



Arm® Debugger

Version 6.5.0

Command Reference

Non-Confidential

Copyright © 2018–2025 Arm Limited (or its affiliates).
All rights reserved.

Issue 00

101471_6.5.0_00_en



Arm® Debugger Command Reference

This document is Non-Confidential.

Copyright © 2018–2025 Arm Limited (or its affiliates). All rights reserved.

This document is protected by copyright and other intellectual property rights.

Arm only permits use of this document if you have reviewed and accepted [Arm's Proprietary Notice](#) found at the end of this document.

This document (101471_6.5.0_00_en) was issued on 2025-03-13. There might be a later issue at <https://developer.arm.com/documentation/101471>

The product version is 6.5.0.

See also: [Proprietary notice](#) | [Product and document information](#) | [Useful resources](#)

Start reading

If you prefer, you can skip to [the start of the content](#).

Intended audience

Arm® Debugger supports software development on Arm processor-based targets and Fixed Virtual Platform (FVP) targets. This document includes a full list of Arm Debugger commands with usage instructions and examples.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>.

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

Contents

1. Arm Debugger overview.....	12
2. Arm Debugger commands.....	13
2.1 Conformance and usage rules for Arm Debugger commands.....	13
2.1.1 Syntax of Arm Debugger commands.....	13
2.1.2 Special characters and environment variables in paths in Arm Debugger.....	14
2.1.3 Expressions in the Arm Debugger.....	15
2.1.4 printf() style format string in Arm Debugger.....	22
2.1.5 Address space prefixes.....	24
2.1.6 Memory parameters.....	26
2.2 Arm Debugger commands listed in groups.....	28
2.2.1 Breakpoints and watchpoints.....	29
2.2.2 Execution control.....	31
2.2.3 Tracing.....	33
2.2.4 Scripts.....	33
2.2.5 Call stack.....	34
2.2.6 Operating System.....	35
2.2.7 Files.....	36
2.2.8 Data.....	38
2.2.9 Memory group.....	39
2.2.10 Cache.....	40
2.2.11 Registers.....	40
2.2.12 peripheral.....	40
2.2.13 mmu.....	41
2.2.14 mmu list.....	42
2.2.15 mpu.....	42
2.2.16 mpu list.....	42
2.2.17 Display.....	43
2.2.18 Information.....	43
2.2.19 log.....	45
2.2.20 Set.....	46
2.2.21 set elf.....	48

2.2.22 show.....	48
2.2.23 show elf.....	50
2.2.24 flash.....	51
2.2.25 Support.....	51
2.3 Arm Debugger commands listed in alphabetical order.....	53
2.3.1 add-symbol-file.....	61
2.3.2 advance.....	62
2.3.3 append.....	64
2.3.4 assemble.....	65
2.3.5 awatch.....	67
2.3.6 backtrace.....	70
2.3.7 break.....	70
2.3.8 break-script.....	72
2.3.9 break-set-property.....	73
2.3.10 break-stop-on-cores.....	74
2.3.11 break-stop-on-threads, break-stop-on-cores.....	74
2.3.12 break-stop-on-vmid.....	75
2.3.13 cache flush.....	75
2.3.14 cache list.....	76
2.3.15 cache print.....	77
2.3.16 cd.....	78
2.3.17 clear.....	78
2.3.18 clearwatch.....	79
2.3.19 condition.....	80
2.3.20 continue.....	81
2.3.21 core apply.....	81
2.3.22 core.....	82
2.3.23 define.....	83
2.3.24 delete breakpoints.....	84
2.3.25 delete memory.....	85
2.3.26 directory, set directories.....	85
2.3.27 disable breakpoints.....	86
2.3.28 disable memory.....	87
2.3.29 disassemble.....	88
2.3.30 discard-symbol-file.....	88
2.3.31 document.....	89

