

If you set a a breakpoint at `ptr_oc->increment()` and run to it, then the following expressions can also be used to query the instances of the outer class:

```
oc.value
ptr_oc->value
```

2.1.6 `printf()` style format string

Certain commands use `printf()` style format strings to specify how to format values. For example the `set print double-format` and `set print float-format` commands specify how to format floating-point values. It works in a similar way to the ANSI C standard library function `printf()`.

Format string syntax

The commands specify the format using a string. If there are no `%` characters in the string, the message is written out and any arguments are ignored. The `%` symbol is used to indicate the start of an argument conversion specification.

The syntax of the format string is:

`%[`*`flag...`*`][`*`fieldwidth`*`][`*`precision`*`]`*`format`*

where:

`flag` An optional conversion modification flag.

`"-"`	result is left-justified
`"#"`	result uses a conversion-dependent alternate form
`"+"`	result includes a sign
`" "`	result includes a leading space for positive values
`"0"`	result is zero-padded
`","`	result includes locale-specific grouping separator
`"("`	result encloses negative numbers in parentheses.

`fieldwidth` An optional minimum field width specified in decimal.

`precision` An optional precision specified in decimal, with a preceding `.` (period character) to identify it.

`format` The possible conversion specifier characters are:

%	A literal `%` character.
a, A, e, E, f, g, or G	Results in a decimal number formatted using scientific notation or floating point notation. The capital letter forms use a capital `E` in scientific notation rather than an `e`.
d,or u	Results in a decimal integer. **`d`** indicates a signed integer. **`u`** indicates an unsigned integer.
h, H	Results in a Hexadecimal character in lower or upper case.
x or X	Results in an unsigned Hexadecimal character in lower or upper case.
o	Results in an octal integer.
c or C	Results in a Unicode character in lower or upper case.
s	Results in a string.
b or B	Results in a string containing either "true" or "false" in lower or upper case.
n	Results in a platform-specific line separator.
t or T	Prefix for date and time conversion specifier characters. For example: `"%ta %tb %td %tT" results in "Sun Jul 20 16:17:00"`

See also

- *echo* on page 2-63
- *output* on page 2-131
- *print, inspect* on page 2-134
- *set print* on page 2-163
- *show print* on page 2-195.

2.2 DS-5 Debugger commands listed in groups

The DS-5 Debugger commands grouped according to specific tasks are:

- *Breakpoints and watchpoints*
- *Execution control* on page 2-12
- *Scripts* on page 2-14
- *Call stack* on page 2-15
- *Operating System (OS)* on page 2-15
- *Files* on page 2-17
- *Data* on page 2-18
- *Memory* on page 2-19
- *Registers* on page 2-20
- *Display* on page 2-20
- *Information* on page 2-20
- *Log commands* on page 2-22
- *Set commands* on page 2-22
- *Show commands* on page 2-23
- *Flash commands* on page 2-25
- *Supporting commands* on page 2-25.

2.2.1 Breakpoints and watchpoints

List of commands:

break on page 2-38

Sets a software breakpoint.

hbreak on page 2-74

Sets a hardware breakpoint.

tbreak on page 2-216

Sets a temporary software breakpoint that is deleted when it is hit.

thbreak on page 2-218

Sets a temporary hardware breakpoint that is deleted when it is hit.

awatch on page 2-35

Sets a read/write watchpoint for a global/static data symbol.

rwatch on page 2-142

Sets a read watchpoint for a global/static data symbol.

watch on page 2-227

Sets a write watchpoint for a global/static data symbol.

condition on page 2-48

Sets a break condition for a specific breakpoint or watchpoint.

ignore on page 2-78

Sets the ignore counter for a breakpoint or watchpoint condition.

break-script on page 2-40

Assigns a script file to a specific breakpoint for execution when the breakpoint is triggered.

break-stop-on-threads, break-stop-on-cores on page 2-43

Applies an existing breakpoint to one or more threads or processors.

break-stop-on-vmid on page 2-44

Applies an existing hardware breakpoint to a virtual machine.

enable breakpoints on page 2-64

Enables one or more breakpoints or watchpoints by number.

disable breakpoints on page 2-55

Disables one or more breakpoints or watchpoints by number.

delete breakpoints on page 2-52

Deletes one or more breakpoints or watchpoints by number.

resolve on page 2-139

Resolves one or more breakpoints or watchpoints.

clear on page 2-46

Deletes a breakpoint at a specific location.

clearwatch on page 2-47

Deletes a watchpoint at a specific location.

info breakpoints, info watchpoints on page 2-81

Displays information about the status of all breakpoints and watchpoints.

info breakpoints capabilities, info watchpoints capabilities on page 2-82

Displays a list of parameters that you can use with breakpoint and watchpoint commands for the current connection.

set breakpoint on page 2-151

Controls the automatic behavior of breakpoints and watchpoints.

silence on page 2-207

Disables the printing of stop messages for a specific breakpoint.

unsilence on page 2-223

Enables the printing of stop messages for a specific breakpoint.

Type `help` followed by a command name for more information on a specific command.

2.2.2 Execution control

List of commands:

start on page 2-209

Sets a temporary breakpoint and starts running the image until it hits the breakpoint. When the debugger stops, the breakpoint is deleted. By default, the breakpoint is set at the address of the global function `main()`.

set blocking-run-control on page 2-150

Controls whether run control operations such as stepping and running are blocked until the target stops or released immediately.

show blocking-run-control on page 2-183

Displays the current setting for blocking run control operations.

set debug-from on page 2-154

Specifies the address of the temporary breakpoint for subsequent use by the `start` command.

show debug-from on page 2-187

Displays the current setting for the expression that is used by the `start` command to set a temporary breakpoint.

run on page 2-141

Starts running the target.

continue on page 2-49

Continues running the target.

advance on page 2-33

Sets a temporary breakpoint and continues running the image until it hits the breakpoint. When the debugger stops, the breakpoint is deleted.

finish on page 2-70

Continues running the device to the next instruction after the selected stack frame finishes.

interrupt, stop on page 2-111

Interrupts the target and stops the current application if it is running.

wait on page 2-226

Instructs the debugger to wait until either the application completes or a breakpoint is hit.

reset on page 2-138

Performs a reset on the target.

step on page 2-211

Source level stepping including stepping into all function calls where there is debug information.

stepi on page 2-212

Instruction level stepping including stepping into all function calls.

steps on page 2-213

Source level stepping through statements including stepping into all function calls where there is debug information.

next on page 2-127

Source level stepping over all function calls.

nexti on page 2-128

Instruction level stepping over all function calls.

nexts on page 2-129	Source level stepping through statements but stepping over all function calls.
thread, core on page 2-220	Displays information about the current thread or processor.
thread apply, core apply on page 2-221	Temporarily switches control to a thread or processor to execute a DS-5 Debugger command and then switches back to the original state.
set step-mode on page 2-169	Specifies whether to step into or step over a function with no debug information.
show step-mode on page 2-200	Displays the current step setting for functions without debug information.
handle on page 2-73	Controls the handler settings for one or more signals or exceptions.
info signals, info handle on page 2-102	Displays information about the handling of signals.

Type `help` followed by a command name for more information on a specific command.

2.2.3 Scripts

List of commands:

define on page 2-51	Enables you to derive a new user-defined command from existing commands.
document on page 2-59	Enables you to add integrated help for a new user-defined command.
newvar on page 2-126	Declares and initializes a new debugger convenience variable.
if on page 2-77	Enables you to write scripts that conditionally execute debugger commands.
while on page 2-231	Enables you to write looping scripts that conditionally execute debugger commands.
end on page 2-66	Enables you to terminate conditional scripts.
source on page 2-208	Loads and runs a script file containing debugger commands to control and debug your target.

Type `help` followed by a command name for more information on a specific command.

2.2.4 Call stack

List of commands:

up on page 2-224
: Controls and displays the current position in the call stack.

up-silently on page 2-225
: Controls the current position in the call stack.

down on page 2-60
: Controls and displays the current position in the call stack.

down-silently on page 2-61
: Controls the current position in the call stack.

frame on page 2-72
: Displays stack frame information at the selected position.

select-frame on page 2-144
: Controls the current position in the call stack.

info frame on page 2-88
: Displays stack frame information at the selected position.

info stack, backtrace, where on page 2-104
: Displays information about the call stack.

set backtrace on page 2-149
: Controls the default behavior when using the `info stack` command.

show backtrace on page 2-182
: Displays current behavior settings for use with the `info stack` command.

Type `help` followed by a command name for more information on a specific command.

2.2.5 Operating System (OS)

List of commands:

sharedlibrary on page 2-176
: Loads shared library symbols.

nosharedlibrary on page 2-130
: Discards all loaded shared library symbols except for the symbols that are loaded explicitly using the `sharedlibrary` command.

info sharedlibrary on page 2-101
: Displays the names of the loaded shared libraries.

set os on page 2-161
: Controls the OS settings in the debugger.

show os on page 2-194
: Displays the current OS settings in the debugger.

set sysroot, set solib-absolute-prefix on page 2-172

Specifies the system root for prefixing shared library paths.

show sysroot, show solib-absolute-prefix on page 2-203

Displays the system root directory in use by the debugger when searching for shared library symbols.

set auto-solib-add on page 2-148

Controls the automatic loading of shared library symbols.

show auto-solib-add on page 2-181

Displays the current automatic setting for use when loading shared library symbols.

set solib-search-path on page 2-168

Specifies additional directories to search for shared library symbols.

show solib-search-path on page 2-199

Displays the current search paths in use by the debugger when searching for shared libraries.

set stop-on-solib-events on page 2-170

Specifies whether the debugger stops execution when it is notified of an event by the dynamic linker.

show stop-on-solib-events on page 2-201

Displays the current debugger setting that controls whether execution stops when shared library events occur.

thread, core on page 2-220

Sets the current thread and displays thread state and stack frame.

thread apply, core apply on page 2-221

Temporarily switches control to a thread or processor to execute a DS-5 Debugger command and then switches back to the original state.

info threads on page 2-107

Displays a list of threads showing ID, current state and related stack frame information.

info processes on page 2-98

Displays a list of processes showing ID, current state and related stack frame information.

info os-log on page 2-95

Displays the contents of the *Operating System* (OS) log buffer for connections that supports this feature.

info os-modules on page 2-96

Displays a list of the *Operating System* (OS) modules for connections that supports this feature.

info os-version on page 2-97

Displays the version of the *Operating System* (OS) for connections that supports this feature.

Type `help` followed by a command name for more information on a specific command.

2.2.6 Files

List of commands:

load on page 2-114

Loads an image on to the target and records the entry point address for future use by the `run` and `start` commands.

loadfile on page 2-115

Loads debug information into the debugger, an image on to the target and records the entry point address for future use by the `run` and `start` commands.

file, symbol-file on page 2-68

Loads debug information from an image into the debugger.

reload-symbol-file on page 2-137

Reloads debug information from an already loaded image into the debugger using the same settings as the original load operation.

add-symbol-file on page 2-31

Loads additional debug information into the debugger.

discard-symbol-file on page 2-58

Discards debug information relating to a specific file.

dump on page 2-62

Reads data from memory or an expression and writes to a file.

append on page 2-34

Reads data from memory or an expression and appends to an existing file.

restore on page 2-140

Reads data from a file and writes it to memory.

info files, info target on page 2-86

Displays information about the loaded image and symbols.

info sources on page 2-103

Displays the names of the source files.

cd on page 2-45

Sets the working directory.

pwd on page 2-135

Displays the working directory.

directory on page 2-54

Defines additional directories to search for source files.

show directories on page 2-188

Displays the list of directories to search for source files.

set substitute-path on page 2-171

Modifies the search paths used when displaying source code.

show substitute-path on page 2-202

Displays the current search path substitution rules in use by the debugger when searching for source files.

Type `help` followed by a command name for more information on a specific command.

2.2.7 Data

List of commands:

list on page 2-112

Displays lines of source code.

set listsize on page 2-160

Modifies the default number of source lines that the `list` command displays.

show listsize on page 2-193

Displays the number of source lines that the `list` command displays.

set variable on page 2-174

Specifies an expression and assigns the result to a variable.

whatis on page 2-229

Displays the data type of an expression.

x on page 2-232

Displays the content of memory at a specific address.

disassemble on page 2-57

Displays disassembly for a specific section of memory.

info address on page 2-79

Displays the location of a symbol.

info symbol on page 2-105

Displays the symbol name at a specific address.

info locals on page 2-92

Displays all local variables.

info functions on page 2-89

Displays the name and data types for all functions.

info variables on page 2-108

Displays the name and data types of global and static variables.

info classes on page 2-84

Displays C++ class names.

info members on page 2-93

Displays the name and data types for all class member variables that are accessible in the function corresponding to the selected stack frame.

Type `help` followed by a command name for more information on a specific command.

2.2.8 Memory

List of commands:

memory on page 2-118

Specifies the attributes and size for a memory region.

memory auto on page 2-120

Resets the memory regions to the default target settings.

memory debug-cache on page 2-121

Controls the caching by the debugger for all memory regions.

enable memory on page 2-65

Enables one or more user-defined memory regions.

disable memory on page 2-56

Disables one or more user-defined memory regions.

delete memory on page 2-53

Deletes one or more user-defined memory regions.

info memory on page 2-94

Displays the attributes for all memory regions.

memory fill on page 2-122

Writes a specific pattern of bytes to memory.

memory set on page 2-123

Writes to memory.

memory set_typed on page 2-125

Writes a list of values to memory.

dump on page 2-62

Reads data from memory or an expression and writes to a file.

append on page 2-34

Reads data from memory or an expression and appends to an existing file.

restore on page 2-140

Reads data from a file and writes it to memory.

x on page 2-232

Displays the content of memory at a specific address.

disassemble on page 2-57

Displays disassembly for a specific section of memory.

Type `help` followed by a command name for more information on a specific command.

2.2.9 Registers

List of commands:

info registers on page 2-99

Displays the name and content of registers for the current stack frame.

info all-registers on page 2-80

Displays the name and content of grouped registers for the current stack frame.

Type `help` followed by a command name for more information on a specific command.

2.2.10 Display

List of commands:

echo on page 2-63

Displays only textual strings.

output on page 2-131

Displays only the output of an expression.

print, inspect on page 2-134

Displays the output of an expression and records the result in a debugger variable.

set print on page 2-163

Controls the current debugger print settings.

show print on page 2-195

Displays the current debugger print settings.

Type `help` followed by a command name for more information on a specific command.

2.2.11 Information

List of commands:

info address on page 2-79

Displays the location of a symbol.

info all-registers on page 2-80

Displays the name and content of all registers.

info breakpoints, info watchpoints on page 2-81

Displays information about the status of all breakpoints and watchpoints.

info capabilities on page 2-83

Displays a list of capabilities for the target device that is currently connected to the debugger.

info classes on page 2-84

Displays C++ class names.

info cores on page 2-85

Displays information about the running processors.

info files, info target on page 2-86

Displays information about the loaded image and symbols.

info frame on page 2-88

Displays stack frame information at the selected position.

info functions on page 2-89

Displays the name and data types for all functions.

info inst-sets on page 2-91

Displays the available instruction sets.

info locals on page 2-92

Displays all local variables for the current stack frame.

info members on page 2-93 Displays the name and data types for class member variables.

info memory on page 2-94

Displays the attributes for all memory regions.

info os-log on page 2-95

Displays the contents of the *Operating System* (OS) log buffer for connections that support this feature.

info os-modules on page 2-96

Displays a list of loadable kernel modules for connections that support this feature.

info os-version on page 2-97

Displays the version of the *Operating System* (OS) for connections that support this feature.

info processes on page 2-98

Displays information about the user space processes.

info registers on page 2-99

Displays the name and content of all application level registers.

info semihosting on page 2-100

Displays semihosting information for the server, client, or all.

info sharedlibrary on page 2-101

Displays the names of the loaded shared libraries.

info signals, info handle on page 2-102

Displays information about the handling of signals or exceptions.

info sources on page 2-103

Displays the names of the source files.

info stack, backtrace, where on page 2-104

Displays information about the call stack.

info symbol on page 2-105

Displays the symbol name at a specific address.

info threads on page 2-107

Displays information about the available threads.

info variables on page 2-108

Displays the name and data types for all global and static variables.

Type `help` followed by a command name for more information on a specific command.

2.2.12 Log commands

List of commands:

log config on page 2-116

Specifies the type of logging configuration to output runtime messages from the debugger.

log file on page 2-117

Specifies an output file to receive runtime messages from the debugger.

Type `help` followed by a command name for more information on a specific command.

2.2.13 Set commands

List of commands:

set `set` is an alias for `set variable`.

set arm on page 2-146

Controls the behavior of the debugger when selecting the instruction set for disassembly and setting breakpoints.

set auto-solib-add on page 2-148

Controls the automatic loading of shared library symbols.

set backtrace on page 2-149

Controls the default behavior when using the `info stack` command.

set blocking-run-control on page 2-150

Controls whether run control operations such as stepping and running are blocked until the target stops or released immediately.

set breakpoint on page 2-151

Controls the automatic behavior of breakpoints and watchpoints.

set case-insensitive-source-matching on page 2-152

Controls the case sensitivity when the debugger performs source file matching operations.

set debug-from on page 2-154

Specifies the address of the temporary breakpoint for subsequent use by the `start` command.

set endian on page 2-157

Specifies the byte order for use by the debugger.

set escape-strings on page 2-158

Controls how special characters in strings are printed on the debugger command-line.

set escapes-in-filenames on page 2-159

Controls the use of special characters in paths.

set listsize on page 2-160

Modifies the default number of source lines that the `list` command displays.

set os on page 2-161

Controls the *Operating System* (OS) settings in the debugger.

set print on page 2-163

Controls the current debugger print settings.

set semihosting on page 2-165

Controls the semihosting operations in the debugger.

set solib-search-path on page 2-168

Specifies additional directories to search for shared library symbols.

set step-mode on page 2-169

Specifies whether to step into or step over a function with no debug information.

set stop-on-solib-events on page 2-170

Specifies whether the debugger stops execution when it is notified of an event by the dynamic linker.

set substitute-path on page 2-171

Modifies the search paths used when displaying source code.

set sysroot, set solib-absolute-prefix on page 2-172

Specifies the system root for prefixing shared library paths.

set variable on page 2-174

Specifies an expression and assigns the result to a variable.

set wildcard-style on page 2-175

Specifies the wildcard style to use for pattern matching in strings.

Type `help` followed by a command name for more information on a specific command.

2.2.14 Show commands

List of commands:

show on page 2-178

Displays the current debugger settings.

show architecture on page 2-179

Displays the current target architecture.

show arm on page 2-180

Displays the current instruction set settings in use by the debugger for disassembly and setting breakpoints.

show auto-solib-add on page 2-181

Displays the current automatic setting for use when loading shared library symbols.

show backtrace on page 2-182

Displays the current behavior settings for use with the `info stack` command.

show blocking-run-control on page 2-183

Displays the current setting for blocking run control operations.

show breakpoint on page 2-184

Displays the current breakpoint and watchpoint behavior settings.

show case-insensitive-source-matching on page 2-185

Displays the current breakpoint and watchpoint behavior settings.

show debug-from on page 2-187

Displays the current setting for the address of the temporary breakpoint used by the `start` command.

show directories on page 2-188

Displays the list of search directories.

show endian on page 2-190

Displays the current byte order setting.

show escape-strings on page 2-191

Displays the current setting for controlling how special characters in strings are printed on the debugger command-line.

show escapes-in-filenames on page 2-192

Displays the current setting for controlling the use of special characters in paths.

show listsize on page 2-193

Displays the listing size for the list command.

show os on page 2-194 Displays the current *Operating System* (OS) settings in the debugger.

show print on page 2-195

Displays the current debugger print settings.

show semihosting on page 2-196

Displays the current setting for semihosting operations.

show solib-search-path on page 2-199

Displays the current search path for shared libraries.

show step-mode on page 2-200
: Displays the current step setting for functions without debug information.

show stop-on-solib-events on page 2-201
: Displays the current debugger setting that controls whether execution stops when shared library events occur.

show substitute-path on page 2-202
: Displays all the substitution rules.

show sysroot, show solib-absolute-prefix on page 2-203
: Displays the system root prefix for shared library paths.

show version on page 2-205
: Displays the current version number of the debugger.

show wildcard-style on page 2-206
: Displays the current wildcard style in use for pattern matching.

Type `help` followed by a command name for more information on a specific command.

2.2.15 Flash commands

List of commands:

flash load on page 2-71
: Loads sections from an image into one or more flash devices.

info flash on page 2-87
: Displays information about the flash devices on the current target.

Type `help` followed by a command name for more information on a specific command.

2.2.16 Supporting commands

List of commands:

preprocess on page 2-133
: Displays a preprocessed value.

help on page 2-76
: Displays help information for a specific command or a group of commands listed according to specific debugging tasks.

pause on page 2-132
: Pauses the execution of a script for a specified period of time.

shell on page 2-177
: Runs a shell command within the current debug session.

quit, exit on page 2-136
: Quits the debugger session.

show version on page 2-205
: Displays the current version number of the debugger.

show architecture on page 2-179

Displays the architecture of the current target.

set arm on page 2-146

Controls the behavior of the debugger when selecting the instruction set for disassembly and setting breakpoints.

show arm on page 2-180

Displays the current instruction set settings in use by the debugger for disassembly and setting breakpoints.

info inst-sets on page 2-91

Displays the available instruction sets.

set endian on page 2-157

Specifies the byte order for use by the debugger.

show endian on page 2-190

Displays the current byte order setting in use by the debugger.

info capabilities on page 2-83

Displays a list of capabilities for the target device that is currently connected to the debugger.

set semihosting on page 2-165

Controls the semihosting options in the debugger.

show semihosting on page 2-196

Displays the current semihosting settings.

stdin on page 2-210

Specifies semihosting input requested by application code. For use only in a command-line console with interactive mode.

unset on page 2-222

Modifies the current debugger settings.

Type `help` followed by a command name for more information on a specific command.