2.3.32 down.....	90
2.3.33 down-silently.....	90
2.3.34 dump.....	91
2.3.35 echo.....	92
2.3.36 enable breakpoints.....	93
2.3.37 enable memory.....	94
2.3.38 end.....	94
2.3.39 exit.....	95
2.3.40 file, symbol-file.....	95
2.3.41 finish.....	96
2.3.42 flash erase-device.....	96
2.3.43 flash erase-image-sectors.....	97
2.3.44 flash load.....	98
2.3.45 flash load-multiple.....	99
2.3.46 frame.....	101
2.3.47 gcs print, info gcs.....	101
2.3.48 handle.....	103
2.3.49 hbreak.....	104
2.3.50 help.....	106
2.3.51 if.....	108
2.3.52 ignore.....	108
2.3.53 info address.....	109
2.3.54 info all-registers.....	110
2.3.55 info breakpoints, info watchpoints.....	111
2.3.56 info breakpoints capabilities.....	112
2.3.57 info capabilities.....	112
2.3.58 info classes.....	113
2.3.59 info cores.....	114
2.3.60 info files.....	114
2.3.61 info flash.....	115
2.3.62 info frame.....	115
2.3.63 info functions.....	116
2.3.64 info gcs.....	117
2.3.65 info handle, info signals.....	117
2.3.66 info inst-sets.....	118
2.3.67 info locals.....	118

2.3.68 info members.....	119
2.3.69 info memory.....	119
2.3.70 info memory-parameters.....	122
2.3.71 info os.....	125
2.3.72 info os-log.....	125
2.3.73 info os-modules.....	126
2.3.74 info os-version.....	127
2.3.75 info overlays.....	127
2.3.76 info processes.....	128
2.3.77 info registers.....	128
2.3.78 info semihosting.....	129
2.3.79 info sharedlibrary.....	131
2.3.80 info signals.....	132
2.3.81 info sources.....	132
2.3.82 info stack, backtrace, where.....	133
2.3.83 info symbol.....	134
2.3.84 info target.....	134
2.3.85 info threads.....	135
2.3.86 info variables.....	135
2.3.87 info watchpoints.....	136
2.3.88 info watchpoints capabilities.....	136
2.3.89 inspect.....	137
2.3.90 interrupt, stop.....	137
2.3.91 list.....	137
2.3.92 load.....	139
2.3.93 loadfile.....	140
2.3.94 log config.....	140
2.3.95 log file.....	141
2.3.96 memory.....	142
2.3.97 memory auto.....	144
2.3.98 memory debug-cache.....	145
2.3.99 memory fill.....	146
2.3.100 memory set.....	147
2.3.101 memory set_typed.....	149
2.3.102 mmu dirty-state.....	150
2.3.103 mmu list dirty-state.....	153

2.3.104 mmu list memory-maps, mpu list memory-maps.....	153
2.3.105 mmu list tables, mpu list tables.....	154
2.3.106 mmu list translations.....	155
2.3.107 mmu memory-map, mpu memory-map.....	155
2.3.108 mmu print, mpu print.....	156
2.3.109 mmu translate.....	157
2.3.110 newvar.....	158
2.3.111 next.....	160
2.3.112 nexti.....	161
2.3.113 nexts.....	162
2.3.114 nosharedlibrary.....	162
2.3.115 output.....	163
2.3.116 pause.....	164
2.3.117 peripheral add.....	165
2.3.118 peripheral delete.....	166
2.3.119 peripheral show.....	166
2.3.120 preprocess.....	167
2.3.121 print, inspect.....	168
2.3.122 pwd.....	170
2.3.123 quit, exit.....	170
2.3.124 reload-symbol-file.....	171
2.3.125 reset.....	171
2.3.126 resolve.....	172
2.3.127 restore.....	173
2.3.128 run.....	174
2.3.129 rwatch.....	175
2.3.130 select-frame.....	178
2.3.131 set arm.....	178
2.3.132 set auto-solib-add.....	179
2.3.133 set backtrace.....	180
2.3.134 set blocking-run-control.....	181
2.3.135 set breakpoint.....	181
2.3.136 set case-insensitive-source-matching.....	183
2.3.137 set cde-coprocessors.....	183
2.3.138 set debug-agent.....	184
2.3.139 set debug-from.....	185