2.3 DS-5 Debugger commands listed in alphabetical order

The DS-5 Debugger commands in alphabetical order are:

- *add-symbol-file* on page 2-31
- *advance* on page 2-33
- *append* on page 2-34
- *awatch* on page 2-35
- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *cd* on page 2-45
- *clear* on page 2-46
- *clearwatch* on page 2-47
- *condition* on page 2-48
- *continue* on page 2-49
- *define* on page 2-51
- *delete breakpoints* on page 2-52
- *delete memory* on page 2-53
- *directory* on page 2-54
- *disable breakpoints* on page 2-55
- *disable memory* on page 2-56
- *disassemble* on page 2-57
- *discard-symbol-file* on page 2-58
- *document* on page 2-59
- *down* on page 2-60
- *down-silently* on page 2-61
- *dump* on page 2-62
- *echo* on page 2-63
- *enable breakpoints* on page 2-64
- *enable memory* on page 2-65
- *end* on page 2-66
- *file, symbol-file* on page 2-68
- *finish* on page 2-70
- *flash load* on page 2-71
- *frame* on page 2-72
- *handle* on page 2-73
- *hbreak* on page 2-74
- *help* on page 2-76
- *if* on page 2-77
- *ignore* on page 2-78
- *info address* on page 2-79
- *info all-registers* on page 2-80
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *info capabilities* on page 2-83
- *info classes* on page 2-84
- *info cores* on page 2-85

- *info files, info target* on page 2-86
- *info flash* on page 2-87
- *info frame* on page 2-88
- *info functions* on page 2-89
- *info inst-sets* on page 2-91
- *info locals* on page 2-92
- *info memory* on page 2-94
- *info members* on page 2-93
- *info os-log* on page 2-95
- *info os-modules* on page 2-96
- *info os-version* on page 2-97
- *info processes* on page 2-98
- *info registers* on page 2-99
- *info semihosting* on page 2-100
- *info sharedlibrary* on page 2-101
- *info signals, info handle* on page 2-102
- *info sources* on page 2-103
- *info stack, backtrace, where* on page 2-104
- *info symbol* on page 2-105
- *info target* on page 2-106
- *info threads* on page 2-107
- *info variables* on page 2-108
- *interrupt, stop* on page 2-111
- *list* on page 2-112
- *load* on page 2-114
- *loadfile* on page 2-115
- *log config* on page 2-116
- *log file* on page 2-117
- *memory* on page 2-118
- *memory auto* on page 2-120
- *memory debug-cache* on page 2-121
- *memory fill* on page 2-122
- *memory set* on page 2-123
- *memory set_typed* on page 2-125
- *newvar* on page 2-126
- *next* on page 2-127
- *nexti* on page 2-128
- *nexts* on page 2-129
- *nosharedlibrary* on page 2-130
- *output* on page 2-131
- *pause* on page 2-132
- *preprocess* on page 2-133
- *print, inspect* on page 2-134
- *pwd* on page 2-135
- *quit, exit* on page 2-136
- *reload-symbol-file* on page 2-137
- *reset* on page 2-138

- *resolve* on page 2-139
- *restore* on page 2-140
- *run* on page 2-141
- *rwatch* on page 2-142
- *select-frame* on page 2-144
- *set arm* on page 2-146
- *set auto-solib-add* on page 2-148
- *set backtrace* on page 2-149
- *set blocking-run-control* on page 2-150
- *set breakpoint* on page 2-151
- *set case-insensitive-source-matching* on page 2-152
- *set debug-agent* on page 2-153
- *set debug-from* on page 2-154
- *set directories* on page 2-155
- *set dtsl-options* on page 2-156
- *set endian* on page 2-157
- *set escape-strings* on page 2-158
- *set escapes-in-filenames* on page 2-159
- *set listsize* on page 2-160
- *set os* on page 2-161
- *set print* on page 2-163
- *set semihosting* on page 2-165
- *set solib-search-path* on page 2-168
- *set step-mode* on page 2-169
- *set stop-on-solib-events* on page 2-170
- *set substitute-path* on page 2-171
- *set sysroot, set solib-absolute-prefix* on page 2-172
- *set trust-ro-sections-for-opcodes* on page 2-173
- *set variable* on page 2-174
- *set wildcard-style* on page 2-175
- *sharedlibrary* on page 2-176
- *shell* on page 2-177
- *show architecture* on page 2-179
- *show arm* on page 2-180
- *show auto-solib-add* on page 2-181
- *show backtrace* on page 2-182
- *show blocking-run-control* on page 2-183
- *show breakpoint* on page 2-184
- *show case-insensitive-source-matching* on page 2-185
- *show debug-agent* on page 2-186
- *show debug-from* on page 2-187
- *show directories* on page 2-188
- *show dtsl-options* on page 2-189
- *show endian* on page 2-190
- *show escape-strings* on page 2-191
- *show escapes-in-filenames* on page 2-192
- *show listsize* on page 2-193

- *show os* on page 2-194
- *show print* on page 2-195
- *show semihosting* on page 2-196
- *show solib-search-path* on page 2-199
- *show step-mode* on page 2-200
- *show stop-on-solib-events* on page 2-201
- *show substitute-path* on page 2-202
- *show sysroot, show solib-absolute-prefix* on page 2-203
- *show trust-ro-sections-for-opcodes* on page 2-204
- *show version* on page 2-205
- *show wildcard-style* on page 2-206
- *silence* on page 2-207
- *source* on page 2-208
- *start* on page 2-209
- *stdin* on page 2-210
- *step* on page 2-211
- *stepi* on page 2-212
- *steps* on page 2-213
- *tbreak* on page 2-216
- *thbreak* on page 2-218
- *thread, core* on page 2-220
- *thread apply, core apply* on page 2-221
- *unset* on page 2-222
- *unsilence* on page 2-223
- *up* on page 2-224
- *up-silently* on page 2-225
- *wait* on page 2-226
- *watch* on page 2-227
- *whatis* on page 2-229
- *while* on page 2-231
- *x* on page 2-232.

2.3.1 `add-symbol-file`

This command loads additional debug information into the debugger.

Syntax

`add-symbol-file` *`filename`* `[`*`offset`*`] [-`*`option`*`] [-s` *`section address`*`]…`

Where:

`filename`	Specifies the image, shared library, or *Operating System* (OS) module. **Note** Shared library and OS modules depend on connections that support loading these types of files. This option pends the file until the library or OS module is loaded.
`offset`	Specifies the offset that is added to all addresses within the image. If *`offset`* is not specified then the default for: • An image is zero. • A shared library is the load address of the library. If the application has not currently loaded the specified library then the request is pended until the library is loaded and the offset can be determined.
`option`	Controls how debug information is loaded: `readnow` Specifies loading all debug information immediately. This option uses more memory and is slower to load but it enables faster debugging. `demandload` Specifies loading debug information when required by the debugger. This option enables a faster load and uses less memory but debugging might be slower. This is the default.
`s`	Specifies the relocation of symbols being loaded from a relocatable object file.
`section`	Specifies the name of a section in a relocatable file.
`address`	Specifies the address of the section. This can be either an address or an expression that evaluates to an address.

You can use the `info files` command to display information about the loaded files.

Example

Example 2-6 `add-symbol-file`

```
add-symbol-file myFile.axf           # Load symbols at entry point+0x0000
add-symbol-file myLib.so             # Pends symbol file for shared library
add-symbol-file myModule.ko          # Pends symbol file for OS module
add-symbol-file myFile.axf 0x2000    # Load symbols at entry point+0x2000
add-symbol-file relocate.o -s .text 0x1000 -s .data 0x2000
                                     # Load symbols from relocate.o and relocate
                                     # symbols defined in .text or .data sections
```

See also

- *cd* on page 2-45

- *discard-symbol-file* on page 2-58
- *file, symbol-file* on page 2-68
- *load* on page 2-114
- *info files, info target* on page 2-86
- *info os-modules* on page 2-96
- *loadfile* on page 2-115
- *reload-symbol-file* on page 2-137.

2.3.2 advance

This command sets a temporary breakpoint and calls the debugger `continue` command. The temporary breakpoint is subsequently deleted when it is hit.

Note

Control is returned as soon as the target is running. You can use the `wait` command to block the debugger from returning control until either the application completes or a breakpoint is hit.

Syntax

`advance [-p] [`*`filename`*`:]`*`location`*`|`*`*address`*

Where:

`p`	Specifies whether or not the resolution of an unrecognized breakpoint location results in a pending breakpoint being created.
`filename`	Specifies the file.
`location`	Specifies the location:
	`line_num` is a line number
	`function` is a function name.
	`label` is a label name.
	`+offset``|`*`-offset`* Specifies the line offset from the current location.
`address`	Specifies the address. This can be either an address or an expression that evaluates to an address.

Example

Example 2-7 advance

```
advance func1       # Sets a temporary breakpoint at func1 and continues
                    # running the target
advance -p lib.c:20 # Sets a pendable temporary breakpoint at line 20 in lib.c
                    # and continues running the target
```

See also

- *continue* on page 2-49
- *tbreak* on page 2-216.

2.3.3 append

This command reads data from memory or the result of an expression and appends it to an existing file.

Syntax

`append [format] memory filename start_address {end_address|+size}`

`append [format] value filename expression`

Where:

format	Specifies the output format: `binary` Binary. This is the default. `ihex` Intel Hex-32. `srec` Motorola 32-bit (S-records). `vhx` Byte oriented hexadecimal (Verilog Memory Model).
filename	Specifies the file.
start_address	Specifies the start address for the memory.
end_address	Specifies the inclusive end address for the memory.
size	Specifies the size of the region.
expression	Specifies an expression that is evaluated and the result is returned.

Example

Example 2-8 append

```
append memory myFile.bin 0x8000 0x8FFF  # Append content of memory 0x8000-0x8FFF
                                        # to binary file myFile.bin
append srec value myFile.m32 myArray    # Append content of myArray to
                                        # Motorola 32-bit file myFile.m32
```

See also

- *Using expressions* on page 2-4
- *dump* on page 2-62
- *restore* on page 2-140.

2.3.4 awatch

This command sets a watchpoint for a data symbol. The debugger stops the target when the memory at the specified address is read or written.

This command records the ID of the watchpoint in a new debugger variable, $*n*, where *n* is a number. You can use this variable, in a script, to delete or modify the watchpoint behavior. If $*n* is the last or second-to-last debugger variable, then you can also access the ID using $ or $$, respectively.

Note

Watchpoints are only supported on scalar values.

Some targets do not support watchpoints. Currently you can only set a watchpoint on a hardware target using a debug hardware agent.

The address of the instruction that triggers the watchpoint might not be the address shown in the PC register. This is because of pipelining effects.

Syntax

`awatch [-d] [-p] {[filename:]symbol|*address} [vmid vmid]`

Where:

d	Disables the watchpoint immediately after creation.
p	Specifies whether or not the resolution of an unrecognized watchpoint location results in a pending watchpoint being created.
filename	Specifies the file.
symbol	Specifies a global/static data symbol. For arrays or structs you must specify the element or member.
address	Specifies the address. This can be either an address or an expression that evaluates to an address.
vmid	Specifies the *Virtual Machine ID* (VMID) to apply the breakpoint to. This can be either an integer or an expression that evaluates to an integer.

Example

Example 2-9 `awatch`

```
awatch myVar1                 # Set read/write watchpoint on myVar1
awatch *0x80D4                # Set read/write watchpoint on address 0x80D4
```

See also

- *Using expressions* on page 2-4
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clearwatch* on page 2-47
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82

- *rwatch* on page 2-142
- *watch* on page 2-227.

2.3.5 backtrace

`backtrace` is an alias for `info stack`.

See *info stack, backtrace, where* on page 2-104.

2.3.6 break

This command sets an execution breakpoint at a specific location. You can also specify a conditional breakpoint by using an `if` statement that stops only when the conditional expression evaluates to true.

This command records the ID of the breakpoint in a new debugger variable, $*n*, where *n* is a number. You can use this variable, in a script, to delete or modify the breakpoint behavior. If $*n* is the last or second-to-last debugger variable, then you can also access the ID using $ or $$, respectively.

Note

Breakpoints that are set within a shared object or kernel module become pending when the shared object or kernel module is unloaded.

Use `set breakpoint` to control the automatic breakpoint behavior when using this command.

Syntax

```
break [-d] [-p] [[filename:]location|*address] [thread|core number…] [if expression]
```

Where:

d	Disables the breakpoint immediately after creation.
p	Specifies whether or not the resolution of an unrecognized breakpoint location results in a pending breakpoint being created.
filename	Specifies the file.
location	Specifies the location:

line_num	is a line number
function	is a function name.
label	is a label name.
+offset|*-offset*	Specifies the line offset from the current location.

address	Specifies the address. This can be either an address or an expression that evaluates to an address.
number	Specifies one or more threads or processors to apply the breakpoint to. You can use `$thread` to refer to the current thread. If *number* is not specified then all threads are affected.
expression	Specifies an expression that is evaluated when the breakpoint is hit.

If no arguments are specified then a breakpoint is set at the current PC.

You can use `info breakpoints` to display the number and status of all breakpoints and watchpoints.

Example

Example 2-10 `break`

```
break *0x8000                # Set breakpoint at address 0x8000
break *0x8000 thread $thread # Set breakpoint at address 0x8000 on
                             # current thread
break *0x8000 thread 1 3     # Set breakpoint at address 0x8000 on
                             # threads 1 and 3
break main                   # Set breakpoint at address of main()
break SVC_Handler            # Set breakpoint at address of label SVC_Handler
break +1                     # Set breakpoint at address of next source line
break my_File.c:main         # Set breakpoint at address of main() in my_File.c
break my_File.c:10           # Set breakpoint at address of line 10 in my_File.c
break function1 if x>0       # Set conditional breakpoint that stops when x>0
```

See also

- *Using expressions* on page 2-4
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *condition* on page 2-48
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *hbreak* on page 2-74
- *ignore* on page 2-78
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *resolve* on page 2-139
- *set arm* on page 2-146
- *set breakpoint* on page 2-151
- *tbreak* on page 2-216
- *thbreak* on page 2-218.

2.3.7 `break-script`

This command assigns a script file to a specific breakpoint. When the breakpoint is triggered then the script is executed.

Syntax

`break-script` *`number`* `[`*`filename`*`]`

Where:

`number`	Specifies the breakpoint number. This is the number assigned by the debugger when it is set. You can use `info breakpoints` to display the number and status of all breakpoints and watchpoints.
`filename`	Specifies the script file that you want to execute when the specified breakpoint is triggered. If *`filename`* is not specified then the currently assigned *`filename`* is removed from the breakpoint.

Usage

Be aware of the following when using scripts with breakpoints:

- You must not assign a script to a breakpoint that has sub-breakpoints. If you do, the debugger attempts to execute the script for each sub-breakpoint. If this happens, an error message is displayed.
- Take care with the commands you use in a script that is attached to a breakpoint. For example, if you use the `quit` command in a script, the debugger disconnects from the target when the breakpoint is hit.
- If you put the `continue` command at the end of a script, this has the same effect as setting the **Continue Execution** checkbox on the Breakpoint Properties dialog box.

Example

Example 2-11 `break-script`

```
break-script 1 myScript.ds      # Run myScript.ds when breakpoint 1 is triggered
```

See also

- *Using expressions* on page 2-4
- *break* on page 2-38
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *condition* on page 2-48
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *hbreak* on page 2-74
- *ignore* on page 2-78
- *info breakpoints, info watchpoints* on page 2-81

- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *resolve* on page 2-139
- *set arm* on page 2-146
- *set breakpoint* on page 2-151
- *tbreak* on page 2-216
- *thbreak* on page 2-218.

2.3.8 break-stop-on-cores

`break-stop-on-cores` is an alias for `break-stop-on-threads`.

See *break-stop-on-threads, break-stop-on-cores* on page 2-43.

2.3.9 `break-stop-on-threads`, `break-stop-on-cores`

This command applies an existing breakpoint to one or more threads or processors.

Syntax

`break-stop-on-threads` *`number`* `[`*`id`*`]…`

`break-stop-on-cores` *`number`* `[`*`id`*`]…`

Where:

`number`	Specifies the breakpoint number. This is a unique breakpoint number assigned by the debugger when it is set. You can use `info breakpoints` to display the breakpoint numbers and status.
`id`	Specifies one or more threads or processors to apply the breakpoint to. You can use `$thread` or `$core` to refer to the current thread or processor. If *`id`* is not specified then apply the breakpoint to all threads or processors. You can use `info cores`, or `info threads` to display the *`id`* numbers.

Example

Example 2-12 `break-stop-on-threads`**,** `break-stop-on-cores`

```
break-stop-on-threads 1 2           # Apply breakpoint 1 to thread 2
break-stop-on-threads 4 9 11        # Apply breakpoint 4 to threads 9 and 11
break-stop-on-cores 4               # Apply breakpoint 4 to all processors
```

See also

- *Using expressions* on page 2-4
- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *condition* on page 2-48
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *hbreak* on page 2-74
- *ignore* on page 2-78
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *info cores* on page 2-85
- *info threads* on page 2-107
- *resolve* on page 2-139
- *set arm* on page 2-146
- *set breakpoint* on page 2-151
- *tbreak* on page 2-216
- *thbreak* on page 2-218
- *thread, core* on page 2-220.

2.3.10 break-stop-on-vmid

This command applies an existing hardware breakpoint to a *Virtual Machine* (VM).

Syntax

`break-stop-on-vmid` *`number`* `[`*`vmid`*`]`

Where:

`number`	Specifies the hardware breakpoint number. This is a unique breakpoint number assigned by the debugger when it is set. You can use `info breakpoints` to display the breakpoint numbers and status.
`vmid`	Specifies the *Virtual Machine ID* (VMID) to apply the breakpoint to. This can be either an integer or an expression that evaluates to an integer. If *`vmid`* is not specified then the VM effect is removed from the breakpoint.

Example

Example 2-13 `break-stop-on-vmid`

```
break-stop-on-vmid 1 2              # Apply hardware breakpoint 1 to vmid 2
```

See also

- *Using expressions* on page 2-4
- *break* on page 2-38
- *break-script* on page 2-40
- *clear* on page 2-46
- *condition* on page 2-48
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *hbreak* on page 2-74
- *ignore* on page 2-78
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *info cores* on page 2-85
- *info threads* on page 2-107
- *resolve* on page 2-139
- *set arm* on page 2-146
- *set breakpoint* on page 2-151
- *tbreak* on page 2-216
- *thbreak* on page 2-218
- *thread, core* on page 2-220.

2.3.11 `cd`

This command changes the current working directory.

Syntax

`cd` *`dir`*

Where:

`dir`	Specifies the directory.

Example

Example 2-14 `cd`

```
cd "\usr\source"                    # Change the current working directory
```

See also

- *add-symbol-file* on page 2-31
- *file, symbol-file* on page 2-68
- *load* on page 2-114
- *loadfile* on page 2-115
- *pwd* on page 2-135.

2.3.12 `clear`

This command deletes a breakpoint at a specific location.

Syntax

`clear [[`*`filename`*`:]`*`location`*`|`*`*address`*`]`

Where:

filename	Specifies the file.
location	Specifies the location:
	line_num — is a line number.
	function — is a function name.
	label — is a label name.
	+offset|*-offset* — Specifies the line offset from the current location.
address	Specifies the address. This can be either an address or an expression that evaluates to an address.

If no arguments are specified then the breakpoint at the current PC is deleted.

Example

Example 2-15 `clear`

```
clear *0x8000       # Clear breakpoint at address 0x8000
clear main          # Clear breakpoint at address of main()
clear SVC_Handler   # Clear breakpoint at address of label SVC_Handler
clear +1            # Clear breakpoint at address of next source line
clear my_File.c:main # Clear breakpoint at address of main() in my_File.c
clear my_File.c:10   # Clear breakpoint at address of line 10 in my_File.c
```

See also

- *Using expressions* on page 2-4
- *clearwatch* on page 2-47
- *condition* on page 2-48
- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *hbreak* on page 2-74
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *tbreak* on page 2-216
- *thbreak* on page 2-218.

2.3.13 `clearwatch`

This command deletes a watchpoint at a specific location.

Syntax

`clearwatch [`*`filename`*`:]`*`symbol`*`|*`*`address`*

Where:

`filename`	Specifies the file.
`symbol`	Specifies a global/static data symbol. For arrays or structs you must specify the element or member.
`address`	Specifies the address. This can be either an address or an expression that evaluates to an address.

Example

Example 2-16 `clearwatch`

```
clearwatch *0x8000        # Clear watchpoint at address 0x8000
clearwatch my_File.c:myVar # Clear watchpoint at address of myVar in my_File.c
```

See also

- *Using expressions* on page 2-4
- *awatch* on page 2-35
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82.

2.3.14 `condition`

This command sets a break condition for a specific breakpoint or watchpoint. If the value of a specific expression evaluates to true then the debugger stops the target otherwise execution resumes.

Syntax

`condition` *`number`* `[`*`expression`*`]`

Where:

`number`	Specifies the breakpoint or watchpoint number. This is the number assigned by the debugger when it is set. You can use `info breakpoints` to display the number and status of all breakpoints and watchpoints.
`expression`	Specifies an expression that is evaluated when the breakpoint or watchpoint is hit. If no *`expression`* is specified then the breakpoint or watchpoint condition is deleted.

Example

Example 2-17 `condition`

```
condition 1 myVar<5      # Set break condition myVar<5 for breakpoint number 1
```

See also

- *Using expressions* on page 2-4
- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *hbreak* on page 2-74
- *ignore* on page 2-78
- *info breakpoints, info watchpoints* on page 2-81
- *tbreak* on page 2-216
- *thbreak* on page 2-218.

2.3.15 `continue`

This command continues running the target.

Note

Control is returned as soon as the target is running. You can use the `wait` command to block the debugger from returning control until either the application completes or a breakpoint is hit.

Syntax

`continue [`*`count`*`]`

Where:

`count`	Specifies the number of times to ignore the breakpoint or watchpoint at the current location.

Example

Example 2-18 `continue`

```
continue         # Continue running target
continue 5       # Continue running target, ignoring current breakpoint 5 times
```

See also

- *advance* on page 2-33
- *run* on page 2-141
- *start* on page 2-209
- *wait* on page 2-226.

2.3.16 `core`

`core` is an alias for `threads`.

See *thread, core* on page 2-220.

2.3.17 define

This command enables you to a derive new user-defined command from existing commands. User-defined commands accept arguments separated by whitespace. You can use the arguments in expressions by using `$arg0`...`$arg`*`n`*, to refer to specific arguments or `$argv` to refer to all the supplied arguments. For example:

```
print 4+$arg0      # add 4 to the first argument and print result
echo $argv         # echo all arguments
```

Syntax

```
define cmd
   ...
end
```

Where:

cmd	Specifies the command name followed by one or more debugger commands. Enter each debugger command on a new line and terminate the `define` command by using the `end` command.

Note

Existing built in commands cannot be redefined.

Example

Example 2-19 `define`

```
# Define add-args command to print sum of first 3 arguments
define add-args
    print $arg0+$arg1+$arg2
end
```

See also

- *document* on page 2-59
- *end* on page 2-66
- *if* on page 2-77
- *while* on page 2-231
- *Using expressions* on page 2-4.

2.3.18 `delete breakpoints`

This command deletes one or more breakpoints or watchpoints.

Syntax

`delete [breakpoints] number…`

Where:

`number`	Specifies the breakpoint or watchpoint number. This is the number assigned by the debugger when it is set. You can use `info breakpoints` to display the number and status of all breakpoints and watchpoints.

Note

Multiple-statements on a single line of source code are assigned sub-numbers, for example *n.n*. You can specify all multiple-statement breakpoints by specifying *n*.0 or individually by specifying *n.n*.

If no *`number`* is specified then all breakpoints and watchpoints are deleted.

Example

Example 2-20 `delete breakpoints`

```
delete breakpoints 1          # Delete breakpoint number 1
delete breakpoints 1 2        # Delete breakpoints number 1 and 2
delete breakpoints            # Delete all breakpoints and watchpoints
delete breakpoint $           # Delete breakpoint whose number is in the
                              # most recently created debugger variable
```

See also

- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *clearwatch* on page 2-47
- *condition* on page 2-48
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *hbreak* on page 2-74
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *tbreak* on page 2-216
- *thbreak* on page 2-218.

2.3.19 `delete memory`

This command deletes one or more user-defined memory regions.

Syntax

`delete memory` *`number…`*

Where:

`number`	Specifies the region number. This is the number assigned by the debugger when the region is set. You can use `info mem` to display the number and status of all regions.

Example

Example 2-21 `delete memory`

```
delete memory 1              # Delete region number 1
delete memory 1 2            # Delete regions number 1 and 2
delete memory $              # Delete memory region whose number is in
                             # the most recently created debugger variable
```

See also

- *disable memory* on page 2-56
- *enable memory* on page 2-65
- *info memory* on page 2-94
- *memory* on page 2-118.

2.3.20 `directory`

This command specifies additional directories to search for source files. If you use this command without an argument then the search directories are reset to the default settings. You can use the `show` command to display the current settings.

Syntax

`directory [path]…`

Where:

`path`	Specifies an additional directory to search for source files. This is appended to the beginning of the list.

Note

Multiple directories can be specified but must be separated with either:
- a space
- a colon (Unix)
- a semi-colon (Windows).

Default

The default directories for searching are:
- compilation directory, `$cdir`
- current working directory, `$cwd`
- current image directory, `$idir`.

Example

Example 2-22 `directory`

```
directory "\usr\source"      # Add directory to search list
directory "\usr" "\My Src"   # Add two directories to search list,
                             # first takes precedence
directory                    # Reset to the default directories
```

See also
- *set substitute-path* on page 2-171
- *show directories* on page 2-188
- *show substitute-path* on page 2-202.

2.3.21 `disable breakpoints`

This command disables one or more breakpoints or watchpoints.

Syntax

`disable [breakpoints] number…`

Where:

`number`	Specifies the breakpoint or watchpoint number. This is the number assigned by the debugger when it is set. You can use `info breakpoints` to display the number and status of all breakpoints and watchpoints.

——— Note ———

Multiple-statements on a single line of source code are assigned sub-numbers, for example *n.n*. You can specify all multiple-statement breakpoints by specifying *n*.0 or individually by specifying *n.n*.

If no *`number`* is specified then all breakpoints and watchpoints are disabled.

——— Note ———

The breakpoints sub-command is optional.

Example

Example 2-23 `disable`

```
disable breakpoints 1          # Disable breakpoint number 1
disable breakpoints 1 2        # Disable breakpoints number 1 and 2
disable breakpoints            # Disable all breakpoints and watchpoints
disable breakpoints $          # Disable the breakpoint whose number is in
                               # the most recently created debugger variable
```

See also

- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *condition* on page 2-48
- *delete breakpoints* on page 2-52
- *enable breakpoints* on page 2-64
- *hbreak* on page 2-74
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *tbreak* on page 2-216
- *thbreak* on page 2-218.

2.3.22 `disable memory`

This command disables one or more user-defined memory regions.

Syntax

`disable memory` *`number`*…

Where:

`number`	Specifies the region number. This is the number assigned by the debugger when the region is set. You can use `info mem` to display the number and status of all regions.

Example

Example 2-24 `disable memory`

```
disable memory 1              # Disable region number 1
disable memory 1 2            # Disable regions number 1 and 2
disable memory $              # Disable memory region whose number is in
                              # the most recently created debugger variable
```

See also

- *delete memory* on page 2-53
- *enable memory* on page 2-65
- *info memory* on page 2-94
- *memory* on page 2-118.

2.3.23 `disassemble`

This command displays the disassembly for the function surrounding a specific address or the disassembly for a specific address range.

Syntax

`disassemble [`*`address`*` [`*`address`*`|+size]]`

Where:

`address`	Specifies an expression that evaluates to an address. Two *`address`* arguments specify an inclusive address range. If no *`address`* argument is specified then the debugger displays the disassembly for the function surrounding the program counter for the current frame.
`size`	Specifies the size of the region.

Example

Example 2-25 `disassemble`

```
disassemble               # Display disassembly for current function
disassemble 0x8140 0x8157 # Display disassembly for address range 0x8140-0x8157
disassemble 0x8140 +0x18  # Display disassembly for address range 0x8140-0x8157
disassemble 0xC0040AC0    # Display disassembly for address range 0xC0040AC0-0xC0040ADC
```

See also

- *set arm* on page 2-146
- *x* on page 2-232.

2.3.24 `discard-symbol-file`

This command discards debug information relating to a specific file.

Syntax

`discard-symbol-file` *`filename`*

Where:

`filename` Specifies the image, shared library, or *Operating System* (OS) module.

Note

Shared library and OS modules depend on connections that support loading these types of files.

You can use the `info files` command to display information about the loaded files.

Example

Example 2-26 `discard-symbol-file`

```
discard-symbol-file myFile.axf          # Discard symbols relating to myFile.axf
discard-symbol-file myLib.so            # Discard symbols relating to shared library
discard-symbol-file myModule.ko         # Discard symbols relating to OS module
```

See also

- *add-symbol-file* on page 2-31
- *cd* on page 2-45
- *file, symbol-file* on page 2-68
- *load* on page 2-114
- *info files, info target* on page 2-86
- *info os-modules* on page 2-96
- *loadfile* on page 2-115
- *reload-symbol-file* on page 2-137.

2.3.25 `document`

This command enables you to add integrated help for a new user-defined command.

Syntax

```
document cmd
   ...
end
```

Where:

cmd	Specifies the user-defined command name. Enter the description on one of more lines of text and terminate the `document` command by using the `end` command.

Note

Documentation for existing built in commands cannot be redefined.

Example

Example 2-27 `document`

```
# Documentation for the new user-defined add-args command
document add-args
    This user-defined command prints the sum of the first 3 arguments
end
```

See also

- *define* on page 2-51
- *end* on page 2-66
- *if* on page 2-77
- *while* on page 2-231
- *Using expressions* on page 2-4.

2.3.26 down

This command moves the current frame pointer down the call stack towards the bottom frame. It also displays the function name and source line number for the specified frame.

Note

Each frame is assigned a number that increases from the bottom frame (zero) through the call stack to the top frame that is the start of the application.

Syntax

`down [offset]`

Where:

`offset`	Specifies a frame offset from the current frame pointer in the call stack. If no *`offset`* is specified then the default is one.

Example

Example 2-28 down

```
down     # Move and display information 1 frame down from current frame pointer
down 2   # Move and display information 2 frames down from current frame pointer
```

See also

- *down-silently* on page 2-61
- *finish* on page 2-70
- *frame* on page 2-72
- *info frame* on page 2-88
- *info all-registers* on page 2-80
- *info registers* on page 2-99
- *info stack, backtrace, where* on page 2-104
- *select-frame* on page 2-144
- *up* on page 2-224
- *up-silently* on page 2-225.

2.3.27 `down-silently`

This command moves the current frame pointer down the call stack towards the bottom frame.

Note

Each frame is assigned a number that increases from the bottom frame (zero) through the call stack to the top frame that is the start of the application.

Syntax

`down-silently [`*`offset`*`]`

Where:

`offset`	Specifies a frame offset from the current frame pointer in the call stack. If no *`offset`* is specified then the default is one.

Example

Example 2-29 `down-silently`

```
down-silently                  # Move 1 frame down from current frame pointer
down-silently 2                # Move 2 frames down from current frame pointer
```

See also

- *down* on page 2-60
- *finish* on page 2-70
- *frame* on page 2-72
- *info frame* on page 2-88
- *info all-registers* on page 2-80
- *info registers* on page 2-99
- *info stack, backtrace, where* on page 2-104
- *select-frame* on page 2-144
- *up* on page 2-224
- *up-silently* on page 2-225.

2.3.28 dump

This command reads data from memory or the result of an expression and writes it to a file.

Syntax

`dump [`*`format`*`] memory `*`filename start_address`*` {`*`end_address`*`|+size}`

`dump [`*`format`*`] value `*`filename expression`*

Where:

format	Specifies the output format:
	`binary` — Binary. This is the default.
	`elf` — 32-bit ARM ELF.
	`elf64` — 64-bit ARM ELF.
	`ihex` — Intel Hex-32.
	`srec` — Motorola 32-bit (S-records).
	`vhx` — Byte oriented hexadecimal (Verilog Memory Model).
filename	Specifies the file.
start_address	Specifies the start address for the memory.
end_address	Specifies the inclusive end address for the memory.
size	Specifies the size of the region.
expression	Specifies an expression that is evaluated to an address and the data from that address is written to the file.

Example

Example 2-30 `dump`

```
dump memory myFile.bin 0x8000 0x8FFF     # Write content of memory 0x8000-0x8FFF
                                         # to binary file myFile.bin
dump srec value myFile.m32 &myArray      # Write contents of myArray to
                                         # Motorola 32-bit file myFile.m32
```

See also

- *Using expressions* on page 2-4
- *append* on page 2-34
- *restore* on page 2-140.

2.3.29 echo

This command displays textual strings only.

Backslashes can be used as follows:

- C escape sequences, for example, "\n" can be used to print a new line
- Leading and trailing spaces are not displayed unless escaped with a backslash
- Quoted strings are printed literally including the quote marks.

Syntax

`echo` *`string`*

Where:

`string`	Specifies a string of characters.

Example

Example 2-31 `echo`

```
echo "   initializing..."    # Display: "   initializing..." (includes quotes)
echo Stage 1\n               # Display: Stage 1 (followed by a new line)
echo \   Init                # Display:    Init (includes leading spaces)
echo 4+4                     # Display: 4+4
```

See also

- *output* on page 2-131
- *print, inspect* on page 2-134
- *printf() style format string* on page 2-9.

2.3.30 `enable breakpoints`

This command enables one or more breakpoints or watchpoints.

Syntax

`enable [breakpoints] number…`

Where:

number Specifies the breakpoint or watchpoint number. This is the number assigned by the debugger when it is set. You can use `info breakpoints` to display the number and status of all breakpoints and watchpoints.

Note

Multiple-statements on a single line of source code are assigned sub-numbers, for example *n.n*. You can specify all multiple-statement breakpoints by specifying *n*.0 or individually by specifying *n.n*.

If no *number* is specified then all breakpoints and watchpoints are enabled.

Note

The breakpoints sub-command is optional.

Example

Example 2-32 `enable`

```
enable breakpoints 1          # Enable breakpoint number 1
enable breakpoints 1 2        # Enable breakpoints number 1 and 2
enable breakpoints            # Enable all breakpoints and watchpoints
enable breakpoints $          # Enable the breakpoint whose number is in the
                              # most recently created debugger variable
```

See also

- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *condition* on page 2-48
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *hbreak* on page 2-74
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *tbreak* on page 2-216
- *thbreak* on page 2-218.

2.3.31 `enable memory`

This command enables one or more user-defined memory regions.

Syntax

`enable memory number…`

Where:

number	Specifies the region number. This is the number assigned by the debugger when the region is set. You can use `info mem` to display the number and status of all regions.

Example

Example 2-33 `enable memory`

```
enable memory 1              # Enable region number 1
enable memory 1 2            # Enable regions number 1 and 2
enable memory $              # Enable memory region whose number is in
                             # the most recently created debugger variable
```

See also

- *delete memory* on page 2-53
- *disable memory* on page 2-56
- *info memory* on page 2-94
- *memory* on page 2-118.

2.3.32 `end`

This command enables you to terminate conditional blocks when using the `define`, `if`, and `while` commands.

Example

Example 2-34 `end`

```
# Define a while loop containing commands to conditionally execute
# myVar is is a variable in the application code
while myVar<10
    step
    wait
    x
    set myVar++
end
```

See also

- *define* on page 2-51
- *document* on page 2-59
- *if* on page 2-77
- *while* on page 2-231
- *Using expressions* on page 2-4.

2.3.33 `exit`

`exit` is an alias for `quit`.

See *quit, exit* on page 2-136.

2.3.34 `file, symbol-file`

This command loads debug information from an image into the debugger and records the entry point address for future use by the `run` and `start` commands. Subsequent use of the `file` command discards existing information before loading the new debug information. If you want to append debug information instead of replacing it, you can use the `add-symbol-file` command.

Note

The PC register is not set with this command.

Syntax

`file [`*`filename`*`] [`*`offset`*`] [-`*`option`*`]`

`symbol-file [`*`filename`*`] [`*`offset`*`] [-`*`option`*`]`

Where:

`filename` Specifies the image. If no *`filename`* is specified then the current debug information is discarded.

`offset` Specifies the offset that is added to all addresses within the image. If *`offset`* is not specified then the default for:

- An image is zero.
- A shared library is the load address of the library. If the application has not currently loaded the specified library then the request is pended until the library is loaded and the offset can be determined.

`option` Controls how debug information is loaded:

`readnow`	Specifies loading all debug information immediately. This option uses more memory and is slower to load but it enables faster debugging.
`demandload`	Specifies loading debug information when required by the debugger. This option enables a faster load and uses less memory but debugging might be slower. This is the default.

Example

Example 2-35 `file, symbol-file`

```
file "myFile.axf"                   # Load debug information on demand
file "images\myFile.axf"            # Load debug information on demand
file                                # Discard all current debug information
file "myFile.axf" -readnow          # Load all debug information
```

See also

- *add-symbol-file* on page 2-31
- *cd* on page 2-45
- *discard-symbol-file* on page 2-58
- *load* on page 2-114
- *info files, info target* on page 2-86
- *loadfile* on page 2-115

- *reload-symbol-file* on page 2-137
- *run* on page 2-141
- *start* on page 2-209.

2.3.35 `finish`

This command continues running the target to the next instruction after the selected number of stack frames finish.

Syntax

`finish [n]`

Where:

n	Specifies the number of stack frames to finish executing. The default is one.

Example

Example 2-36 `finish`

```
finish               # Continues running until the current stack frame finishes
finish 5             # Continues running until 5 stack frames finish
```

See also

- *down* on page 2-60
- *down-silently* on page 2-61
- *frame* on page 2-72
- *next* on page 2-127
- *nexts* on page 2-129
- *step* on page 2-211
- *steps* on page 2-213
- *select-frame* on page 2-144
- *up* on page 2-224
- *up-silently* on page 2-225.

2.3.36 flash load

This command loads sections from an image into one or more flash devices.

Syntax

`flash load` *`filename`* `[`*`device`*`[:`*`parameter`*`=`*`value`*`]…]…`

Where:

`filename`	Specifies the image.
`device`	Specifies the flash device name. Use this option to restrict the load to the specified device only.
`parameter`	Specifies a parameter or comma separated list of parameters to override.

If no *`device`* is specified then all devices can be loaded. This is dependent on the sections in the image that correspond to the flash device regions.

You can use `info flash` to display information about the flash devices on the current target.

Example

Example 2-37 `flash load`

```
flash load "foo.axf"    # loads the file to flash
flash load "foo.axf" MainFlash:ramAddress=0x20000100,ramSize=0xFF00
                        # Loads the file to a flash device and overrides the parameters
```

See also

- *info flash* on page 2-87
- *load* on page 2-114
- *loadfile* on page 2-115.

2.3.37 frame

This command sets the current frame pointer in the call stack and also displays the function name and source line number for the specified frame.

Note

Each frame is assigned a number that increases from the bottom frame (zero) through the call stack to the top frame that is the start of the application.

Syntax

`frame [number]`

Where:

number	Specifies the frame number. The default is the current frame.

Example

Example 2-38 `frame`

```
frame 1          # Move to and display information for stack frame 1
frame            # Display stack frame information at current frame pointer
```

See also

- *down* on page 2-60
- *down-silently* on page 2-61
- *finish* on page 2-70
- *info frame* on page 2-88
- *info all-registers* on page 2-80
- *info registers* on page 2-99
- *info stack, backtrace, where* on page 2-104
- *select-frame* on page 2-144
- *up* on page 2-224
- *up-silently* on page 2-225.

2.3.38 `handle`

This command controls the handler settings for one or more signals or processor exceptions. The default handler settings are dependant on the type of debug activity. For example, by default on a Linux kernel connection, all signals are handled by Linux on the target. You can use `info signals` to display the current settings.

When connected to an application running on a remote target using gdbserver, the debugger handles Unix signals but on bare-metal it handles processor exceptions.

Syntax

`handle [`*`name`*`]… `*`keyword…`*

Where:

`name`	Specifies the signal or processor exception name.
`keyword`	Specifies the following keywords:

`noprint`	Disables the print property.
`nostop`	Disables the stop property.
`print`	Enables the print property. When using gdbserver the debugger can only print if `stop` is enabled.
`stop`	Enables the stop and print properties.

If no *`name`* is specified then all handler settings are modified.

Example

Example 2-39 `handle`

```
handle SVC stop                  # Enable stop and print for SVC handler
handle IRQ noprint               # Disable print for IRQ handler
handle noprint                   # Disable print for all handlers
```

See also

- *info signals, info handle* on page 2-102.

2.3.39 `hbreak`

This command sets a hardware execution breakpoint at a specific location. You can also specify a conditional breakpoint by using an `if` statement that stops only when the conditional expression evaluates to true.

This command records the ID of the breakpoint in a new debugger variable, $*n*, where *n* is a number. You can use this variable, in a script, to delete or modify the breakpoint behavior. If $*n* is the last or second-to-last debugger variable, then you can also access the ID using $ or $$, respectively.

Note

The number of hardware breakpoints are usually limited. If you run out of hardware breakpoints then delete or disable one that you are no longer using.

Breakpoints that are set within a shared object or kernel module become pending when the shared object or kernel module is unloaded.

Syntax

```
hbreak [-d] [-p] [[filename:]location|*address] [thread|core number...] [vmid vmid] [if expression]
```

Where:

d	Disables the breakpoint immediately after creation.
p	Specifies whether or not the resolution of an unrecognized breakpoint location results in a pending breakpoint being created.
filename	Specifies the file.
location	Specifies the location:

line_num	is a line number.
function	is a function name.
label	is a label name.
+offset|*-offset*	Specifies the line offset from the current location.

address	Specifies the address. This can be either an address or an expression that evaluates to an address.
number	Specifies one or more threads or processors to apply the breakpoint to. You can use `$thread` to refer to the current thread. If *number* is not specified then all threads are affected.
vmid	Specifies the *Virtual Machine ID* (VMID) to apply the breakpoint to. This can be either an integer or an expression that evaluates to an integer.
expression	Specifies an expression that is evaluated when the breakpoint is hit.

If no arguments are specified then a hardware breakpoint is set at the current PC.

Example

Example 2-40 hbreak

```
hbreak *0x8000               # Set breakpoint at address 0x8000
hbreak *0x8000 thread $thread # Set breakpoint at address 0x8000 on current thread
hbreak *0x8000 thread 1 3     # Set breakpoint at address 0x8000 on threads 1 and 3
hbreak main                   # Set breakpoint at address of main()
hbreak SVC_Handler            # Set breakpoint at address of label SVC_Handler
hbreak +1                     # Set breakpoint at address of next source line
hbreak my_File.c:main         # Set breakpoint at address of main() in my_File.c
hbreak my_File.c:8            # Set breakpoint at address of line 8 in my_File.c
hbreak function1 if x>0       # Set conditional breakpoint that stops when x>0
```

See also

- *Using expressions* on page 2-4
- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *condition* on page 2-48
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *ignore* on page 2-78
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *resolve* on page 2-139
- *tbreak* on page 2-216
- *thbreak* on page 2-218.

2.3.40 help

This command displays help information for a specific command or a group of commands listed according to specific debugging tasks.

Syntax

`help [command|group]`

Where:

command	Specifies an individual command.
group	Specifies a group name for specific debugging tasks:

`group_all`	Displays all the commands by group.
`group_breakpoints`	Displays the breakpoint and watchpoint commands.
`group_data`	Displays the commands that displays source data.
`group_display`	Displays the output and print settings commands.
`group_files`	Displays the commands that interact with files.
`group_info`	Displays the program information commands.
`group_log`	Displays the message logging commands.
`group_flash`	Displays the flash commands.
`group_memory`	Displays the commands that interact with memory.
`group_os`	Displays the operating system commands.
`group_registers`	Displays the register commands.
`group_running`	Displays the target execution and stepping group.
`group_show`	Displays the show commands for debugger settings.
`group_set`	Displays the set commands for debugger settings.
`group_scripts`	Displays the commands for use in script files.
`group_stack`	Displays the call stack commands.
`group_support`	Displays the supporting commands.

Example

Example 2-41 `help`

```
help load               # Display help information for load command
help print              # Display help information for print command
help group_breakpoints  # Display group of breakpoint and watchpoint commands
help group_files        # Display group of file commands
```

2.3.41 `if`

This command enables you to write scripts that conditionally execute debugger commands.

Syntax

```
if condition
   ...
else
   ...
end
```

Where:

`condition`	Specifies a conditional expression. Follow the `if` statement with one or more debugger commands that execute when the expression evaluates to true. **Note** The `else` statement is optional and the debugger commands that follow it only execute when *`condition`* evaluates to false.

Enter each debugger command on a new line and terminate the `if` command by using the `end` command.

Example

Example 2-42 `if`

```
# Define an if statement containing commands to conditionally execute
if $pc==0x80000
    break
    info stack full
end
```

See also

- *define* on page 2-51
- *document* on page 2-59
- *end* on page 2-66
- *while* on page 2-231
- *Using expressions* on page 2-4.

2.3.42 ignore

This command sets the ignore counter for a breakpoint or watchpoint condition.

Syntax

`ignore number count`

Where:

number	Specifies the breakpoint or watchpoint number. This is the number assigned by the debugger when it is set.
count	Specifies the number of times to ignore the specified breakpoint or watchpoint. The ignore counter is incremented only when the condition evaluates to true.

You can use `info breakpoints` to display the number and status of all breakpoints and watchpoints.

Example

Example 2-43 `ignore`

```
ignore 2 3      # Ignore breakpoint 2 for 3 hits
ignore $ 3      # Ignore breakpoint, whose number is in the
                # most recently created debugger variable, for 3 hits
```

See also

- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *condition* on page 2-48
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *hbreak* on page 2-74
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *tbreak* on page 2-216.
- *thbreak* on page 2-218.

2.3.43 `info address`

This command displays the location of a symbol.

Syntax

`info address` *`symbol`*

Where:

`symbol`	Specifies the symbol.

Example

Example 2-44 `info address`

```
info address mySymbol                # Display location of symbol
```

2.3.44 `info all-registers`

This command displays the name and content of registers for the current stack frame.

Unless you specify otherwise, the registers listed by this command are the full set made available by the target, including co-processor and floating-point registers where available. You can use the `info registers` command to display a subset of registers that are most useful when debugging C/C++ applications.

When application code calls a function it is common for any existing register values to be saved, so that the registers can be used by the calling function for other purposes. The original register values are then restored when the function returns. When displaying register values the debugger tries to show the value of the actual registers prior to each function call, according to the currently selected stack frame. A consequence of this is that some registers might be shown with undefined values because the debugger is unable to determine the actual value.

Syntax

`info all-registers [group]`

Where:

group	Specifies a group name for a specific registers. If no *group* is specified then all registers and groups are displayed.

Example

Example 2-45 `info all-registers`

```
info all-registers                  # Display info for all registers
info all-registers USR              # Display info for all user mode registers
```

See also

- *down* on page 2-60
- *down-silently* on page 2-61
- *frame* on page 2-72
- *info registers* on page 2-99
- *select-frame* on page 2-144
- *up* on page 2-224
- *up-silently* on page 2-225.

2.3.45 `info breakpoints, info watchpoints`

This command displays information about the status of all breakpoints and watchpoints.

Note

This command sets a default address variable to the location of the last breakpoint or watchpoint listed. Some commands, such as `x`, use this default value if no address is specified.

Syntax

`info breakpoints`

`info watchpoints`

Example

Example 2-46 `info breakpoints, info watchpoints`

```
info breakpoints            # Display status for all breakpoints and watchpoints
```

See also

- *awatch* on page 2-35
- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *clearwatch* on page 2-47
- *condition* on page 2-48
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *hbreak* on page 2-74
- *ignore* on page 2-78
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *rwatch* on page 2-142
- *tbreak* on page 2-216
- *thbreak* on page 2-218
- *watch* on page 2-227
- *x* on page 2-232.

2.3.46 `info breakpoints capabilities, info watchpoints capabilities`

This command displays a list of parameters that you can use with breakpoint and watchpoint commands for the current connection.

Syntax

`info breakpoints capabilities`

`info watchpoints capabilities`

Example

Example 2-47 `info breakpoints capabilities, info watchpoints capabilities`

```
info breakpoints capabilities      # Display list of parameters for current connection
```

See also

- *awatch* on page 2-35
- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *clearwatch* on page 2-47
- *condition* on page 2-48
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *hbreak* on page 2-74
- *ignore* on page 2-78
- *info breakpoints, info watchpoints* on page 2-81
- *rwatch* on page 2-142
- *tbreak* on page 2-216
- *thbreak* on page 2-218
- *watch* on page 2-227
- *x* on page 2-232.

2.3.47 `info capabilities`

This command displays a list of capabilities for the target device that is currently connected to the debugger. For more information, see the documentation for your target.

Syntax

`info capabilities`

Example

Example 2-48 `info capabilities`

```
info capabilities                 # Display target device capabilities
```

See also

- *reset* on page 2-138.

2.3.48 info classes

This command displays C++ class names.

Syntax

`info classes [expression]`

Where:

expression Specifies a class name or a wildcard expression. You can use wildcard expressions to enhance your pattern matching.

If no *expression* is specified then all classes are displayed.

Example

Example 2-49 info classes

```
info classes                   # Display info for all classes
info classes m*                # Display info for names starting with m
                               # (use when set wildcard-style=glob)
info classes my_class[0-9]+    # Display info for names with my_class followed
                               # by a number (use when set wildcard-style=regex)
```

See also

- *Using wildcards* on page 2-5
- *set wildcard-style* on page 2-175.

2.3.49 `info cores`

This command displays a list of processors. It shows the number (a unique number assigned by the debugger), name, current state, and related stack frame including the function names and source line number.

Syntax

`info cores`

Example

Example 2-50 `info cores`

```
info cores                    # Display all processors
```

See also

- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *thread, core* on page 2-220.

2.3.50 `info files, info target`

This command displays information about the loaded image and symbols.

Syntax

`info files`

`info target`

Example

Example 2-51 `info files, info target`

```
info files          # Display information for loaded image and symbols
```

See also

- *add-symbol-file* on page 2-31
- *discard-symbol-file* on page 2-58
- *file, symbol-file* on page 2-68
- *load* on page 2-114
- *loadfile* on page 2-115
- *reload-symbol-file* on page 2-137.

2.3.51 info flash

This command displays information about the flash devices on the current target.

Syntax

```
info flash
```

Example

Example 2-52 `info flash`

```
info flash          # Display information about the current flash devices.
```

See also

- *flash load* on page 2-71.

2.3.52 info frame

This command gives the following information about the selected frame:

- stack frame address
- current PC address
- saved PC address
- calling frame address
- source language
- frame arguments and associated addresses
- address of the local variables
- stack pointer address for the previous frame
- saved registers and associated location.

Note

Each frame is assigned a number that increases from the bottom frame (zero) through the call stack to the top frame that is the start of the application.

Syntax

`info frame [number]`

Where:

number	Specifies the frame number.

If no arguments are specified then the stack frame information for the current frame pointer is displayed.

Example

Example 2-53 `info frame`

```
info frame 1          # Display information for stack frame 1
info frame            # Display information for stack frame at current location
```

See also

- *down* on page 2-60
- *down-silently* on page 2-61
- *frame* on page 2-72
- *info stack, backtrace, where* on page 2-104
- *select-frame* on page 2-144
- *up* on page 2-224
- *up-silently* on page 2-225.

2.3.53 `info functions`

This command displays the name and data types for all functions.

Syntax

`info functions [`*`expression`*`]`

Where:

`expression`	Specifies a function name or a wildcard expression. You can use wildcard expressions to enhance your pattern matching If no *`expression`* is specified then all functions are displayed.

Example

Example 2-54 `info functions`

```
info functions                 # Display info for all functions
info functions m*              # Display info for names starting with m
                               # (use when set wildcard-style=glob)
info functions my_func[0-9]+   # Display info for names with my_func followed
                               # by a number (use when set wildcard-style=regex)
```

See also

- *Using wildcards* on page 2-5
- *set wildcard-style* on page 2-175.

2.3.54 `info handle`

`info handle` is an alias for `info signals`.

See *info signals, info handle* on page 2-102.

2.3.55 info inst-sets

This command displays the available instruction sets.

Syntax

```
info inst-sets
```

Example

Example 2-55 `info inst-sets`

```
info inst-sets           # Display available instruction sets
```

See also

- *set arm* on page 2-146
- *show arm* on page 2-180.

2.3.56 `info locals`

This command displays all local variables that are accessible in the function corresponding to the current stack frame.

Syntax

`info locals`

Example

Example 2-56 `info locals`

```
info locals          # Display all local variables for the current stack frame
```

2.3.57 `info members`

This command displays the name and data types for all class member variables that are accessible in the function corresponding to the selected stack frame.

Syntax

`info members [expression]`

Where:

`expression`	Specifies the name of a class member or a C expression that evaluates to a struct, union or class variable. If no *`expression`* is specified then all members of the current function identified by **`this`** pointer are displayed.

——— **Note** ———

Using high compiler optimization levels such as `-O2` with --debug can produce a less than satisfactory debug view because the mapping of object code to source code is not always clear. If the compiler optimizes away the **`this`** pointer then using the `info members` command without an expression produces an error.

———————

Example

Example 2-57 `info members`

```
info members                    # Display members for the current function
info members my_Struct[0-9]+    # Display members for matching struct variables
```

See also

- *Using expressions* on page 2-4.

2.3.58 info memory

This command displays the attributes for all memory regions.

Syntax

`info memory`

Example

Example 2-58 `info memory`

```
info memory                # Display attributes for all memory regions
```

See also

- *delete memory* on page 2-53
- *disable memory* on page 2-56
- *enable memory* on page 2-65
- *memory* on page 2-118
- *memory debug-cache* on page 2-121.

2.3.59 `info os-log`

This command displays the contents of the *Operating System* (OS) log buffer for connections that support this feature. On Linux this is the contents of the kernel `dmesg` log.

Note

A Linux kernel connection must be established and the target is stopped before you can use this command.

Syntax

`info os-log`

Example

Example 2-59 `info os-log`

```
info os-log              # Displays the OS log buffer
```

See also

- *info os-modules* on page 2-96
- *info os-version* on page 2-97
- *info processes* on page 2-98
- *set os* on page 2-161
- *show os* on page 2-194.

2.3.60 info os-modules

This command displays a list of loadable kernel modules for connections that support this feature.

Note

A connection must be established and operating system support must be enabled within the debugger before a loadable module can be detected. You can use the `set os` command to control operating system support in the debugger.

Syntax

`info os-modules [-s]`

Where:

`s`	Displays the section information of the modules.

Example

Example 2-60 `info os-modules`

```
info os-modules              # Displays info for loaded OS modules
```

See also

- *info os-log* on page 2-95
- *info os-version* on page 2-97
- *info processes* on page 2-98
- *set os* on page 2-161
- *show os* on page 2-194.

2.3.61 `info os-version`

This command displays the version of the *Operating System* (OS) for connections that support this feature.

Syntax

```
info os-version
```

Example

Example 2-61 `info os-version`

```
info os-version              # Displays the version of the OS
```

See also

- *info os-log* on page 2-95
- *info os-modules* on page 2-96
- *info processes* on page 2-98
- *set os* on page 2-161
- *show os* on page 2-194.

2.3.62 `info processes`

This command displays a list of all user space processes. It shows the number (a unique number assigned by the debugger), OS ID (pid), OS Parent ID, kind, OS state, current state, and related stack frame including the function names and source line number.

Syntax

`info processes`

Example

Example 2-62 `info processes`

```
info processes                      # Display all user space processes
```

See also

- *info os-log* on page 2-95
- *info os-modules* on page 2-96
- *info os-version* on page 2-97
- *info threads* on page 2-107
- *set os* on page 2-161
- *show os* on page 2-194
- *thread, core* on page 2-220.

2.3.63 `info registers`

This command displays the name and content of registers for the current stack frame. The registers listed by this command are a subset that are most useful when debugging C/C++ applications. You can use the `info all-registers` command to list the full set of registers.

When application code calls a function it is common for any existing register values to be saved, so that the registers can be used by the calling function for other purposes. The original register values are then restored when the function returns. When displaying register values the debugger tries to show the value of the actual registers prior to each function call, according to the currently selected stack frame. A consequence of this is that some registers might be shown with undefined values because the debugger is unable to determine the actual value.

Syntax

`info registers [register]`

Where:

`register`	Specifies the register name. If no *`register`* is specified then all application level registers are displayed.

Example

Example 2-63 `info registers`

```
info registers              # Display info for all application level registers
info registers pc           # Display info for PC register
```

See also

- *down* on page 2-60
- *down-silently* on page 2-61
- *frame* on page 2-72
- *info all-registers* on page 2-80
- *select-frame* on page 2-144
- *up* on page 2-224
- *up-silently* on page 2-225.

2.3.64 `info semihosting`

This command displays semihosting information.

Syntax

`info semihosting [server|clients|all]`

Where:

`all`	Displays information on the semihosting server listener port, a list of the connected clients, and the heap and stack. This is the default.
`server`	Displays information on the semihosting server listener port.
`clients`	Displays information on each of the semihosting streams `stdin`, `stdout`, `stderr`. This includes a list of the connected clients.
`heap`	Displays the heap information that the debugger used to initialize the heap. **Note**: This information is only displayed if the debugger performs the initialization.
`stack`	Displays the stack information that the debugger used to initialize the stack. **Note**: This information is only displayed if the debugger performs the initialization.

Example

Example 2-64 `info semihosting`

```
info semihosting           # Displays all semihosting information
info semihosting clients   # Display clients info for semihosting streams
```

2.3.65 `info sharedlibrary`

This command displays the names of the loaded shared libraries, the base address, and whether the debug symbols of the shared libraries are loaded or not.

Note

You must launch the debugger with `--target_os` command-line option before you can use this feature. In Eclipse this option is automatically selected when you connect to a target using `gdbserver`.

Syntax

`info sharedlibrary [/order] [/sort_by] [/group]`

Where:

order	Specifies the sorting order:	
	a	Ascending order. This is the default.
	d	Descending order.
sort_by	Specifies the sorting order of the shared objects:	
	b	Sort by base addresses. This is the default.
	n	Sort by library names.
group	Specifies whether to group the debug symbols:	
	s	Group loaded symbols followed by unloaded symbols.
	sn	Group unloaded symbols followed by loaded symbols.

Example

Example 2-65 `info sharedlibrary`

```
info sharedlibrary          # Display shared libraries by base address, asc
info sharedlibrary /n       # Display shared libraries by library name, asc
info sharedlibrary /d       # Display shared libraries by base address, desc
info sharedlibrary /n /a /s # Display shared libraries grouped loaded->unloaded
                            # and by library name, asc
```

See also

- *nosharedlibrary* on page 2-130
- *sharedlibrary* on page 2-176.

2.3.66 `info signals, info handle`

This command displays information about the handling of signals or processor exceptions.

When connected to an application running on a remote target using gdbserver, the debugger handles Unix signals but on bare-metal it handles processor exceptions.

Syntax

`info signals [`*`name`*`]`

`info handle [`*`name`*`]`

Where:

`name`	Specifies the signal name. If no *`name`* is specified then all handler settings are displayed.

Example

Example 2-66 `info signals, info handle`

```
info signals                    # Display info for all signals
info signals IRQ                # Display info for IRQ signal
```

See also

- *handle* on page 2-73.

2.3.67 info sources

This command displays the names of the source files used in the current image being debugged. Where possible the names are resolved to the location on the host system.

Syntax

`info sources`

Example

Example 2-67 `info sources`

```
info sources                       # Display the names of source files
```

See also

- *add-symbol-file* on page 2-31
- *file, symbol-file* on page 2-68
- *load* on page 2-114
- *loadfile* on page 2-115.

2.3.68 info stack, backtrace, where

This command displays a numbered list of the calling stack frames including the function names and source line numbers. You can use `set backtrace` to control the default call stack display settings.

——— **Note** ———

Each frame is assigned a number that increases from the bottom frame (zero) through the call stack to the top frame that is the start of the application.

————————

Syntax

`info stack [n|-n] [full]`

`backtrace [n|-n] [full]`

`where [n|-n] [full]`

Where:

n	Specifies `n` frames from the bottom of the call stack.
-n	Specifies `n` frames from the top of the call stack.
`full`	Specifies the additional display of local variables.

Example

Example 2-68 `info stack, backtrace, where`

```
info stack                # Display call stack
backtrace -5              # Display top 5 frames of the call stack
backtrace full            # Display call stack including local variables
where                     # Display call stack
```

See also

- *down* on page 2-60
- *down-silently* on page 2-61
- *frame* on page 2-72
- *info frame* on page 2-88
- *select-frame* on page 2-144
- *set backtrace* on page 2-149
- *show backtrace* on page 2-182
- *thread, core* on page 2-220
- *up* on page 2-224
- *up-silently* on page 2-225.

2.3.69 `info symbol`

This command displays the symbol name at a specific address.

Syntax

`info symbol` *`address`*

Where:

`address`	Specifies the address.

Example

Example 2-69 `info symbol`

```
info symbol 0x8000                # Display symbol name at address 0x8000
```

2.3.70 `info target`

`info target` is an alias for `info files`.

See *info files, info target* on page 2-86.

2.3.71 `info threads`

This command displays a list of all threads. It shows the number (a unique number assigned by the debugger), OS ID (pid), OS Parent ID, kind, OS state, current state, and related stack frame including the function names and source line number.

Note

When kernel debugging this command displays kernel threads only. For user space processes you can use the `info processes` command.

Syntax

`info threads`

Example

Example 2-70 `info threads`

```
info threads                 # Display all threads
```

See also

- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *info processes* on page 2-98
- *thread, core* on page 2-220.

2.3.72 `info variables`

This command displays the name and data types of global and static variables.

Syntax

`info variables [expression]`

Where:

expression	Specifies a symbol name or a wildcard expression. You can use wildcard expressions to enhance your pattern matching. If no *expression* is specified then all global and static variables are displayed.

Example

Example 2-71 `info variables`

```
info variables               # Display info for all variables
info variables num           # Display info for num variable
info variables m*            # Display info for names starting with m
                             # (use when set wildcard-style=glob)
info variables my_var[0-9]+  # Display info for names with my_var followed
                             # by a number (use when set wildcard-style=regex)
```

See also

- *Using wildcards* on page 2-5
- *set wildcard-style* on page 2-175
- *set variable* on page 2-174.

2.3.73 `info watchpoints`

`info watchpoints` is an alias for `info breakpoints`.

See *info breakpoints, info watchpoints* on page 2-81.

2.3.74 `inspect`

`inspect` is an alias for `print`.

See *print, inspect* on page 2-134.

2.3.75 `interrupt, stop`

This command interrupts the target and stops the current application if it is running.

Syntax

```
interrupt
```

```
stop
```

Example

Example 2-72 `interrupt`

```
interrupt                   # interrupt current application
```

See also

- *continue* on page 2-49
- *run* on page 2-141
- *start* on page 2-209.

2.3.76 `list`

This command displays lines of source code surrounding the current or specified location. The default listing is 10 lines of source code unless you specify start and finish line numbers. You can use the `set listsize` command to modify the default settings.

Repeated commands display successive source lines in the same direction through the source file.

Syntax

`list [[filename:]location|+|-|+offset|-offset]|[*address]`

Where:

`filename`	Specifies the file.
`location`	Specifies the location:
	`line_num` is a line number
	`first,last` are start and finish line numbers
	`function` is a function.
`+`	Displays the source lines after the current location.
`-`	Displays the source lines before the current location.
`offset`	Specifies the line offset from the current location.
`address`	Specifies the address. This can be either an address or an expression that evaluates to an address.

Default

The default directories for searching are:

- compilation directory, `$cdir`
- current working directory, `$cwd`
- current image directory, `$idir`.

You can use the `directory` command to define additional search directories.

Example

Example 2-73 `list`

```
list main          # Set current location to main() and display source
list +3            # Increment current location then display source
list -             # Decrement current location then display source
list *0x8120       # Set current location to address 0x8120 and display source
list 35            # Set current location to line 35 and display source
list dhry_1.c:10,23 # Display source lines 10 to 23 in dhry_1.c
list *main         # Set current location to address of main and display source
```

See also

- *Using expressions* on page 2-4
- *directory* on page 2-54

- *set listsize* on page 2-160
- *show listsize* on page 2-193.

2.3.77 load

This command loads an image on to the target and records the entry point address for future use by the `run` and `start` commands.

Note

The PC register is not set with this command.

Debug information is not loaded with this command. You can use either the `add-symbol-file`, `file`, or `loadfile` command to load debug information.

Syntax

`load [`*`filename`*`] [`*`offset`*`]`

Where:

`filename`	Specifies the image. If no *`filename`* is specified then the executable image specified by the previous command is loaded. You can use `info files` to display information about the current image and symbols.
`offset`	Specifies the offset that is added to all addresses within the image.

Example

Example 2-74 `load`

```
load "myFile.axf"                         # Load image
load "images\myFile.axf"                  # Load image
load myFile.axf 0x2000                    # Load image with offset 0x2000
```

See also

- *add-symbol-file* on page 2-31
- *cd* on page 2-45
- *discard-symbol-file* on page 2-58
- *file, symbol-file* on page 2-68
- *flash load* on page 2-71
- *info files, info target* on page 2-86
- *loadfile* on page 2-115
- *run* on page 2-141
- *start* on page 2-209.

2.3.78 `loadfile`

This command loads debug information into the debugger, an image on to the target and records the entry point address for future use by the `run` and `start` commands. Subsequent use of the `loadfile` command discards existing information before loading the new debug information.

Note

The PC register is not set with this command.

Syntax

`loadfile [filename] [offset] [-option]`

Where:

filename	Specifies the image. If no *filename* is specified then the executable image specified by a previous command is loaded. You can use `info files` to display information about the current image and symbols.
offset	Specifies the offset that is added to all addresses within the image.
option	Controls how debug information is loaded:

`readnow`	Specifies loading all debug information immediately. This option uses more memory and is slower to load but it enables faster debugging.
`demandload`	Specifies loading debug information when required by the debugger. This option enables a faster load and uses less memory but debugging might be slower. This is the default.

Example

Example 2-75 `loadfile`

```
loadfile "myFile.axf"         # Load image and debug information when required
loadfile "images\myFile.axf"  # Load image and debug information when required
loadfile myFile.axf 0x2000    # Load image with offset 0x2000 and load debug
                              # information when required
loadfile "myFile.axf" -readnow # Load image and all debug information
```

See also

- *add-symbol-file* on page 2-31
- *cd* on page 2-45
- *discard-symbol-file* on page 2-58
- *file, symbol-file* on page 2-68
- *flash load* on page 2-71
- *info files, info target* on page 2-86
- *load* on page 2-114
- *reload-symbol-file* on page 2-137
- *run* on page 2-141
- *start* on page 2-209.

2.3.79 `log config`

This command specifies the type of logging configuration to output runtime messages from the debugger.

Syntax

`log config` *`option`*

Where:

`option`	Specifies a predefined logging configuration or a user-defined logging configuration file:
`info`	Output messages using the predefined INFO level configuration. This is the default.
`debug`	Output messages using the predefined DEBUG level configuration.
`filename`	Specifies a user-defined logging configuration file to customize the output of messages. The debugger supports log4j configuration files.

You can use this command with the `log file` command to output messages to a file in addition to the console.

Example

Example 2-76 `log config`

```
log config debug                  # Display all debug messages
```

See also

- *log file* on page 2-117
- *Log4j* in *Apache Logging Services*, `http://logging.apache.org`

2.3.80 `log file`

This command outputs messages to a file in addition to the console.

Syntax

`log file [`*`filename`*`]`

Where:

`filename`	Specifies the output file. If no *`filename`* is specified then output messages are sent only to the console.

Example

Example 2-77 `log file`

```
log file myOutput.log       # Output debugger messages to myOutput.log and console
```

See also

- *cd* on page 2-45
- *log config* on page 2-116.

2.3.81 `memory`

This command defines a memory region. It records the ID of the memory region in a new debugger variable, $*n*, where *n* is a number. You can use this variable, in a script, to delete or modify the status of the memory region. If $*n* is the last or second-to-last debugger variable, then you can also access the ID using $ or $$, respectively.

Syntax

`memory` *`start_address`* `{`*`end_address`*`|+size}` `[`*`attributes`*`]…`

Where:

start_address	Specifies the start address for the region.
end_address	Specifies the inclusive end address for the region. You can use `0x0` as a shortcut to represent the end of the address space.
size	Specifies the size of the region.
attributes	Specifies additional attributes:

access_mode	Specifies the access mode for the region:	
	`na`	no access
	`ro`	read-only
	`wo`	write-only
	`rw`	read/write. This is the default.
width	Specifies the access width:	
	`8`	8-bit
	`16`	16-bit
	`32`	32-bit
	`64`	64-bit.
	It is only necessary to specify a specific access width where the memory region is sensitive to this, for example, when accessing some peripherals. If no *width* is specified then the debugger uses any available access width and generally provides the highest performance.	
`bp|nobp`	Controls whether or not software breakpoints can be set in the region. `bp` is the default.	
`hbp|nohbp`	Controls whether or not hardware breakpoints can be set in the region. `hbp` is the default.	
`cache|nocache`	Controls whether the debugger can cache data read from the memory region. Enabling the caching of memory can improve debugger performance. Memory regions that can be modified by external sources should not be cached by the debugger. For example volatile peripherals. `nocache` is the default.	
`verify|noverify`	Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written. The `verify` option also requires the `rw` attribute to be specified so	

that the verify operation to be performed. ARM recommends that you mark areas of memory containing peripherals as `noverify`, because some peripheral registers are volatile such that reading their value changes their contents as a side-effect.

`verify` is the default.

Example

Example 2-78 `memory`

```
memory 0x1000 0x2FFF cache     # specify RW region 0x1000-0x2FFF (cache)
memory 0x3000 0x7FFF ro 8      # specify 8-bit RO region 0x3000-0x7FFF (nocache)
memory 0x8000 0x0              # specify RW region 0x8000-0xFFFF (nocache)
```

See also

- *delete memory* on page 2-53
- *disable memory* on page 2-56
- *enable memory* on page 2-65
- *info memory* on page 2-94
- *memory auto* on page 2-120
- *memory debug-cache* on page 2-121.

2.3.82 `memory auto`

This command resets the memory regions to the default target settings and discards all user-defined regions.

Syntax

```
memory auto
```

Example

Example 2-79 `memory auto`

```
memory auto                         # reset default memory regions
```

See also

- *delete memory* on page 2-53
- *disable memory* on page 2-56
- *enable memory* on page 2-65
- *info memory* on page 2-94
- *memory* on page 2-118.

2.3.83 `memory debug-cache`

This command globally controls the caching of memory regions by the debugger. You can use `info mem` to display the caching attributes.

Syntax

`memory debug-cache` *`option`*

Where:

`option` Specifies additional options:

`off`	Globally disables debugger caching of memory regions. All memory accesses are performed directly on the target.
`on`	Globally enables debugger caching of memory regions. When caching is globally enabled the debugger might cache the results of read operations from memory regions that allow caching. This is the default.
`invalidate`	Invalidates all the caches, so that the next subsequent read from memory is performed on the target and not the cache.

Example

Example 2-80 `memory debug-cache`

```
memory debug-cache off                      # Disable caching
memory debug-cache invalidate               # Invalidates all caches
```

See also

- *info memory* on page 2-94
- *memory* on page 2-118.

2.3.84 memory fill

This command writes a specific pattern of bytes to memory.

Syntax

memory fill *start_address* {*end_address*|+offset} *fill_size* *pattern*

Where:

start_address	Specifies the start address for the region. This can be either an address or an expression that evaluates to an address.
end_address	Specifies the inclusive end address for the region. This can be either an address or an expression that evaluates to an address.
offset	Specifies the length of the region in bytes.
fill_size	Specifies the size of the fill pattern in bytes.
pattern	Specifies an expression that defines the fill pattern. If the pattern does not fit exactly into the specified region, then the remaining bytes are filled with partial bytes from the pattern.

Example

Example 2-81 memory fill

```
memory fill 0x0 0xFFFFFFFF 4 0x12345678   # Fill 0x0 to 0xFFFFFFFF inclusive with int
                                          # value 0x12345678 using default access width
memory fill main (main+15) 1 (char)0x0    # Fill 16 bytes from symbol main with byte
                                          # value 0x0
```

See also

- *info memory* on page 2-94
- *memory set* on page 2-123
- *memory set_typed* on page 2-125.

2.3.85 `memory set`

This command writes to memory.

Syntax

`memory set address width expression`

Where:

address	Specifies an address at which to write the first value. The address must be correctly aligned for the type of the specified expression.
width	Specifies the access width (bits) to use when writing to memory. If the width is narrower than the value being written then more than one access is used to write the value. For example:

`0`	enables the debugger to determine the access width
`8`	8-bit
`16`	16-bit
`32`	32-bit
`64`	64-bit.

Widths are dependent on the target, address region and address alignment. Some access sizes might not be supported.

expression	Specifies either a single expression or an aggregate of expressions with the same size enclosed in curly braces. If there is more than one expression, then the values are written to memory sequentially with the addresses determined by the width of the type of the values.

Note

This command sets a default address variable to the value of the memory address. Some commands, such as x, use this default value if no address is specified.

Example

Example 2-82 `memory set`

```
memory set 0x8000 0 "Hello"  # Writes a string to memory
memory set 0x1000 0 {(char)0x10,(char)0xFF,(char)1,(char)2,(char)3,(char)42}
                             # Is equivalent to the following commands:
                                         # set variable *(char*)0x1000 = (char)0x10
                                         # set variable *(char*)0x1001 = (char)0xFF
                                         # set variable *(char*)0x1002 = (char)1
                                         # set variable *(char*)0x1003 = (char)2
                                         # set variable *(char*)0x1004 = (char)3
memory set 0x1008 0 0x1234   # Equivalent to set variable *(int*)0x1008 = 0x1234
memory set 0x1008 8 0x1234   # Same effect but forces use of 4 writes of one byte each
```

See also

- *info memory* on page 2-94
- *memory fill* on page 2-122
- *memory set_typed* on page 2-125

- *x* on page 2-232.

2.3.86 `memory set_typed`

This command writes a list of values to memory.

Syntax

`memory set_typed` *`address type expressions`*

Where:

`address`	Specifies an address at which to write the first value. The address must be correctly aligned for the specified *`type`*.
`type`	Specifies the data type to which each of the series of expressions is converted and the width of each value in memory. For example, long.
`expressions`	Specifies a space separated list of expressions. If an expression contains spaces it must be enclosed in parentheses. The expressions are evaluated, converted to the specified type, and then written to memory sequentially.

Note

This command sets a default address variable to the value of the memory address. Some commands, such as `x`, use this default value if no address is specified.

Example

Example 2-83 `memory set_typed`

```
memory set_typed 0x8000 (long long) 0x100 0x200
                              # Is equivalent to the following commands:
                              # set variable *((long long*)0x8000) = (long long)0x100
                              # set variable *((long long*)0x8008) = (long long)0x200
```

See also

- *info memory* on page 2-94
- *memory fill* on page 2-122
- *memory set* on page 2-123
- *x* on page 2-232.

2.3.87 `newvar`

This command declares and initializes a new debugger convenience variable. Convenience variables have a dynamic type, which means that they take the value and type of anything assigned to them. They can be used in debugger scripts to store information for later use.

Syntax

`newvar [global] $`*`name`* `[=`*`initial_value`*`]`

Where:

`global`	Specifies that the variable has `global` scope. If `global` is not specified, then the variable is only accessible within its enclosing lexical scope.
`name`	Specifies the name of the new variable. The name must be a valid C identifier but prefixed with $.
`intial_value`	Specifies the initial value of the variable. If an initial value is not specified, then by default, the variable is of integer type with value 0.

Note

- Debugger scripts and the top-level interactive interpreter are considered separate lexical scopes where non-global convenience variables are not visible to any child or parent debugger script.
- A user-defined command created with `define` is considered a separate lexical scope and cannot reference non-global convenience variables in surrounding scripts or from the top-level interpreter.
- The `if`, `else`, and `while` commands define new lexical scopes that inherit parent lexical scopes up to the level of a script, top-level interpreter, or user-defined command.
- Any non-global convenience variables, declared within a lexical scope, are destroyed at the end of the lexical scope.

Example

Example 2-84 `newvar`

```
define advance_hw     # This defines a new command that runs
                      # to an address using a hardware breakpoint.
  hbreak $arg0        # Set a hardware breakpoint at the value of the first parameter.
  newvar $bp_num = $  # Save the number of the breakpoint in a new variable.
  continue
  wait
  delete $bp_num      # Delete the hardware breakpoint.
end
advance_hw 0x00008000
```

See also

- *Memory* on page 2-19
- *break* on page 2-38
- *watch* on page 2-227.

2.3.88 `next`

This command steps through an application at the source level stopping at the first instruction of each source line but stepping over all function calls. You must compile your code with debug information to use this command successfully.

Syntax

`next [count]`

Where:

count	Specifies the number of source lines to execute.

Note

Execution stops immediately if a breakpoint is reached, even if fewer than *count* source lines are executed.

Example

Example 2-85 `next`

```
next                                  # Execute one source line
next 5                                # Execute five source lines
```

See also

- *finish* on page 2-70
- *nexti* on page 2-128
- *nexts* on page 2-129
- *step* on page 2-211
- *stepi* on page 2-212
- *steps* on page 2-213.

2.3.89 nexti

This command steps through an application at the instruction level but stepping over all function calls.

Syntax

`nexti [count]`

Where:

count	Specifies the number of instructions to execute.

Note

Execution stops immediately if a breakpoint is reached, even if fewer than *count* instructions are executed.

Example

Example 2-86 `nexti`

```
nexti                                   # Execute one instruction
nexti 5                                 # Execute five instructions
```

See also

- *next* on page 2-127
- *nexts* on page 2-129
- *step* on page 2-211
- *stepi* on page 2-212
- *steps* on page 2-213.

2.3.90 `nexts`

This command steps through an application at the source level stopping at the first instruction of each source statement but stepping over all function calls. You must compile your code with debug information to use this command successfully.

Syntax

`nexts [count]`

Where:

count	Specifies the number of source statements to execute.

Note

Execution stops immediately if a breakpoint is reached, even if fewer than *count* source statements are executed.

Example

Example 2-87 `nexts`

```
nexts                               # Execute one source statement
nexts 5                             # Execute five source statements
```

See also

- *finish* on page 2-70
- *next* on page 2-127
- *nexti* on page 2-128
- *step* on page 2-211
- *stepi* on page 2-212
- *steps* on page 2-213.

2.3.91 `nosharedlibrary`

This command discards all loaded shared library symbols.

Note

You must launch the debugger with `--target_os` command-line option before you can use this feature. In Eclipse this option is automatically selected when you connect to a target using `gdbserver`.

Syntax

```
nosharedlibrary
```

Example

Example 2-88 `nosharedlibrary`

```
nosharedlibrary                 # Discards loaded shared library symbols
```

See also

- *info sharedlibrary* on page 2-101
- *sharedlibrary* on page 2-176.

2.3.92 `output`

This command displays only the result of an expression. This is similar to the `print` command but it does not record the results in a debugger variable.

Syntax

`output [/`*`flag`*`]` *`expression`*

Where:

`flag`	Specifies the output format:
	`x` Hexadecimal (casts the value to an unsigned integer prior to printing in hexadecimal)
	`d` Signed decimal. This is the default.
	`u` Unsigned decimal
	`o` Octal
	`t` Binary
	`a` Absolute hexadecimal address
	`c` Character
	`f` Floating-point
	`s` Default format from the expression.
`expression`	Specifies an expression that is evaluated and the result is returned.

Note

If your expression accesses memory then a default address variable is set to the location after the last accessed address. Some commands, such as `x`, use this default value if no address is specified.

Example

Example 2-89 `output`

```
output (int*)8                # Cast a number as a pointer
output 4+4                    # Display result of expression in decimal
output "initializing..."      # Display progress information
output $PC /x                 # Display address in PC register (hexadecimal)
```

See also

- *Using expressions* on page 2-4
- *echo* on page 2-63
- *print, inspect* on page 2-134
- *x* on page 2-232
- *printf() style format string* on page 2-9.

2.3.93 `pause`

This command pauses the execution of a script for a specified period of time.

Syntax

`pause` *`number`*`[ ms | s ]`

Where:

`number`	Specifies the period of time.
`ms`	Specifies the time in milliseconds. This is the default.
`s`	Specifies the time in seconds.

Example

Example 2-90 `pause`

```
pause 1000                                    # Pause for 1 second
pause 0.5s                                    # Pause for half a second
```

2.3.94 `preprocess`

This command displays the preprocessed expression, not the evaluated expression.

Syntax

```
preprocess [expression]
```

Note

This functionality is dependent on the compiler generating accurate macro debug information.

Example

Example 2-91 `preprocess`

If your application contained the following code:

```
#define BASE_ADDRESS (0x1000)
#define REG_ADDRESS  (BASE_ADDRESS + 0x10)

int main () {
    return REG_ADDRESS;
}
```

During a debug session, you can display the `REG_ADDRESS` by using:

```
>preprocess REG_ADDRESS
((0x1000) + 0x10)
```

This compares with the expression value as output by the print command:

```
>print/x REG_ADDRESS
0x1010
```

See also

- *print, inspect* on page 2-134

2.3.95 `print, inspect`

This command displays the output of an expression (128 character limit) and also records the result in a new debugger variable, $*n*, where *n* is a number. Results from the `print` command can be used successively in expressions using the $ character. If you do not want the results recorded in a debugger variable, use the `output` command instead.

Syntax

`print [/`*`flag`*`] [`*`expression`*`]`

`inspect [/`*`flag`*`] [`*`expression`*`]`

Where:

`flag` Specifies the output format:

`x`	Hexadecimal (casts the value to an unsigned integer prior to printing in hexadecimal)
`d`	Signed decimal. This is the default.
`u`	Unsigned decimal
`o`	Octal
`t`	Binary
`a`	Absolute hexadecimal address
`c`	Character
`f`	Floating-point
`s`	Default format from the expression.

`expression` Specifies an expression that is evaluated and the result is returned. If no *`expression`* is specified then the last expression is repeated.

Note

If your expression accesses memory then a default address variable is set to the location after the last accessed address. Some commands, such as `x`, use this default value if no address is specified.

Example

Example 2-92 `print, inspect`

```
print (int*)8                  # Cast a number as a pointer
print 4+4                      # Display result of expression in decimal
print "initializing..."        # Display progress information
print /x $PC                   # Display address in PC register (hexadecimal)
```

See also

- *Using expressions* on page 2-4
- *echo* on page 2-63
- *output* on page 2-131
- *x* on page 2-232
- *printf() style format string* on page 2-9.

2.3.96 pwd

This command displays the current working directory.

Syntax

```
pwd
```

Example

Example 2-93 pwd

```
pwd                                 # Display current working directory
```

See also

- *cd* on page 2-45.

2.3.97 `quit, exit`

This command quits the debugger session.

Syntax

`quit`

`exit`

Example

Example 2-94 `quit, exit`

```
quit                               # Quit debugger session
```

2.3.98 `reload-symbol-file`

This command reloads debug information from an already loaded image into the debugger using the same settings as the original load operation. For example, you can use this command to reload debug information into the debugger after you have rebuilt your image.

——— **Note** ———

The PC register is not set with this command.

———————————

Syntax

`reload-symbol-file [`*`filename`*`]`

Where:

`filename`	Specifies the image to reload. If is not already loaded then an error is generated.

Example

Example 2-95 `reload-symbol-file`

```
reload-symbol-file "myFile.axf"                    # Reload debug information
```

See also

- *add-symbol-file* on page 2-31
- *cd* on page 2-45
- *discard-symbol-file* on page 2-58
- *file, symbol-file* on page 2-68
- *info files, info target* on page 2-86
- *load* on page 2-114
- *loadfile* on page 2-115
- *run* on page 2-141
- *start* on page 2-209.

2.3.99 `reset`

This command performs a reset on the target. The exact behavior of the `reset` command is dependent on the debug agent and the target.

For example:

- a debug agent can be configured to reset the target in different ways
- the position of the switches on the target.
- a `gdbserver` connection can be configured to restart gdbserver and run scripts.

For more information, see the documentation for your target or debug agent.

Note

Reset does not affect the symbols loaded in the debugger. Registers and memory might contain different values after a reset.

Syntax

`reset [`*`key`*`]`

Where:

key	Specifies the reset key. The reset capabilities are target dependent and might not all be enabled. You can use `info capabilities` to display a list of capability settings for the target device that is currently connected to the debugger. Possible options for the reset key are: **app** Application restart. **system** General hardware reset that is not specific to a bus or processor.

If no *key* is specified then the first enabled reset capability is performed.

Example

Example 2-96 `reset`

```
reset                           # Performs the first enabled reset capability
reset app                       # Performs an application restart
reset system                    # Performs a general hardware reset
```

See also

- *info capabilities* on page 2-83.

2.3.100 `resolve`

This command re-evaluates the specified breakpoints or watchpoints and those with addresses that can be resolve are set. Unresolved addresses remain pending.

Syntax

`resolve [number]…`

Where:

`number`	Specifies the breakpoint or watchpoint number. This is the number assigned by the debugger when it is set. You can use `info breakpoints` to display the number and status of all breakpoints and watchpoints.

If no *`number`* is specified then all breakpoints and watchpoints are re-evaluated.

Example

Example 2-97 `resolve`

```
resolve 1                    # Resolve breakpoint/watchpoint number 1
resolve 1 2                  # Resolve breakpoints/watchpoint number 1 and 2
resolve                      # Resolve all breakpoints/watchpoints
resolve $                    # Resolve the breakpoint/watchpoint whose number is in
                             # the most recently created debugger variable
```

See also

- *break* on page 2-38
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *condition* on page 2-48
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *hbreak* on page 2-74
- *ignore* on page 2-78
- *info breakpoints, info watchpoints* on page 2-81
- *tbreak* on page 2-216
- *thbreak* on page 2-218.

2.3.101 `restore`

This command reads data from a file and writes it to memory.

Syntax

`restore filename [binary] [offset [start_address [end_address|+size]]]`

Where:

filename	Specifies the file.
`binary`	Specifies binary format. The file format is only required for binary files. All other files are automatically recognized by the debugger. See the `append` command for a list of the file formats supported by the debugger.
offset	Specifies an offset that is added to all addresses in the image prior to writing to memory. Some image formats do not contain embedded addresses and in this case the offset is the absolute address where the image is restored.
start_address	Specifies the minimum address that can be written to. Any data prior to this address is not written. If no `start_address` is given then the default is address zero.
end_address	Specifies the maximum address that can be written to. Any data after this address is not written. If no `end_address` is given then the default is the end of the address space.
size	Specifies the size of the region.

Example

Example 2-98 `restore`

```
restore myFile.bin binary 0x200       # Restore content of binary file
                                      # myFile.bin starting at 0x200
restore myFile.m32 0x100 0x8000 0x8FFF # Add 0x100 to addresses in Motorola
                                      # 32-bit (S-records) file and restore
                                      # content between 0x8000-0x8FFF
```

See also

- *append* on page 2-34
- *dump* on page 2-62.

2.3.102 `run`

The operation of this command depends on what the target is:

Bare-metal This command sets the PC register to the entry point address previously recorded by the `load`, `loadfile`, or `file` command and starts running the target. Subsequent `run` commands also reload the executable image if it follows a previous load operation.

Linux application

This command sends a request to the server to restart the application and then start running it.

——— Note ———

Control is returned as soon as the target is running. You can use the `wait` command to block the debugger from returning control until either the application completes or a breakpoint is hit.

———————————

Syntax

`run [args]`

Where:

args	Specifies the command-line arguments that are passed to the `main()` function in the application using the `argv` parameter. The name of the image is always implicitly passed in `argv[0]` and it is not necessary to pass this as an argument to the `run` command.

Example

Example 2-99 `run`

```
run                                   # Start running the device
```

See also

- *continue* on page 2-49
- *file, symbol-file* on page 2-68
- *load* on page 2-114
- *loadfile* on page 2-115
- *set semihosting* on page 2-165
- *show semihosting* on page 2-196
- *start* on page 2-209
- *wait* on page 2-226.

2.3.103 `rwatch`

This command sets a watchpoint for a data symbol. The debugger stops the target when the memory at the specified address is read.

This command records the ID of the watchpoint in a new debugger variable, $*n*, where *n* is a number. You can use this variable, in a script, to delete or modify the watchpoint behavior. If $*n* is the last or second-to-last debugger variable, then you can also access the ID using $ or $$, respectively.

Note

Watchpoints are only supported on scalar values.

Some targets do not support watchpoints. Currently you can only set a watchpoint on a hardware target using a debug hardware agent.

The address of the instruction that triggers the watchpoint might not be the address shown in the PC register. This is because of pipelining effects.

Syntax

`rwatch [-d] [-p] {[`*`filename`*`:]`*`symbol`*`|*`*`address`*`} [vmid `*`vmid`*`]`

Where:

d	Disables the watchpoint immediately after creation.
p	Specifies whether or not the resolution of an unrecognized watchpoint location results in a pending watchpoint being created.
filename	Specifies the file.
symbol	Specifies a global/static data symbol. For arrays or structs you must specify the element or member.
address	Specifies the address. This can be either an address or an expression that evaluates to an address.
vmid	Specifies the *Virtual Machine ID* (VMID) to apply the breakpoint to. This can be either an integer or an expression that evaluates to an integer.

Example

Example 2-100 `rwatch`

```
rwatch myVar1                    # Set read watchpoint on myVar1
rwatch *0x80D4                   # Set read watchpoint on address 0x80D4
```

See also

- *Using expressions* on page 2-4
- *awatch* on page 2-35
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *clearwatch* on page 2-47
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82

- *watch* on page 2-227.

2.3.104 `select-frame`

This command moves the current frame pointer in the call stack.

Note

Each frame is assigned a number that increases from the bottom frame (zero) through the call stack to the top frame that is the start of the application.

Syntax

`select-frame` *`number`*

Where:

`number`	Specifies the frame number.

Example

Example 2-101 `select-frame`

```
select-frame 1                  # Move to stack frame 1
```

See also

- *down* on page 2-60
- *down-silently* on page 2-61
- *finish* on page 2-70
- *frame* on page 2-72
- *info frame* on page 2-88
- *info all-registers* on page 2-80
- *info registers* on page 2-99
- *info stack, backtrace, where* on page 2-104
- *up* on page 2-224
- *up-silently* on page 2-225.

2.3.105 `set`

`set` is an alias for `set variable`.

See *set variable* on page 2-174.

2.3.106 `set arm`

This command controls the behavior of the debugger when selecting the instruction set for disassembly and setting breakpoints.

Note

Available instruction sets depend on the target that the debugger is connected to.

Syntax

`set arm` *`option`*

Where:

`option` Specifies additional options:

`force-mode`	Controls the default debugger behavior overriding the `fallback-mode` setting.	
	`a32|arm`	Forces the debugger to use the A32 instruction set.
	`a64`	Forces the debugger to use the A64 instruction set.
	`t32|thumb`	Forces the debugger to use the T32 instruction set.
	`auto`	Forces the debugger to use debug information when available or the `fallback-mode` if this is not available. This is the default.
`fallback-mode`	Controls the default debugger behavior when `force-mode` is set to auto and debug information is not available.	
	`a32|arm`	Forces the debugger to use the A32 instruction set when debug information is not available.
	`a64`	Forces the debugger to use the A64 instruction set when debug information is not available.
	`t32|thumb`	Forces the debugger to use the T32 instruction set when debug information is not available.
	`auto`	Forces the debugger to use the current instruction set of the target. This is the default.

Example

Example 2-102 `set arm`

```
set arm force-mode thumb             # Force the use of Thumb
set arm fallback-mode arm            # When force-mode is auto, use ARM
                                     # if no debug information is available
```

See also

- *break* on page 2-38
- *disassemble* on page 2-57
- *info inst-sets* on page 2-91
- *show arm* on page 2-180
- *start* on page 2-209
- *tbreak* on page 2-216
- *x* on page 2-232.

2.3.107 set auto-solib-add

This command controls the automatic loading of shared library symbols.

Note

You must launch the debugger with `--target_os` command-line option before you can use this feature. In Eclipse this option is automatically selected when you connect to a target using `gdbserver`.

Syntax

`set auto-solib-add {off|on}`

Where:

`off`	No automatic loading. When automatic loading is off you must explicitly load shared library symbols using the `sharedlibrary` command.
`on`	Loads shared library symbols automatically. This is the default.

Example

Example 2-103 `set auto-solib-add`

```
set auto-solib-add off         # No automatic loading of shared library symbols
```

See also

- *show auto-solib-add* on page 2-181.

2.3.108 `set backtrace`

This command controls the default behavior when using the `info stack` command.

Syntax

`set backtrace` *`option`*

Where:

`option`	Specifies additional options:
	`limit` *`n`* — Specifies the maximum limit when displaying the call stack. You can specify zero as the maximum limit to display the entire call stack. The default call stack limit is 100.

Example

Example 2-104 `set backtrace`

```
set backtrace limit 10          # Limit the call stack display to 10 frames
set backtrace limit 0           # No limit, display the entire call stack
```

See also

- *info stack, backtrace, where* on page 2-104
- *show backtrace* on page 2-182.

2.3.109 `set blocking-run-control`

This command controls whether run control operations such as stepping and running are blocked until the target stops or released immediately.

Syntax

```
set blocking-run-control {off|on|script-only}
```

Where:

`off`	Specifies asynchronous, control is returned before the target stops.
`on`	Specifies synchronous, run control operations are blocked until the target stops. This has the same effect as issuing a wait command after each run control operation.
`script-only`	Specifies that run control operations block only when executed as commands from within a script. This is the default.

Example

Example 2-105 `set blocking-run-control`

```
set blocking-run-control on      # Block run control operations until target stops
```

See also

- *show blocking-run-control* on page 2-183.

2.3.110 `set breakpoint`

This command controls the automatic behavior of breakpoints and watchpoints.

Syntax

`set breakpoint [`*`option`*`]`

Where:

`option` Specifies additional options:

`auto-hw` Controls the automatic breakpoint selection when using the `break` command:

`off`	Disables automatic breakpoint selection.
`on`	Uses the memory map attributes to decide if hardware or software breakpoints must be used. This is the default.

`auto-remove`

Controls the automatic removal of breakpoints and watchpoints when disconnecting from the target:

`off`	Disables automatic removal.
`on`	Enables automatic removal. This is the default.

Note

If the target is running, the debugger temporarily stops the target before removing breakpoints and watchpoints.

`skipmode` Controls whether to skip all breakpoints and watchpoints:

`off`	Disables skip mode. This is the default.
`on`	Enables skip mode.

Example

Example 2-106 `set breakpoint`

```
set breakpoint auto-hw off        # No automatic breakpoint selection
set breakpoint skipmode on        # Skip all breakpoints and watchpoints
set breakpoint auto-remove off    # No automatic removal of breakpoints and watchpoints
```

See also

- *break* on page 2-38
- *show breakpoint* on page 2-184.

2.3.111 `set case-insensitive-source-matching`

This command controls the case sensitivity of debugger file matching operations.

Syntax

`set case-insensitive-source-matching [off|on]`

Where:

`off`	Specifies case sensitive file matching. This is the default.
`on`	Specifies case insensitive file matching. This is useful if the file paths or filenames in the debug data have a different case to those in the filesystem.

Example

Example 2-107 `set case-insensitive-source-matching`

```
# By default the debugger performs case sensitive file matching.
# Assume that the debug data contains the filename main.c.
break -p "C:/example/Main.c":2 # This fails because Main.c does not match main.c.
WARNING(CMD452-COR167):
! Breakpoint 8 has been pended
! No compilation unit matching "C:/example/Main.c" was found.

set case-insensitive-source-matching on         # case insensitive matching.
break -p "C:/EXAmple/Main.c" # This file matching operation succeeds.
Breakpoint 9 at S:0x000080A8
    on file main.c, line 2
```

See also

- *show case-insensitive-source-matching* on page 2-185
- *set escapes-in-filenames* on page 2-159
- *set wildcard-style* on page 2-175.

2.3.112 `set debug-agent`

This command sets a parameter in the launch configuration for DSTREAM/RVI connections.

Syntax

`set debug-agent` *`name value`*

Where:

`name`	Specifies a name of the parameter to set.
`value`	Specifies the value of the parameter. Values are dependent on the parameter being set. An error is reported if the value is not valid.

Example

Example 2-108 `set debug-agent`

```
set debug-agent options.cortexA9.coreTrace.cycleAccurate False
                      # Set debug agent configuration cycleAccurate parameter to false
```

See also

- *show dtsl-options* on page 2-189.

2.3.113 set debug-from

This command specifies the address of the temporary breakpoint for subsequent use by the `start` command. If you do not specify this command then the default value used by the `start` command is the address of the global function `main()`.

Syntax

`set debug-from` *`expression`*

Where:

`expression` Specifies an expression that evaluates to an address. The expression is only evaluated when the `start` command is processed, therefore, you can refer to symbols that might not exist yet but might be made available in the future. You can use the debugger variable `$entrypoint` to refer to the entry point for the currently loaded image.

Example

Example 2-109 `set debug-from`

```
set debug-from *0x8000          # Set start-at setting to address 0x8000
set debug-from *$entrypoint     # Set start-at setting to address of $entrypoint
set debug-from main+8           # Set start-at setting to address of main+8
set debug-from function1        # Set start-at setting to address of function1
```

See also

- *Using expressions* on page 2-4
- *show debug-from* on page 2-187
- *start* on page 2-209.

2.3.114 `set directories`

`set directories` is an alias for `directory`.

See *directory* on page 2-54.

2.3.115 `set dtsl-options`

This command sets a parameter in the connection DTSL configuration.

Syntax

`set dtsl-options` *`name value`*

Where:

`name`	Specifies a name of the parameter to set.
`value`	Specifies the value of the parameter. Values are dependent on the parameter being set. An error is reported if the value is not valid.

Example

Example 2-110 `set dtsl-options`

```
set dtsl-options options.cortexA9.coreTrace.cycleAccurate False
                            # Set DTSL configuration cycleAccurate parameter to false
```

See also

- *show dtsl-options* on page 2-189.

2.3.116 `set endian`

This command specifies the byte order for use by the debugger. The endianness of the target is not modified by this command.

Syntax

```
set endian {auto|be8|big|little}
```

Where:

`auto`	Uses the same byte order as the image where possible, otherwise it uses the current endianness of the target. This is the default.
`be8`	Specifies Byte Invariant Addressing big-endian mode introduced in architecture ARMv6 (data is big endian and code is little endian).
`big`	Specifies big endian mode.
`little`	Specifies little endian mode.

Example

Example 2-111 `set endian`

```
set endian little                  # Debug using little endian
```

See also

- *show endian* on page 2-190.

2.3.117 `set escape-strings`

This command controls how special characters in strings are printed on the debugger command-line.

Syntax

```
set escape-strings off|on
```

Where:

`off`	Specifies that any backslash characters in strings are treated as escape sequences. For example, if the string contains "`\t`" then this is printed as a tab character. This is the default.
`on`	Specifies that any backslashes in strings are not treated as escape sequences and are instead output literally. For example, if the string contains "`\t`" then this is printed as a "`\`" character followed by a "`t`" character.

Example

Example 2-112 `set escape-strings`

```
set escape-strings on

output "Say \"hello\""

“Say \“hello\””

set escape-strings off

output "Say \"hello\""

"Say ”hello"”
```

See also

- *show escape-strings* on page 2-191.

2.3.118 `set escapes-in-filenames`

This command controls the use of special characters in paths.

Syntax

```
set escapes-in-filenames off|on
```

Where:

`off` Specifies that a backslash in a path is treated as a directory separator (with the exception that it can be used to escape spaces). For example:

```
C:\test\ file.c
```

The first backslash is treated as a separator followed by a t, not an escape sequence representing the tab character. The second backslash escapes the space.

This is the default.

`on` Specifies that a backslash is to be treated as part of an escape sequence to indicate that the character following is a special character. For example:

```
C:\\test\\file.c
```

The backslash in this example is a directory separator and must be identified as a special character.

Example

Example 2-113 `set escapes-in-filenames`

```
set escapes-in-filenames on      # Use backslash as an escape character in paths
```

See also

- *show escapes-in-filenames* on page 2-192.

2.3.119 `set listsize`

This command modifies the default number of source lines that the `list` command displays.

Syntax

`set listsize` *`n`*

Where:

`n`	Specifies the number of source lines.

Example

Example 2-114 `set listsize`

```
set listsize 20                        # Set listing size for list command
```

See also

- *list* on page 2-112
- *show listsize* on page 2-193.

2.3.120 `set os`

This command controls *Operating System* (OS) settings in the debugger.

Note

An OS aware connection must be established before you can use this command.

Syntax

`set os` *`option`*

Where:

`option` Specifies additional options:

`log-capture`	`off`	Disables OS log capture and printing of Linux kernel dmesg logs to console. This is the default.
	`on`	Enables OS log capture and printing to console.
		Note This option automatically checks the connection state and, if required, stops the target before changing this setting.
`enabled`	`auto`	Automatically stops the target and enables OS support when an OS image is loaded into the debugger. For example, Linux kernel images are detected by reading the members for the structure returned by the expression `init_nsproxy.uts_ns->name`. Unloading the image disables OS support. This is the default for Linux kernel connections.
	`deferred`	Automatically enables OS support when an OS image is loaded into the debugger but only when the target next stops. Unloading the image disables OS support. This is the default for *Real-Time Operating System* (RTOS) aware connections.
	`off`	Disables OS support.
	`on`	Enables OS support. Use this option when the OS image is already loaded into the debugger and the target is stopped.

Example

Example 2-115 `set os`

```
set os log-capture on                # Enable OS log capture and printing to console
set os enabled off                   # Disable OS support in debugger
```

See also

- *info os-log* on page 2-95
- *info os-modules* on page 2-96

- *info os-version* on page 2-97
- *info processes* on page 2-98
- *show os* on page 2-194.

2.3.121 `set print`

This command controls the current debugger print settings.

Syntax

`set print option`

Where:

option Specifies additional options:

`library-not-found-warnings`
: Controls the printing of "unable to find library..." messages.
  - `off` Disables these messages. This is the default.
  - `on` Enables these messages.

`full-source-path`
: Controls the printing of source file names in messages.
  - `off` Disables printing the full path. This is the default.
  - `on` Enables printing the full path.

`stop-info`
: Controls the printing of event messages when the target stops.
  - `off` Disables printing of event messages. This setting takes precedence over the `silence` and `unsilence` commands.
  - `on` Enables printing of event messages. This is the default.

`current-vmid`
: Controls the printing of current VMID messages when the target stops.
  - `off` Disables printing of VMID messages. This is the default.
  - `on` Enables printing of VMID messages.

`double-format` *`format`*
: Controls the formatting of double precision floating-point values. *`format`* is a `printf()` style format string. The default is "`%,.16g`".

`float-format` *`format`*
: Controls the formatting of single precision floating-point values. *`format`* is a `printf()` style format string. The default is "`%,.6g`".

Example

Example 2-116 `set print`

```
set print library-not-found-warnings off  # Disable unfound library messages
set print full-source-path on             # Display full source path in messages
set print double-format %+g               # Print decimal scientific notation with sign
set print float-format %08.4e             # Print decimal scientific notation, zero-pad
                                          # min 8 characters, 4 digit precision
```

See also

- *show print* on page 2-195
- *silence* on page 2-207
- *unsilence* on page 2-223
- *printf() style format string* on page 2-9.

2.3.122 `set semihosting`

This command controls the semihosting settings in the debugger. Semihosting is used to communicate input/output requests from application code to the host workstation running the debugger.

Note

These settings only apply if the target supports semihosting and they cannot be changed while the target is running.

Syntax

`set semihosting` *`option`*

Where:

`option` Specifies additional options:

`args` *`arguments`* Specifies the command-line arguments that are passed to the `main()` function in the application using the `argv` parameter. The name of the image is always implicitly passed in `argv[0]` and it is not necessary to pass this as an argument.

`file-base` *`directory`*

Specifies the base directory where the files that the application opens are relative to.

`stderr "stderr"|`*`filename`*

Specifies either console streams or a file to write `stderr` for semihosting operations.

`stdin "stdin"|`*`filename`*

Specifies either console streams or a file to read `stdin` for semihosting operations.

`stdout "stdout"|`*`filename`*

Specifies either console streams or a file to write `stdout` for semihosting operations.

`top-of-memory` *`address`*

Specifies the top of memory.

`stack_heap_options` Specifies finer controls to manually configure the base address and limits for the stack and heap. If you use *`stack_heap_options`*, then these settings take precedence over the top-of-memory and all of the following options must be specified:

`stack-base` *`address`*

The base address of the stack.

`stack-limit` *`address`*

The end address of the stack.

`heap-base` *`address`*

The base address of the heap.

`heap-limit` *`address`*

The end address of the heap.

`enabled`	`auto`	Automatically enables semihosting operations if appropriate when an image is loaded. This is the default.
	`off`	Disables all semihosting operations.
	`on`	Enables all semihosting operations. **Note** You must configure semihosting addresses before you enable semihosting. For example: `set semihosting top-of-memory` *`address`* `set semihosting enabled on`
`vector` *`address`*	Specifies a breakpoint address for semihosting support. If it is not set, the debugger uses vector catch (if supported) or 0x8.	

Example

Example 2-117 `set semihosting`

```
set semihosting args 500              # Set 500 as command-line argument
set semihosting stdout output.log     # Write stdout to output.log
set semihosting enabled on            # Enable semihosting operations
```

See also

- *show semihosting* on page 2-196
- *unset* on page 2-222.

2.3.123 `set solib-absolute-prefix`

`set solib-absolute-prefix` is an alias for `set sysroot`.

See *set sysroot, set solib-absolute-prefix* on page 2-172.

2.3.124 set solib-search-path

This command specifies additional directories to search for shared library symbols. If you use this command without an argument then any additional search directories, previously added using this command, are removed. You can use `show solib-search-path` to display the current settings.

Note

You must launch the debugger with `--target_os` command-line option before you can use this feature. In Eclipse this option is automatically selected when you connect to a target using `gdbserver`.

Syntax

`set solib-search-path [`*`path`*`]…`

Where:

path	Specifies an additional directory to search for shared libraries. The debugger uses the system root directory first, then it searches the additional directories specified with this command. You can use `set sysroot` to specify the system root directory. **Note** Multiple directories can be specified but must be separated with either: • a colon (Unix) • a semi-colon (Windows).

Example

Example 2-118 `set solib-search-path`

```
set solib-search-path "\usr\lib"        # Specify search directory
set solib-search-path "/lib":"/My Lib"   # Specify two search directories(Unix)
```

See also

- *set sysroot, set solib-absolute-prefix* on page 2-172
- *show solib-search-path* on page 2-199
- *show sysroot, show solib-absolute-prefix* on page 2-203.

2.3.125 `set step-mode`

This command controls the default behavior of the `step` and `steps` commands.

Syntax

`set step-mode {step-over|stop|step-until-source}`

Where:

`step-over`	If the instruction is a function call then the debugger performs a step-over. Otherwise, it stops. This is the default.
`stop`	The debugger stops when execution reaches an address with no source.
`step-until-source`	The debugger performs steps until it reaches source. To speed up the execution, the debugger might use abstract interpretation and break or run until the line of source is reached.

Example

Example 2-119 `set step-mode`

```
set step-mode step-over             # Step over a function call and stop.
                                    # Otherwise stop
```

See also

- *show step-mode* on page 2-200
- *step* on page 2-211
- *steps* on page 2-213.

2.3.126 `set stop-on-solib-events`

This command controls whether the debugger stops execution when a shared object is loaded or unloaded.

——— **Note** ———

You must launch the debugger with `--target_os` command-line option before you can use this feature. In Eclipse this option is automatically selected when you connect to a target using `gdbserver`.

Syntax

```
set stop-on-solib-events {off|on}
```

Where:

`off`	Ignore event. This is the default.
`on`	Stop execution. Use this option only when you want the debugger to stop execution. For example, you might want to set a breakpoint in a shared library prior to use or perhaps you might want to check the initialization of global variables.

Example

Example 2-120 `set stop-on-solib-events`

```
set stop-on-solib-events on            # Stop execution when event occurs
```

See also

- *show stop-on-solib-events* on page 2-201.

2.3.127 `set substitute-path`

This command modifies the search paths used by the debugger when it executes any of the commands that look up and display source code. This command is useful when the source files have moved from the original location used during compilation.

Subsequent use of the `set substitute-path` command appends rules to the current list.

Syntax

`set substitute-path` *`path1 path2`*

Where:

`path1`	Specifies the existing search path.
`path2`	Specifies the replacement search path.

Example

Example 2-121 `set substitute-path`

```
set substitute-path "\src" "\My Src"        # Substitute "\src" with "\My Src"
```

See also

- *directory* on page 2-54
- *show substitute-path* on page 2-202
- *unset* on page 2-222.

2.3.128 `set sysroot, set solib-absolute-prefix`

This command specifies the system root directory to search for shared library symbols.

The debugger uses this directory to search for a copy of the debug versions of target shared libraries. The system root on the host workstation must contain an exact representation of the libraries on the target root filesystem.

Note

You must launch the debugger with `--target_os` command-line option before you can use this feature. In Eclipse this option is automatically selected when you connect to a target using `gdbserver`.

Syntax

`set sysroot` *`path`*

`set solib-absolute-prefix` *`path`*

Where:

`path`	Specifies the system root directory.

Example

Example 2-122 `set sysroot, set solib-absolute-prefix`

```
set sysroot "\mySystem"            # Set system root directory "\mySystem"
```

See also

- *set solib-search-path* on page 2-168
- *show solib-search-path* on page 2-199
- *show sysroot, show solib-absolute-prefix* on page 2-203.

2.3.129 `set trust-ro-sections-for-opcodes`

This command controls whether the debugger can read opcodes from read-only sections of images on the host workstation rather than from the target itself.

Syntax

```
set trust-ro-sections-for-opcodes {off|on}
```

Where:

`off`	Disables this behavior. Use this option to trace self-modifying code or when the code on the target is modified before being loaded to the target. This is the default.
`on`	Enables reading opcodes from read-only sections of images on the host machine. Reading opcodes from the host workstation is usually faster than reading them from the target.

Example

Example 2-123 `set trust-ro-sections-for-opcodes`

```
set trust-ro-sections-for-opcodes on            # Enable reading opcodes from host
```

See also

- *show trust-ro-sections-for-opcodes* on page 2-204.

2.3.130 `set variable`

This command evaluates an expression and assigns the result to a variable, register or memory.

Syntax

`set [variable] expression`

Where:

expression	Specifies an expression and assigns the result to a variable, register or memory address.

Example

Example 2-124 `set variable`

```
set variable myVar=10                              # Assign 10 to variable myVar
set variable $PC=0x8000                            # Assign address 0x8000 to
                                                   # PC register
set variable $CPSR.N=0                             # Clear N bit
set variable (*(int*)0x8000)=1                     # Assign 1 to address 0x8000
set variable *0x8000=1                             # Assign 1 to address 0x8000
set variable strcpy((char*)0x8000,"My String")     # Assign string to address 0x8000
set variable memcpy(void*)0x8000,{10,20,30,40},4)  # Assign array to address 0x8000
```

See also

- *Using expressions* on page 2-4
- *info variables* on page 2-108
- *ARM Architecture Reference Manual*, http://infocenter.arm.com/help/topic/com.arm.doc.set.architecture/index.html.

2.3.131 `set wildcard-style`

This command specifies the type of wildcard pattern matching you can use for examining the contents of strings.

Syntax

```
set wildcard-style glob|regex
```

Where:

`glob`	Specifies a simpler style of pattern matching using glob expressions to refine your search. For example, `you can use m*` to search for strings starting with `m`. This is the default.
`regex`	Specifies a more complex style of pattern matching using regular expressions to refine your search. For example, you can use `my_lib[0-9]+` to search for strings starting with `my_lib` followed by an integer.

Example

Example 2-125 `set wildcard-style`

```
set wildcard-style regex              # Use regular expression pattern matching
```

See also

- *Using wildcards* on page 2-5
- *show wildcard-style* on page 2-206
- *info classes* on page 2-84
- *info functions* on page 2-89
- *info variables* on page 2-108
- *sharedlibrary* on page 2-176.

2.3.132 sharedlibrary

This command loads symbols from shared libraries. Be aware that it can only load symbols for shared libraries that are already loaded by the application.

Note

You must launch the debugger with `--target_os` command-line option before you can use this feature. In Eclipse this option is automatically selected when you connect to a target using `gdbserver`.

Syntax

`sharedlibrary [expression]`

Where:

expression	Specifies a library path or a wildcard expression. You can use wildcard expressions to enhance your pattern matching. If no *expression* is specified then the symbols from all shared libraries are loaded.

Example

Example 2-126 `sharedlibrary`

```
sharedlibrary                 # Load symbols from all shared libraries
sharedlibrary m*              # Load symbols matching path starting with m
                              # (use when set wildcard-style=glob)
sharedlibrary .*my_lib[0-9]+  # Load symbols matching path that ends with my_lib
                              # followed by a number(use when set wildcard-style=regex)
```

See also

- *Using wildcards* on page 2-5
- *info sharedlibrary* on page 2-101
- *nosharedlibrary* on page 2-130
- *set wildcard-style* on page 2-175.

2.3.133 `shell`

This command runs a shell command within the current debug session. The command is launched in the current working directory. You can use `pwd` to display the current working directory.

Syntax

`shell` *`cmd`*

Where:

`cmd`	Specifies the command and associated arguments.

Example

Example 2-127 `shell`

```
shell dir                  # On Windows, list of files in current directory
shell cat my_script.ds     # On Linux, list contents of my_script.ds file
```

See also

- *cd* on page 2-45
- *pwd* on page 2-135.

2.3.134 `show`

This command displays the current debugger settings.

Syntax

```
show
```

Example

Example 2-128 `show`

```
show                                # Display current debugger settings
```

2.3.135 `show architecture`

This command displays the architecture of the current target.

Syntax

```
show architecture
```

Example

Example 2-129 `show architecture`

```
show architecture              # Display current target architecture
```

2.3.136 `show arm`

This command displays the current instruction set settings in use by the debugger for disassembly and setting breakpoints.

Syntax

`show arm` *`option`*

Where:

`option`	Specifies additional options:	
	`force-mode`	Display the current force-mode behavior.
	`fallback-mode`	Display the current fallback-mode behavior.

Example

Example 2-130 `show arm`

```
show arm                         # Display the current instruction set settings
show arm force-mode              # Display the current force-mode setting
```

See also

- *info inst-sets* on page 2-91
- *set arm* on page 2-146.

2.3.137 `show auto-solib-add`

This command displays the current automatic setting for use when loading shared library symbols. You can use the `set auto-solib-add` command to modify this setting.

Note

You must launch the debugger with `--target_os` command-line option before you can use this feature. In Eclipse this option is automatically selected when you connect to a target using `gdbserver`.

Syntax

```
show auto-solib-add
```

Example

Example 2-131 `show auto-solib-add`

```
show auto-solib-add         # display current automatic setting for loading
                            # shared library symbols
```

See also

- *set auto-solib-add* on page 2-148.

2.3.138 `show backtrace`

This command displays current behavior settings for use with the `info stack` command. You can use the `set backtrace` commands to modify these settings.

Syntax

`show backtrace` *`option`*

Where:

`option`	Specifies additional options:	
	`limit`	Displays the current limit when listing the call stack.

Example

Example 2-132 `show backtrace`

```
show backtrace limit              # Display current call stack limit
```

See also

- *info stack, backtrace, where* on page 2-104
- *set backtrace* on page 2-149.

2.3.139 `show blocking-run-control`

This command displays the current run control setting that defines whether run control operations such as stepping and running are blocked until the target stops or released immediately. You can use the `set blocking-run-control` command to modify this setting.

Syntax

```
show blocking-run-control
```

Example

Example 2-133 `show blocking-run-control`

```
show blocking-run-control         # Display current run control setting
```

See also

- *set blocking-run-control* on page 2-150.

2.3.140 `show breakpoint`

This command displays current breakpoint and watchpoint behavior settings. You can use the `set breakpoint` commands to modify these settings.

Syntax

`show breakpoint` *`option`*

Where:

`option`	Specifies additional options:

`auto-hw`	Displays the automatic breakpoint selection setting. The debugger uses this option to decide what type of breakpoint it must use automatically when using the `break` command.
`skipmode`	Displays the breakpoint and watchpoint skipmode setting.

Example

Example 2-134 `show breakpoint`

```
show breakpoint auto-hw      # Display automatic breakpoint selection setting
show breakpoint skipmode     # Display breakpoint and watchpoint skipmode setting
```

See also

- *set breakpoint* on page 2-151.

2.3.141 `show case-insensitive-source-matching`

This command displays the current case sensitivity setting for the debugger file matching operations. You can use the `set case-insensitive-source-matching` command to modify this setting.

Syntax

```
show case-insensitive-source-matching
```

Example

Example 2-135 `show case-insensitive-source-matching`

```
show case-insensitive-source-matching        # Display current case sensitivity setting
```

See also

- *set case-insensitive-source-matching* on page 2-152
- *show escapes-in-filenames* on page 2-192
- *show wildcard-style* on page 2-206.

2.3.142 `show debug-agent`

This command displays the current value of a parameter in the launch configuration for DSTREAM/RVI connections. You can use the `set debug-agent` command to modify this setting.

Syntax

`show debug-agent [`*`option`*`]`

Where:

`option`	Displays the current setting for the specified parameter.

Example

Example 2-136 `show debug-agent`

```
show debug-agent         # Display all current debug agent configuration parameters
```

See also

- *set debug-agent* on page 2-153.

2.3.143 `show debug-from`

This command displays the current setting for the expression that is used by the `start` command to set a temporary breakpoint. You can use the `set debug-from` command to modify this setting.

Syntax

```
show debug-from
```

Example

Example 2-137 `show debug-from`

```
show debug-from          # Display current expression used by start command
```

See also

- *Using expressions* on page 2-4
- *start* on page 2-209
- *set debug-from* on page 2-154.

2.3.144 `show directories`

This command displays the list of directories to search for source files. You can use the `directory` command to modify this list.

Syntax

```
show directories
```

Example

Example 2-138 `show directories`

```
show directories                  # Display list of search paths
```

See also

- *directory* on page 2-54.

2.3.145 show dtsl-options

This command displays the current value of a parameter in the connection DTSL configuration. You can use the `set dtsl-options` command to modify this setting.

Syntax

`show dtsl-options [`*`option`*`]`

Where:

`option`	Displays the current setting for the specified parameter.

Example

Example 2-139 `show dtsl-options`

```
show dtsl-options          # Display all DTSL configuration parameters
```

See also

- *set dtsl-options* on page 2-156.

2.3.146 `show endian`

This command displays the current byte order setting in use by the debugger. You can use the `set endian` command to modify this setting.

Syntax

```
show endian
```

Example

Example 2-140 `show endian`

```
show endian                     # Display current byte order setting
```

See also

- *set endian* on page 2-157.

2.3.147 `show escape-strings`

This command displays the current setting for controlling how special characters in strings are printed on the debugger command-line. You can use the `set escape-strings` command to modify this setting.

Syntax

`show escape-strings`

Example

Example 2-141 `show escape-strings`

```
show escape-strings          # Display current setting for controlling
                             # how special characters in strings are printed
```

See also

- *set escape-strings* on page 2-158.

2.3.148 `show escapes-in-filenames`

This command displays the current setting for controlling the use of special characters in paths. You can use the `set escapes-in-filenames` command to modify this setting.

Syntax

```
show escapes-in-filenames
```

Example

Example 2-142 `show escapes-in-filenames`

```
show escapes-in-filenames           # Display current setting for controlling
                                    # the use of special characters in paths
```

See also

- *set escapes-in-filenames* on page 2-159.

2.3.149 `show listsize`

This command displays the number of source lines that the `list` command displays. You can use the `set listsize` command to modify the display size.

Syntax

```
show listsize
```

Example

Example 2-143 `show listsize`

```
show listsize                    # Display listing size for list command
```

See also

- *list* on page 2-112
- *set listsize* on page 2-160.

2.3.150 `show os`

This command displays the current setting for controlling the *Operating System* (OS) settings. You can use the `set os` command to modify these settings.

Note

An OS aware connection must be established before you can use this command.

Syntax

`show os` *`option`*

Where:

`option`	Specifies additional options:	
	`log-capture`	Displays the current setting for controlling the capturing and printing of OS logging messages.
	`enabled`	Displays the current setting for controlling OS support.

Example

Example 2-144 `show os`

```
show os log-capture          # Display setting for controlling os log capture
show os enabled              # Display OS enabled setting
```

See also

- *info os-log* on page 2-95
- *info os-modules* on page 2-96
- *info os-version* on page 2-97
- *info processes* on page 2-98
- *set os* on page 2-161.

2.3.151 `show print`

This command displays the current debugger print settings. You can use the `set print` commands to modify these settings.

Syntax

`show print` *`option`*

Where:

`option` Specifies additional options:

`library-not-found-warnings`
: Displays the print settings for "unable to find library..." messages.

`full-source-path`
: Displays the print settings for source paths in messages.

`stop-info`
: Displays the print settings for event messages when the target stops.

`current-vmid`
: Displays the print settings for VMID messages when the target stops.

`double-format`
: Displays the print settings that controls the `printf()` style formatting of double values.

`float-format`
: Displays the print settings that controls the `printf()` style formatting of floating-point values.

Example

Example 2-145 `show print`

```
show print library-not-found-warnings   # Display print settings for unfound
                                        # library messages
show print full-source-path             # Display print settings for
                                        # source paths in messages
```

See also

- *set print* on page 2-163
- *printf() style format string* on page 2-9.

2.3.152 `show semihosting`

This command displays the current semihosting settings in the debugger. You can use the `set semihosting` commands to modify these settings.

Syntax

`show semihosting` *`option`*

Where:

`option`	Specifies additional options:	
	`args`	Displays the command-line arguments that are passed to the `main()` function in the application.
	`enabled`	Displays the semihosting enabled setting.
	`file-base`	Displays the setting for the `file-base` directory.
	`stdin`	Displays the `stdin` settings.
	`stdout`	Displays the `stdout` settings.
	`stderr`	Displays the `stderr` settings.
	`top-of-memory`	Displays the address for the top of memory.
	`stack-base`	Displays the address for the stack base.
	`stack-limit`	Displays the address for the stack limit.
	`heap-base`	Displays the address for the heap base.
	`heap-limit`	Displays the address for the heap limit.
	`vector`	When using a semihosting breakpoint, the address is displayed otherwise a message is displayed indicating that a vector is in use.

Example

Example 2-146 `show semihosting`

```
show semihosting args             # Display command-line arguments
show semihosting enabled          # Display semihosting enabled setting
show semihosting top-of-memory    # Display the top of memory address
```

See also

- *set semihosting* on page 2-165.

2.3.153 `show solib-absolute-prefix`

`show solib-absolute-prefix` is an alias for `show sysroot`.

See *show sysroot, show solib-absolute-prefix* on page 2-203.

2.3.154 `show solib-search-path`

This command displays the current search paths in use by the debugger when searching for shared libraries. You can use the `set sysroot` command to specify a system root directory on the host workstation and you can also use the `set solib-search-path` command to specify additional directories.

Note

You must launch the debugger with `--target_os` command-line option before you can use this feature. In Eclipse this option is automatically selected when you connect to a target using `gdbserver`.

Syntax

```
show solib-search-path
```

Example

Example 2-147 `show solib-search-path`

```
show solib-search-path            # Display search path for shared libraries
```

See also

- *set solib-search-path* on page 2-168
- *set sysroot, set solib-absolute-prefix* on page 2-172
- *show sysroot, show solib-absolute-prefix* on page 2-203.

2.3.155 `show step-mode`

This command displays the current step setting for functions without debug information. You can use the `set step-mode` command to modify this setting.

Syntax

```
show step-mode
```

Example

Example 2-148 `show step-mode`

```
show step-mode          # Display current step setting (function without debug)
```

See also

- *set step-mode* on page 2-169
- *step* on page 2-211
- *steps* on page 2-213.

2.3.156 `show stop-on-solib-events`

This command displays the current debugger setting that controls whether execution stops when shared library events occur. You can use the `set stop-on-solib-events` command to modify this setting.

Note

You must launch the debugger with `--target_os` command-line option before you can use this feature. In Eclipse this option is automatically selected when you connect to a target using `gdbserver`.

Syntax

```
show stop-on-solib-events
```

Example

Example 2-149 `show stop-on-solib-events`

```
show stop-on-solib-events    # Display stop setting for shared library events
```

See also

- *set stop-on-solib-events* on page 2-170.

2.3.157 `show substitute-path`

This command displays the current search path substitution rules in use by the debugger when searching for source files. You can use the `set substitute-path` command to modify these substitution rules.

Syntax

```
show substitute-path
```

Example

Example 2-150 `show substitute-path`

```
show substitute-path          # Display all substitution rules
```

See also

- *directory* on page 2-54
- *set substitute-path* on page 2-171.

2.3.158 `show sysroot, show solib-absolute-prefix`

This command displays the system root directory in use by the debugger when searching for shared library symbols. You can use the `set sysroot` command to specify a system root directory on the host workstation.

The debugger uses this directory to search for a copy of the debug versions of target shared libraries. The system root on the host workstation must contain an exact representation of the libraries on the target root filesystem.

Note

You must launch the debugger with `--target_os` command-line option before you can use this feature. In Eclipse this option is automatically selected when you connect to a target using `gdbserver`.

Syntax

`show sysroot`

`show solib-absolute-prefix`

Example

Example 2-151 `show sysroot, show solib-absolute-prefix`

```
show sysroot                  # Display system root directory
```

See also

- *set solib-search-path* on page 2-168
- *set sysroot, set solib-absolute-prefix* on page 2-172
- *show solib-search-path* on page 2-199.

2.3.159 `show trust-ro-sections-for-opcodes`

This command displays the current debugger setting that controls whether the debugger can read opcodes from read-only sections of images on the host workstation rather than from the target itself.

Syntax

```
show trust-ro-sections-for-opcodes
```

Example

Example 2-152 `show trust-ro-sections-for-opcodes`

```
show trust-ro-sections-for-opcodes    # Display trust-ro-sections-for-opcodes setting
```

See also

- *set trust-ro-sections-for-opcodes* on page 2-173.

2.3.160 `show version`

This command displays the current version number of the debugger.

Syntax

```
show version
```

Example

Example 2-153 `show version`

```
show version                           # Display debugger version number
```

2.3.161 `show wildcard-style`

This command displays the current wildcard style in use for pattern matching. You can use the `set wildcard-style` command to modify this setting.

Syntax

```
show wildcard-style
```

Example

Example 2-154 `show wildcard-style`

```
show wildcard-style                # Display current wildcard style
```

See also

- *Using wildcards* on page 2-5
- *set wildcard-style* on page 2-175
- *info classes* on page 2-84
- *info functions* on page 2-89
- *info variables* on page 2-108
- *sharedlibrary* on page 2-176.

2.3.162 `silence`

This command disables the printing of stop messages for a specific breakpoint.

Syntax

`silence [`*`number`*`]`

Where:

`number`	Specifies the breakpoint number. This is the number assigned by the debugger when it is set. You can use `info breakpoints` to display the number and status of all breakpoints and watchpoints.

If no *`number`* is specified then all stop messages are disabled.

Example

Example 2-155 `silence`

```
silence 2                          # Disable printing of stop messages for breakpoint 2
silence $                          # This applies to the breakpoint whose number is in
                                   # the most recently created debugger variable
```

See also

- *set print* on page 2-163
- *unsilence* on page 2-223.

2.3.163 `source`

This command loads and runs a script file to control and debug your target.

The following types of scripts are available:

DS-5	DS-5 Debugger commands.
CMM	CMM is a scripting language supported by some third-party debuggers. DS-5 supports a small subset of CMM-style commands, sufficient for running small target initialization scripts.
Jython	Jython is a Java implementation of the Python scripting language. It provides extensive support for data types, conditional execution, loops and organization of code into functions, classes and modules, as well as access to the standard Jython libraries. Jython is an ideal choice for larger or more complex scripts.

Syntax

`source [/v] *filename* [*args*]`

Where:

`v`	Specifies verbose output. Script commands are interleaved with the debugger output.
`filename`	Specifies the script file. The following file extensions must be used to identify the script type:
	.ds for DS-5 scripts
	.cmm, .t32 for CMM scripts
	.py for Jython scripts.
`args`	Zero or more arguments to pass to the script (only supported for Jython scripts).

Example

Example 2-156 `source`

```
source myScripts\myFile.ds        # Run DS-5 Debugger commands from myFile.ds
source myScripts\myFile.cmm       # Run CMM-style commands from myFile.cmm
source myScripts\myFile.t32       # Run CMM-style commands from myFile.t232
source /v myFile.ds               # Run DS-5 Debugger commands from myFile.ds and
                                  # display commands interleaved with debugger output
source myScripts\myFile.py        # Run a Jython script from file myFile.py
```

2.3.164 `start`

This command sets a temporary breakpoint, calls the debugger `run` command and then deletes the temporary breakpoint. By default, the temporary breakpoint is set at the address of the global function `main()`. You can use the `set debug-from` command to change the breakpoint location. If the breakpoint location cannot be found then the breakpoint is set at the image entry point.

This command records the ID of the breakpoint in a new debugger variable, $*n*, where *n* is a number. If $*n* is the last or second-to-last debugger variable, then you can also access the ID using $ or $$, respectively.

Note

Control is returned as soon as the target is running. You can use the `wait` command to block the debugger from returning control until either the application completes or a breakpoint is hit.

Syntax

`start [`*`args`*`]`

Where:

args	Specifies the command-line arguments that are passed to the `main()` function in the application using the `argv` parameter. The name of the image is always implicitly passed in `argv[0]` and it is not necessary to pass this as an argument.

Example

Example 2-157 `start`

```
start                                  # Start running the target to the
                                       # temporary breakpoint
```

See also

- *continue* on page 2-49
- *file, symbol-file* on page 2-68
- *load* on page 2-114
- *loadfile* on page 2-115
- *run* on page 2-141
- *set arm* on page 2-146
- *set debug-from* on page 2-154
- *set semihosting* on page 2-165
- *show debug-from* on page 2-187
- *show semihosting* on page 2-196
- *wait* on page 2-226.

2.3.165 `stdin`

This command is only for use with semihosted applications when using the debugger interactively in the command-line console.

Note

This command is not required if you launch the debugger within Eclipse or if you use a telnet session to interact directly with the application.

Syntax

`stdin [input]`

Where:

input	Specifies semihosting input requested by application code. This must be terminated by `\n` to tell the debugger that the input is complete.

You can use this command before the input is required by the application code. All input is buffered by the debugger until requested and then discarded when the semihosting operation finishes.

Example

Example 2-158 `stdin`

```
stdin 10000\n               # Pass the number 10000 to the application
```

2.3.166 `step`

This command steps through an application at the source level stopping on the first instruction of each source line including stepping into all function calls. You must compile your code with debug information to use this command successfully.

You can modify the behavior of this command with the `set step-mode` command.

Syntax

`step [`*`count`*`]`

Where:

`count`	Specifies the number of source lines to execute. **Note** Execution stops immediately if a breakpoint is reached, even if fewer than *`count`* source lines are executed.

Example

Example 2-159 `step`

```
step                                  # Execute one source line
step 5                                # Execute five source lines
```

See also

- *finish* on page 2-70
- *next* on page 2-127
- *nexti* on page 2-128
- *nexts* on page 2-129
- *set step-mode* on page 2-169
- *show step-mode* on page 2-200
- *stepi* on page 2-212
- *steps* on page 2-213.

2.3.167 stepi

This command steps through an application at the instruction level including stepping into all function calls.

Syntax

`stepi [count]`

Where:

count Specifies the number of instructions to execute.

Note

Execution stops immediately if a breakpoint is reached, even if fewer than *count* instructions are executed.

Example

Example 2-160 stepi

```
stepi                                  # Execute one instruction
stepi 5                                # Execute five instructions
```

See also

- *next* on page 2-127
- *nexti* on page 2-128
- *nexts* on page 2-129
- *step* on page 2-211
- *steps* on page 2-213.

2.3.168 `steps`

This command steps through an application at the source level stopping on the first instruction of each source statement (for example, statements in a `for()` loop) including stepping into all function calls. You must compile your code with debug information to use this command successfully.

You can modify the behavior of this command with the `set step-mode` command.

Syntax

`steps [count]`

Where:

`count`	Specifies the number of source statements to execute. **Note** Execution stops immediately if a breakpoint is reached, even if fewer than *`count`* source statements are executed.

Example

Example 2-161 `steps`

```
steps                               # Execute one source statement
steps 5                             # Execute five source statements
```

See also

- *finish* on page 2-70
- *next* on page 2-127
- *nexti* on page 2-128
- *nexts* on page 2-129
- *set step-mode* on page 2-169
- *show step-mode* on page 2-200
- *step* on page 2-211
- *stepi* on page 2-212.

2.3.169 `stop`

`stop` is an alias for `interrupt`.

See *interrupt, stop* on page 2-111.

2.3.170 `symbol-file`

`symbol-file` is an alias for `file`.

See *file, symbol-file* on page 2-68.

2.3.171 tbreak

This command sets an execution breakpoint at a specific location and subsequently deletes it when the breakpoint is hit. You can also specify a conditional breakpoint by using an `if` statement that stops only when the conditional expression evaluates to true.

This command records the ID of the breakpoint in a new debugger variable, $*n*, where *n* is a number. You can use this variable, in a script, to delete or modify the breakpoint behavior. If $*n* is the last or second-to-last debugger variable, then you can also access the ID using $ or $$, respectively.

Note

Breakpoints that are set within a shared object or kernel module become pending when the shared object or kernel module is unloaded.

Use `set breakpoint` to control the automatic breakpoint behavior when using this command.

Syntax

`tbreak [-d] [-p] [[`*`filename`*`:]`*`location`*`|*`*`address`*`] [thread|core `*`number…`*`] [if `*`expression`*`]`

Where:

d	Disables the breakpoint immediately after creation.
p	Specifies whether or not the resolution of an unrecognized breakpoint location results in a pending breakpoint being created.
filename	Specifies the file.
location	Specifies the location:
	line_num — is a line number
	function — is a function name.
	label — is a label name.
	+offset|*-offset* — Specifies the line offset from the current location.
address	Specifies the address. This can be either an address or an expression that evaluates to an address.
number	Specifies one or more threads or processors to apply the breakpoint to. You can use `$thread` to refer to the current thread. If *number* is not specified then all threads are affected.
expression	Specifies an expression that is evaluated when the breakpoint is hit.

If no arguments are specified then a breakpoint is set at the current PC.

Example

Example 2-162 `tbreak`

```
tbreak *0x8000               # Set breakpoint at address 0x8000
tbreak *0x8000 thread $thread # Set breakpoint at address 0x8000 on
                             # current thread
tbreak *0x8000 thread 1 3    # Set breakpoint at address 0x8000 on
                             # threads 1 and 3
```

```
tbreak main                 # Set breakpoint at address of main()
tbreak SVC_Handler          # Set breakpoint at address of label SVC_Handler
tbreak +1                   # Set breakpoint at address of next source line
tbreak my_File.c:main       # Set breakpoint at address of main() in my_File.c
tbreak my_File.c:8          # Set breakpoint at address of line 8 in my_File.c
tbreak function1 if x>0     # Set conditional breakpoint that stops when x>0
```

See also

- *Using expressions* on page 2-4
- *advance* on page 2-33
- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *condition* on page 2-48
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *hbreak* on page 2-74
- *ignore* on page 2-78
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *resolve* on page 2-139
- *set arm* on page 2-146
- *thbreak* on page 2-218.

2.3.172 `thbreak`

This command sets a hardware execution breakpoint at a specific location and subsequently deletes it when the breakpoint is hit. You can also specify a conditional breakpoint by using an `if` statement that stops only when the conditional expression evaluates to true.

This command records the ID of the breakpoint in a new debugger variable, $*n*, where *n* is a number. You can use this variable, in a script, to delete or modify the breakpoint behavior. If $*n* is the last or second-to-last debugger variable, then you can also access the ID using $ or $$, respectively.

——— **Note** ———

The number of hardware breakpoints are usually limited. If you run out of hardware breakpoints then delete or disable one that you are no longer using.

Breakpoints that are set within a shared object or kernel module become pending when the shared object or kernel module is unloaded.

———————

Syntax

```
thbreak [-d] [-p] [[filename:]location|*address] [thread|core number…] [vmid vmid] [if
expression]
```

Where:

d	Disables the breakpoint immediately after creation.
p	Specifies whether or not the resolution of an unrecognized breakpoint location results in a pending breakpoint being created.
filename	Specifies the file.
location	Specifies the location:

line_num	is a line number.
function	is a function name.
label	is a label name.
+offset|*-offset*	Specifies the line offset from the current location.

number	Specifies one or more threads or processors to apply the breakpoint to. You can use `$thread` to refer to the current thread. If *number* is not specified then all threads are affected.
address	Specifies the address. This can be either an address or an expression that evaluates to an address.
vmid	Specifies the *Virtual Machine ID* (VMID) to apply the breakpoint to. This can be either an integer or an expression that evaluates to an integer.
expression	Specifies an expression that is evaluated when the breakpoint is hit.

If no arguments are specified then a hardware breakpoint is set at the next instruction.

Example

Example 2-163 `thbreak`

```
thbreak *0x8000              # Set breakpoint at address 0x8000
thbreak *0x8000 thread $thread # Set breakpoint at address 0x8000 on
                               # current thread
thbreak *0x8000 thread 1 3     # Set breakpoint at address 0x8000 on
                               # threads 1 and 3
thbreak main                   # Set breakpoint at address of main()
thbreak SVC_Handler            # Set breakpoint at address of label SVC_Handler
thbreak +1                     # Set breakpoint at address of next source line
thbreak my_File.c:main         # Set breakpoint at address of main(), my_File.c
thbreak my_File.c:8            # Set breakpoint at address of line 8, my_File.c
thbreak function1 if x>0       # Set conditional breakpoint that stops when x>0
```

See also

- *Using expressions* on page 2-4
- *break* on page 2-38
- *break-script* on page 2-40
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clear* on page 2-46
- *condition* on page 2-48
- *delete breakpoints* on page 2-52
- *disable breakpoints* on page 2-55
- *enable breakpoints* on page 2-64
- *hbreak* on page 2-74
- *ignore* on page 2-78
- *info breakpoints, info watchpoints* on page 2-81
- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *resolve* on page 2-139
- *tbreak* on page 2-216.

2.3.173 `thread`, `core`

This command displays the following information:

- Unique *id* number assigned by the debugger.
- Thread or processor state. For example, stopped or running.
- Current stack frame including function names and source line numbers.

Syntax

`thread [`*id*`]`

`core [`*id*`]`

Where:

id	Specifies the unique thread or processor number. If *id* is not specified then the current thread or processor is displayed. You can use `info cores`, `info processes`, or `info threads` to display the *id* numbers.

If *id* is specified then the debugger switches control to that thread or processor before displaying the information. Registers and call stacks are associated with a particular thread or processor. This means that switching context also switches the registers and call stack to those belonging to the current thread or processor.

Example

Example 2-164 `thread`, `core`

```
thread 699                         # Set current thread to number 699
core 2                             # Set current processor to number 2
```

See also

- *break* on page 2-38
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *info cores* on page 2-85
- *info processes* on page 2-98
- *info threads* on page 2-107
- *thread apply, core apply* on page 2-221.

2.3.174 `thread apply`, `core apply`

This command temporarily switches control to a specific thread or processor to execute a DS-5 Debugger command and then switches back to the original state.

If an error occurs then the debugger stops processing the command and switches back to the original state.

Syntax

`thread apply {all|`*`id`*`}` *`command`*

`core apply {all|`*`id`*`}` *`command`*

Where:

`all`	Specifies all threads or all processors.
`id`	Specifies the unique thread or processor number. You can use `info cores`, `info processes`, or `info threads` to display the *`id`* numbers.
`command`	Specifies the DS-5 Debugger command that you want to execute.

If `all` is specified then the command is executed on each thread or processor successively before switching back.

Example

Example 2-165 `thread apply`, `core apply`

```
thread apply all print /x $pc        # Cycle through all threads and print address
                                     # in PC register (hexadecimal)
```

See also

- *break* on page 2-38
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *info cores* on page 2-85
- *info processes* on page 2-98
- *info threads* on page 2-107
- *thread, core* on page 2-220.

2.3.175 `unset`

This command modifies the current debugger settings.

Syntax

`unset` *`option`*

Where:

`option` Specifies additional options:

`substitute-path [`*`path`*`]`
: Deletes all the substituted source paths. If *`path`* is specified then only the substitution for *`path`* is deleted.

`semihosting heap-base`
: Deletes the base address of the heap.

`semihosting heap-limit`
: Deletes the end address of the heap.

`semihosting stack-base`
: Deletes the base address of the stack.

`semihosting stack-limit`
: Deletes the end address of the stack.

`semihosting top-of-memory`
: Deletes the top of memory.

Example

Example 2-166 `unset`

```
unset substitute-path               # Delete all substitution paths
```

See also

- *set semihosting* on page 2-165
- *set substitute-path* on page 2-171.

2.3.176 `unsilence`

This command enables the printing of stop messages for a specific breakpoint.

Syntax

`unsilence [`*`number`*`]`

Where:

`number`	Specifies the breakpoint number. This is the number assigned by the debugger when it is set. You can use `info breakpoints` to display the number and status of all breakpoints and watchpoints.

If no *`number`* is specified then all stop messages are enabled.

Example

Example 2-167 `unsilence`

```
unsilence 2                  # Enable printing of stop messages for breakpoint 2
unsilence $                  # This applies to the breakpoint whose number is in
                             # the most recently created debugger variable
```

See also

- *set print* on page 2-163
- *silence* on page 2-207.

2.3.177 `up`

This command moves the current frame pointer up the call stack towards the top frame. It also displays the function name and source line number for the specified frame.

——— **Note** ———

Each frame is assigned a number that increases from the bottom frame (zero) through the call stack to the top frame that is the start of the application.

———

Syntax

`up [`*`offset`*`]`

Where:

`offset`	Specifies a frame offset from the current frame pointer in the call stack. If no *`offset`* is specified then the default is one.

Example

Example 2-168 `up`

```
up        # Move and display information 1 frame up from current frame pointer
up 2      # Move and display information 2 frames up from current frame pointer
```

See also

- *down* on page 2-60
- *down-silently* on page 2-61
- *info frame* on page 2-88
- *info all-registers* on page 2-80
- *info registers* on page 2-99
- *info stack, backtrace, where* on page 2-104
- *finish* on page 2-70
- *frame* on page 2-72
- *select-frame* on page 2-144
- *up-silently* on page 2-225.

2.3.178 `up-silently`

This command moves the current frame pointer up the call stack towards the top frame.

Note

Each frame is assigned a number that increases from the bottom frame (zero) through the call stack to the top frame that is the start of the application.

Syntax

`up-silently [`*`offset`*`]`

Where:

`offset`	Specifies a frame offset from the current frame pointer in the call stack. If no *`offset`* is specified then the default is one.

Example

Example 2-169 `up-silently`

```
up-silently                    # Move 1 frame up from current frame pointer
up-silently 2                  # Move 2 frames up from current frame pointer
```

See also

- *down* on page 2-60
- *down-silently* on page 2-61
- *info frame* on page 2-88
- *info all-registers* on page 2-80
- *info registers* on page 2-99
- *info stack, backtrace, where* on page 2-104
- *finish* on page 2-70
- *frame* on page 2-72
- *select-frame* on page 2-144
- *up* on page 2-224.

2.3.179 `wait`

This command instructs the debugger to wait until the target stops. For example, when the application completes or a breakpoint is hit. ARM recommends that you specify a time-out parameter to generate an error if the time-out value is reached.

Syntax

`wait` *`time-out`*`[ms | s]`

Where:

`time-out`	Specifies the period of time.
`ms`	Specifies the time in milliseconds. This is the default.
`s`	Specifies the time in seconds.

Example

Example 2-170 `wait`

```
wait 1000                          # Wait or time-out after 1 second
wait 0.5s                          # Wait or time-out after half a second
```

See also

- *continue* on page 2-49
- *run* on page 2-141
- *start* on page 2-209.

2.3.180 `watch`

This command sets a watchpoint for a data symbol. The debugger stops the target when the memory at the specified address is written.

This command records the ID of the watchpoint in a new debugger variable, $*n*, where *n* is a number. You can use this variable, in a script, to delete or modify the watchpoint behavior. If $*n* is the last or second-to-last debugger variable, then you can also access the ID using $ or $$, respectively.

——— **Note** ———

Watchpoints are only supported on scalar values.

Some targets do not support watchpoints. Currently you can only set a watchpoint on a hardware target using a debug hardware agent.

The address of the instruction that triggers the watchpoint might not be the address shown in the PC register. This is because of pipelining effects.

———————

Syntax

`watch [-d] [-p] {[`*`filename`*`:]`*`symbol`*`|`*`*address`*`} [vmid `*`vmid`*`]`

Where:

d	Disables the watchpoint immediately after creation.
p	Specifies whether or not the resolution of an unrecognized watchpoint location results in a pending watchpoint being created.
filename	Specifies the file.
symbol	Specifies a global/static data symbol. For arrays or structs you must specify the element or member.
address	Specifies the address. This can be either an address or an expression that evaluates to an address.
vmid	Specifies the *Virtual Machine ID* (VMID) to apply the breakpoint to. This can be either an integer or an expression that evaluates to an integer.

Example

Example 2-171 `watch`

```
watch myVar1                      # Set write watchpoint on myVar1
watch *0x80D4                     # Set write watchpoint on address 0x80D4
```

See also

- *Using expressions* on page 2-4
- *awatch* on page 2-35
- *break-stop-on-threads, break-stop-on-cores* on page 2-43
- *break-stop-on-vmid* on page 2-44
- *clearwatch* on page 2-47
- *info breakpoints, info watchpoints* on page 2-81

- *info breakpoints capabilities, info watchpoints capabilities* on page 2-82
- *rwatch* on page 2-142.

2.3.181 `whatis`

This command displays the data type of an expression.

Syntax

`whatis [`*`expression`*`]`

Where:

`expression`	Specifies an expression. If no *`expression`* is specified then the last expression is repeated.

Note

This command does not execute the expression.

Example

Example 2-172 `whatis`

```
whatis 4+4                          # Display data type of expression result
whatis myVar                        # Display data type of variable (myVar)
```

See also

- *Using expressions* on page 2-4.

2.3.182 `where`

`where` is an alias for `info stack`.

See *info stack, backtrace, where* on page 2-104.

2.3.183 `while`

This command enables you to write scripts with conditional loops that execute debugger commands.

Syntax

```
while condition
   ...
   optional_commands
   ...
end
```

Where:

`condition`	Specifies a conditional expression. Follow the `while` statement with one or more debugger commands that execute repeatedly while *`condition`* evaluates to true.
`optional_commands`	Specifies optional commands that can also be used inside the `while` statement to change the loop behavior:

`loop_break`	Exit the loop.
`loop_continue`	Skip the remaining commands and return to the start of the loop.

Enter each debugger command on a new line and terminate the `while` command by using the `end` command.

Example

Example 2-173 `while`

```
# Define a while loop containing commands to conditionally execute
# myVar is a variable in the application code
while myVar<10
    step
    wait
    x
    set myVar++
end
```

See also

- *define* on page 2-51
- *document* on page 2-59
- *end* on page 2-66
- *if* on page 2-77
- *Using expressions* on page 2-4.

2.3.184 x

This command displays the content of memory at a specific address.

Syntax

`x [/flag]… [address]`

Where:

flag

Specifies additional flags:

count	Specifies the number of values to display. If none specified then the default is `1`.

Size of memory:

`b`	`1` byte
`h`	`2` bytes
`w`	`4` bytes (default)
`g`	`8` bytes.

Output format:

`x`	hexadecimal (casts the value to an unsigned integer prior to printing in hexadecimal)
`d`	signed decimal
`u`	unsigned decimal
`o`	octal
`t`	binary
`a`	absolute hexadecimal address
`c`	character
`f`	floating-point
`i`	assembler instruction.

Note

If no output format is specified then the initial default is `x`, unless preceded by another command using output format options in which case the same format is retained.

address

Specifies the address. This can be either an address, a symbol name, or an expression that evaluates to an address. If no *address* is specified then the default value is used. Some commands that access memory can set this default value. For example, `x`, `print`, `output` and `info breakpoints`.

Note

This command sets a default address variable to the location after the last accessed address.

Example

Example 2-174 x

```
x 0x8000       # Display memory at address 0x8000
x/3wx 0x8000   # Display 3 words of memory from address 0x8000 (hexadecimal)
x/4b $SP       # Display 4 bytes of memory from address in SP register
x/4i $PC       # Display 4 instructions from address in PC register
x /h 0x8000    # Read a half-word from address 0x8000
```

See also

- *Using expressions* on page 2-4
- *disassemble* on page 2-57
- *info breakpoints, info watchpoints* on page 2-81
- *memory set* on page 2-123
- *memory set_typed* on page 2-125
- *output* on page 2-131
- *print, inspect* on page 2-134
- *set arm* on page 2-146.

Chapter 3
CMM-style commands supported by the debugger

The following topics describe the CMM-style commands:

3.1 General syntax and usage of CMM-style commands

CMM-style commands are a small subset of commands, sufficient for running target initialization scripts. CMM is a scripting language supported by some third-party debuggers.

Note

For full debug support ARM recommends that you use the DS-5 Debugger commands. See Chapter 2 *DS-5 Debugger commands* for more information.

Syntax of CMM-style commands

Many commands accept arguments and flags using the following syntax:

command [*argument*] [/*flag*]…

A flag acts as an optional switch and is introduced with a forward slash character. Where a command supports flags, the flags are described as part of the command syntax.

Note

Commands are not case sensitive. Abbreviations are underlined.

Usage of CMM-style commands

The commands you submit to the debugger must conform to the following rules:

- Each command line can contain only one debugger command.
- When referring to symbols, you must use the same case as the source code.

To execute CMM-style commands you must create a debugger script file containing the CMM-style commands and then use the DS-5 Debugger `source` command to run the script.

Many commands can be abbreviated. For example, `break.set` can be abbreviated to `b.s`. The syntax definition for each command shows how it can be abbreviated by underlining it for example, `break.set`.

In the syntax definition of each command:

- square brackets [...] enclose optional parameters
- braces {...} enclose required parameters
- a vertical pipe | indicates alternatives from which you must choose one
- parameters that can be repeated are followed by an ellipsis (...).

Do not type square brackets, braces, or the vertical pipe. Replace parameters in italics with the value you want. When you supply more than one parameter, use the separator as shown in the syntax definition for each command. If a parameter is a name that includes spaces, enclose it in double quotation marks.

Descriptive comments can be placed either at the end of a command or on a separate line. You can use either `//` or `;` to identify a descriptive comment.

3.1.1 Using expressions

Some commands accept expressions. In an expression you can access the content of registers and variables by using a function-like notation, for example:

```
print "The result of my expression is: " v.value(myVar)+4+r(R0)
```

where `v.value()` can be used to access the content of a variable and `r()` can be used to access the content of a register.

3.2 CMM-style commands listed in groups

The supported CMM-style commands grouped according to specific tasks are:

- *Controlling breakpoints*
- *Controlling data and display settings*
- *Controlling images, symbols, and libraries* on page 3-4
- *Controlling target execution and connections* on page 3-4
- *Displaying the call stack and associated variables* on page 3-4
- *Controlling the debugger and program information* on page 3-4
- *Supporting commands* on page 3-5.

3.2.1 Controlling breakpoints

List of commands:

break.delete on page 3-8

Deletes a specific breakpoint.

break.disable on page 3-9

Disables a specific breakpoint.

break.enable on page 3-10

Enables a specific breakpoint.

break.set on page 3-11

Sets a breakpoint at a specific address.

Type `help` followed by a command name for more information on a specific command.

3.2.2 Controlling data and display settings

List of commands:

data.dump on page 3-12

Displays data at a specific address or address range.

data.set on page 3-15

Writes data to memory.

print on page 3-18

Displays the output of an expression.

register.set on page 3-19

Sets the value of a register.

var.global on page 3-23

Displays all global variables.

var.local on page 3-24

Displays all local variables.

var.print on page 3-26

Displays the output of an expression.

Type `help` followed by a command name for more information on a specific command.

3.2.3 Controlling images, symbols, and libraries

List of commands:

data.load.binary on page 3-13

Loads a binary image file.

data.load.elf on page 3-14

Loads an ELF image file.

Type `help` followed by a command name for more information on a specific command.

3.2.4 Controlling target execution and connections

List of commands:

break on page 3-7

Stops running the target.

go on page 3-16

Starts running the target.

system.down on page 3-20

Disconnects the debugger from the target.

system.up on page 3-21

Connects to the specified target.

Type `help` followed by a command name for more information on a specific command.

3.2.5 Displaying the call stack and associated variables

List of commands:

var.frame on page 3-22

Displays the stack frame.

Type `help` followed by a command name for more information on a specific command.

3.2.6 Controlling the debugger and program information

List of commands:

var.new on page 3-25

Creates a new script variable and zero-initializes it.

var.set on page 3-27

Sets and displays the value of an existing script variable.

Type `help` followed by a command name for more information on a specific command.

3.2.7 Supporting commands

List of commands:

help on page 3-17

Displays help information for a specific command or a group of commands listed according to specific debugging tasks.

wait on page 3-28

Pauses the execution of a script for a specified period of time.

Type `help` followed by a command name for more information on a specific command.

3.3 CMM-style commands listed in alphabetical order

The CMM-style commands in alphabetical order are:

- *break* on page 3-7
- *break.delete* on page 3-8
- *break.disable* on page 3-9
- *break.enable* on page 3-10
- *break.set* on page 3-11
- *data.dump* on page 3-12
- *data.load.binary* on page 3-13
- *data.load.elf* on page 3-14
- *data.set* on page 3-15
- *go* on page 3-16
- *help* on page 3-17
- *print* on page 3-18
- *register.set* on page 3-19
- *system.down* on page 3-20
- *system.up* on page 3-21
- *var.frame* on page 3-22
- *var.global* on page 3-23
- *var.local* on page 3-24
- *var.new* on page 3-25
- *var.print* on page 3-26
- *var.set* on page 3-27
- *wait* on page 3-28.

3.3.1 `break`

This command stops running the target.

Syntax

`break`

Example

Example 3-1 `break`

```
break                         ; Stop running the target
```

See also

- *go* on page 3-16
- *system.down* on page 3-20
- *system.up* on page 3-21.

3.3.2 `break.delete`

This command deletes a breakpoint at the specified address.

Syntax

`break.delete` *`expression`*

Where:

`expression`	Specifies the breakpoint address. This can be either an address, a symbol name, or an expression that evaluates to an address. You can use the syntax *`symbol\line`* to refer to a specific source line offset from a symbol.

Example

Example 3-2 `break.delete`

```
break.delete 0x8000  ; Delete breakpoint at address 0x8000
break.delete main    ; Delete breakpoint at address of main()
break.delete main+4  ; Delete breakpoint 4 bytes after address of main()
break.delete main\2  ; Delete breakpoint 2 source lines after address of main()
```

See also

- *break.disable* on page 3-9
- *break.enable* on page 3-10
- *break.set* on page 3-11.

3.3.3 `break.disable`

This command disables a breakpoint at the specified address.

Syntax

`break.disable` *`expression`*

Where:

`expression`	Specifies the breakpoint address. This can be either an address, a symbol name, or an expression that evaluates to an address. You can use the syntax *`symbol\line`* to refer to a specific source line offset from a symbol.

Example

Example 3-3 `break.disable`

```
break.disable 0x8000 ; Disable breakpoint at address 0x8000
break.disable main   ; Disable breakpoint at address of main()
break.disable main+4 ; Disable breakpoint 4 bytes after address of main()
break.disable main\2 ; Disable breakpoint 2 source lines after address of main()
```

See also

- *break.delete* on page 3-8
- *break.enable* on page 3-10
- *break.set* on page 3-11.

3.3.4 `break.enable`

This command enables a breakpoint at the specified address.

Syntax

`break.enable` *`expression`*

Where:

`expression`	Specifies the breakpoint address. This can be either an address, a symbol name, or an expression that evaluates to an address. You can use the syntax *`symbol\line`* to refer to a specific source line offset from a symbol.

Example

Example 3-4 `break.enable`

```
break.enable 0x8000 ; Enable breakpoint at address 0x8000
break.enable main   ; Enable breakpoint at address of main()
break.enable main+4 ; Enable breakpoint 4 bytes after address of main()
break.enable main\2 ; Enable breakpoint 2 source lines after address of main()
```

See also

- *break.delete* on page 3-8
- *break.disable* on page 3-9
- *break.set* on page 3-11.

3.3.5 break.set

This command sets a software breakpoint at the specified address.

Syntax

`break.set expression [/flag]`

Where:

expression	Specifies the breakpoint address. This can be either an address, a symbol name, or an expression that evaluates to an address. You can use the syntax `symbol\line` to refer to a specific source line offset from a symbol.
flag	Specifies an additional flag: `disable` Disables the breakpoint immediately after setting it.

Example

Example 3-5 `break.set`

```
break.set 0x8000      ; Set breakpoint at address 0x8000
break.set main        ; Set breakpoint at address of main()
break.set main+4      ; Set breakpoint 4 words after address of main()
break.set main\2      ; Set breakpoint 2 source lines after address of main()
```

See also

- *break.delete* on page 3-8
- *break.disable* on page 3-9
- *break.enable* on page 3-10.

3.3.6 data.dump

This command displays data at a specific address or address range. By default, the display size is `0x20` bytes of data unless an address range is specified.

Syntax

`data.dump` *`expression`* `[/`*`flag`*`]…`

Where:

expression	Specifies the address or address range. This can be either an address, an address range, or an expression that evaluates to an address. You can use -- to specify an address range and ++ to specify an offset from an address.
flag	Specifies additional flags:

`byte`	Formats the data as 1 byte
`word`	Formats the data as 2 bytes
`long`	Formats the data as 4 bytes
`quad`	Formats the data as 8 bytes
`width`	Specifies the number of columns
`nohex`	Suppresses the hexadecimal output
`noascii`	Suppresses the ASCII output
`le`	Formats the data as little endian
`be`	Formats the data big endian.

If no endianness is specified then the debugger looks for information at the start address of the loaded image otherwise little endian is used.

Example

Example 3-6 `data.dump`

```
data.dump 0x8000            ; Display 0x20 bytes (default) from address 0x8000
data.dump 0x8000--0x8170    ; Display data in address range 0x8000--0x8170
data.dump r(PC)++0x100      ; Display 0x100 bytes from address in PC register
```

3.3.7 data.load.binary

This command loads a binary image file.

Note

Loading a binary image does not change the program counter or any symbols that are currently loaded.

Syntax

`data.load.binary` *`filename expression`*

Where:

filename	Specifies the image file.
expression	Specifies the load address. This can be either an address, a symbol name, or an expression that evaluates to an address. If none specified then the default is `0x0`.

Example

Example 3-7 `data.load.binary`

```
data.load.binary "myFile.bin"                   ; Load image at address 0x0
data.load.binary "../my directory/myFile.bin"    ; Load image at address 0x0
data.load.binary "myFile.bin" 0x8000            ; Load image at address 0x8000
```

See also

- *data.load.elf* on page 3-14.

3.3.8 `data.load.elf`

This command loads an ARM *Executable and Linking Format* (ELF) file. This format is described in the ARM ELF specification and uses the `.axf` file extension.

Note

Loading an ELF image sets the program counter to the entry point of the image, if present.

Syntax

`data.load.elf` *`filename`* `[/`*`flag`*`]…`

Where:

`filename`	Specifies the image file.
`flag`	Specifies additional flags:

`nocode`	Do not load code and data to the target.
`nosymbol`	Do not load symbols.
`noclear`	Symbol table is not cleared before loading the image.
`noreg`	Do not set register values, for example, PC and status registers.

Default

By default, this command loads code and data to the target, clears the existing symbol table before loading the new symbols into the symbol table, and sets the registers.

You must use additional flags if you want to modify the default options. For example, you must use `/noclear` if you want to load the symbols from multiple images.

Example

Example 3-8 `data.load.elf`

```
data.load.elf "myFile.axf"                ; Load image and symbols
data.load.elf "../my directory/myFile.axf" ; Load image and symbols
data.load.elf "myFile.axf" /nosymbol      ; Load image without symbols
```

See also

- *data.load.binary* on page 3-13.

3.3.9 data.set

This command writes data to memory.

Syntax

`data.set address [%format] expression [/flag]…`

Where:

address	Specifies the address or address range. This can be either an address, an address range, or an expression that evaluates to an address. You can use -- to specify an address range.
format	Specifies additional formatting:

`byte`	Formats the data as 1 byte
`word`	Formats the data as 2 bytes
`long`	Formats the data as 4 bytes
`quad`	Formats the data as 8 bytes
`float.ieee`	Formats the data as a 4 byte floating-point.
`float.ieeedbl`	Formats the data as an 8 byte floating-point.
`le`	Formats the data as little endian
`be`	Formats the data big endian.

If no endianness is specified then the debugger searches for this information in the loaded image otherwise little endian is used.

expression	Specifies the data.
flag	Specifies additional flags:

`verify`	Verifies the write operation.
`compare`	Compares the data in memory but does not write to memory.

Example

Example 3-9 `data.set`

```
data.set r(PC) 0x10                     ; Write 0x10 to address in PC register
data.set 0x100--0x3ff 0x0               ; Zero initialize memory
data.set 0x8000--0x100 %w 0x2000 /compare ; Compare data in memory with 0x2000
data.set 0x100--0x3ff 0x0 /verify       ; Zero initialize memory and verify
```

3.3.10 go

This command starts running the device.

Syntax

`go`

Example

Example 3-10 go

```
go                                ; Start running the device
```

See also

- *break* on page 3-7
- *system.down* on page 3-20
- *system.up* on page 3-21.

3.3.11 `help`

This command displays help information for a specific command or a group of commands listed according to specific debugging tasks.

Syntax

`help [command|group]`

Where:

`command`	Specifies an individual command.
`group`	Specifies a group name for specific debugging tasks:

`all`	Displays all the commands.
`breakpoints`	Controlling breakpoints.
`data`	Controlling data and display settings.
`files`	Controlling images, symbols and libraries.
`running`	Controlling target execution and stepping.
`stack`	Displaying the call stack and associated variables.
`status`	Controlling the default settings and program status information.
`support`	Additional supporting commands.

Example

Example 3-11 `help`

```
help var.frame           # Display help information for var.frame command
help print               # Display help information for print command
help breakpoints         # Display group of breakpoint commands
help status              # Display group of status commands
```

3.3.12 `print`

This command concatenates the results of one or more expressions.

Syntax

`print [%printing_format] expression…`

Where:

printing_format	Specifies either [ascii | binary | decimal | hex]. If none specified then the default is decimal format.
expression	Specifies an expression that is evaluated and the result is returned.

Example

Example 3-12 `print`

```
print %h r(R0)              ; Display R0 register in hexadecimal
print %d r(PC)              ; Display PC register in decimal
print 4+4                   ; Display result of expression in decimal
print "Result is " 4+4      ; Display string and result of expression
print "Value is: " myVar    ; Display string and variable value
print v.value(myVar)        ; Display variable value
```

3.3.13 register.set

This command sets the value of a register.

Syntax

`register.set name expression`

Where:

name	Specifies the name of a register.
expression	Specifies an expression that is evaluated and the result assigned to a register.

Example

Example 3-13 register.set

```
register.set R0 15          ; Set value of R0 register to 15
register.set R0 (10*10)     ; Set value of R0 register to result of expression
register.set R0 r(R0)+1     ; Increment the value of R0 register
register.set PC main        ; Set value of PC register to address of main()
```

3.3.14 system.down

This command disconnects the debugger from the target.

Syntax

`system.down`

Example

Example 3-14 `system.down`

```
system.down                        ; Disconnect from target
```

See also

- *break* on page 3-7
- *go* on page 3-16
- *system.up* on page 3-21.

3.3.15 `system.up`

This command connects to the specified target.

Syntax

`system.up`

Example

Example 3-15 `system.up`

```
system.up                          ; Connect to target
```

See also

- *break* on page 3-7
- *go* on page 3-16
- *system.down* on page 3-20.

3.3.16 `var.frame`

This command displays the stack frame.

Syntax

`var.frame [%printing_format] [/flag]…`

Where:

printing_format	Specifies either [`ascii` | `binary` | `decimal` | `hex`]. If none specified then the default is decimal format.
flag	Specifies additional flags:

`novar`	Disables the display of variables.
`nocaller`	Disables the display of function callers. This is the default.
`args`	Displays arguments. This is the default.
`locals`	Displays local variables.
`caller`	Displays function callers.
`json`	Specifies an output option to display messages in JSON format.

Example

Example 3-16 `var.frame`

```
var.frame /locals /caller       ; Display variables and function callers
var.frame %hex /locals /caller  ; Display variables and callers in hexadecimal
var.frame /novar                ; Do not display any variables
var.frame /json                 ; Display stack frame in JSON format
```

3.3.17 `var.global`

This command displays all global variables.

Syntax

`var.global [%printing_format] [/flag]`

Where:

printing_format	Specifies either [`ascii` | `binary` | `decimal` | `hex`]. If none specified then the default is decimal format.
flag	Specifies an additional flag: `json` Specifies an output option to display messages in JSON format.

Example

Example 3-17 `var.global`

```
var.global                  ; Display all global variables
var.global %h               ; Display all global variables in hexadecimal
```

See also

- *var.local* on page 3-24
- *var.print* on page 3-26.

3.3.18 var.local

This command displays all local variables in a function.

Syntax

`var.local [%printing_format] [/flag]`

Where:

printing_format	Specifies either [`ascii` | `binary` | `decimal` | `hex`]. If none specified then the default is decimal format.
flag	Specifies an additional flag: `json` Specifies an output option to display messages in JSON format.

Example

Example 3-18 `var.local`

```
var.local                   ; Display all local variables
var.local %h                ; Display all local variables in hexadecimal
```

See also

- *var.global* on page 3-23
- *var.print* on page 3-26.

3.3.19 `var.new`

This command creates a new script variable and zero-initializes it. Script variables are for use at runtime only.

Syntax

`var.new \name`

Where:

name	Specifies the name of a script variable.

Example

Example 3-19 `var.new`

```
var.new \myVar                        ; Create new script variable
```

See also

- *var.set* on page 3-27.

3.3.20 var.print

This command concatenates the results of one or more expressions.

Syntax

`var.print [%printing_format] expression… [/flag]`

Where:

printing_format	Specifies either [`ascii` | `binary` | `decimal` | `hex`]. If none specified then the default is decimal format.
expression	Specifies an expression that is evaluated and the result is returned. You can use script variables in an expression by preceding the name with a backslash. Script variables are for use at runtime only.
flag	Specifies an additional flag: `json` Specifies an output option to display messages in JSON format.

Example

Example 3-20 `var.print`

```
var.print "Value is: " myVar1       ; Display string and myVar1
var.print myVar1 " and " myVar2     ; Display concatenated string/variables
var.print %h myVar1                 ; Display myVar1 in hexadecimal
var.print \myVar                    ; Display value of script variable
```

3.3.21 var.set

This command sets and displays the value of an existing script variable. It can also display the result of an expression. Script variables are for use at runtime only.

Syntax

`var.set [\name=]expression`

Where:

name Specifies the name of an existing script variable.

——— **Note** ———

If you specify the name of an existing script variable then you must use this command after the `var.new` command.

———————————

expression Specifies an expression that is evaluated and the result is returned. If you specify an expression with the *name* option then the value of that script variable is also updated with the result of the expression.

Example

Example 3-21 `var.set`

```
var.set \myVar              ; Display value of script variable
var.set \myVar=3+3          ; Set value of script variable and display result
var.set 3+3                 ; Display result
```

See also

- *var.new* on page 3-25
- *var.print* on page 3-26.

3.3.22 `wait`

This command pauses the execution of a script for a specified period of time.

Syntax

`wait` *`number`*`{m|s}`

Where:

`number`	Specifies the period of time.
`m`	Specifies the time in milliseconds.
`s`	Specifies the time in seconds.

Example

Example 3-22 `wait`

```
wait 1s                   ; Wait one second
wait 0.5s                 ; Wait half a second
wait 1000m                ; Wait one thousand milliseconds
```

Appendix A
GNU Free Documentation License

1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
15. Preserve any Warranty Disclaimers.

of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

9. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See *http://www.gnu.org/copyleft/*.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
   Copyright (C)  YEAR  YOUR NAME.   Permission is granted to copy, distribute and/or
modify this document   under the terms of the GNU Free Documentation License, Version
1.2   or any later version published by the Free Software Foundation;   with no
Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.   A copy of the
license is included in the section entitled "GNU   Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
   with the Invariant Sections being list their titles, with the   Front-Cover Texts
being list, and with the Back-Cover Texts being list.
```

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.