2.3.140 set directories.....	185
2.3.141 set dtls-options.....	186
2.3.142 set dtls-temporary-directory.....	186
2.3.143 set elf cache-uninitialized-sections.....	187
2.3.144 set elf load-segments-at-p_paddr.....	188
2.3.145 set elf zero-extra-segment-bytes.....	189
2.3.146 set endian.....	189
2.3.147 set escape-strings.....	190
2.3.148 set escapes-in-filenames.....	191
2.3.149 set idau-region.....	191
2.3.150 set listsize.....	192
2.3.151 set mmu use-cache-for-phys-reads.....	193
2.3.152 set os.....	193
2.3.153 set overlays enabled.....	195
2.3.154 set print.....	195
2.3.155 set semihosting.....	197
2.3.156 set solib-absolute-prefix.....	200
2.3.157 set solib-search-path.....	200
2.3.158 set step-mode.....	201
2.3.159 set stop-on-solib-events.....	201
2.3.160 set substitute-path.....	202
2.3.161 set sysroot, set solib-absolute-prefix.....	202
2.3.162 set trust-ro-sections-for-opcodes.....	203
2.3.163 set variable, set.....	204
2.3.164 set wildcard-style.....	205
2.3.165 sharedlibrary.....	205
2.3.166 shell.....	206
2.3.167 show.....	207
2.3.168 show architecture.....	207
2.3.169 show arm.....	207
2.3.170 show auto-solib-add.....	208
2.3.171 show backtrace.....	209
2.3.172 show blocking-run-control.....	209
2.3.173 show breakpoint.....	210
2.3.174 show case-insensitive-source-matching.....	210
2.3.175 show cde-coprocessors.....	211

2.3.176 show debug-agent.....	211
2.3.177 show debug-from.....	212
2.3.178 show directories.....	212
2.3.179 show dtsl-options.....	213
2.3.180 show dtsl-temporary-directory.....	213
2.3.181 show elf cache-uninitialized-sections.....	213
2.3.182 show elf load-segments-at-p_paddr.....	214
2.3.183 show elf zero-extra-segment-bytes.....	214
2.3.184 show endian.....	215
2.3.185 show escape-strings.....	215
2.3.186 show escapes-in-filenames.....	216
2.3.187 show idau-region.....	216
2.3.188 show listsize.....	216
2.3.189 show mmu use-cache-for-phys-reads.....	217
2.3.190 show os.....	217
2.3.191 show print.....	218
2.3.192 show semihosting.....	219
2.3.193 show solib-absolute-prefix.....	220
2.3.194 show solib-search-path.....	221
2.3.195 show step-mode.....	221
2.3.196 show stop-on-solib-events.....	222
2.3.197 show substitute-path.....	222
2.3.198 show sysroot, show solib-absolute-prefix.....	223
2.3.199 show trust-ro-sections-for-opcodes.....	223
2.3.200 show version.....	224
2.3.201 show wildcard-style.....	224
2.3.202 silence.....	225
2.3.203 snapshot.....	225
2.3.204 source.....	227
2.3.205 start.....	228
2.3.206 stdin.....	229
2.3.207 step.....	229
2.3.208 stepi.....	230
2.3.209 steps.....	231
2.3.210 stop.....	232
2.3.211 symbol-file.....	232

2.3.212 tbreak.....	232
2.3.213 thbreak.....	234
2.3.214 thread.....	236
2.3.215 thread apply.....	237
2.3.216 trace clear.....	237
2.3.217 trace dump.....	238
2.3.218 trace info.....	240
2.3.219 trace list.....	240
2.3.220 trace report.....	241
2.3.221 trace start.....	244
2.3.222 trace stop.....	245
2.3.223 unset.....	245
2.3.224 unsilence.....	246
2.3.225 up.....	247
2.3.226 up-silently.....	247
2.3.227 usecase help.....	248
2.3.228 usecase list.....	249
2.3.229 usecase run.....	250
2.3.230 wait.....	251
2.3.231 watch.....	252
2.3.232 watch-set-property.....	254
2.3.233 whatis.....	255
2.3.234 where.....	255
2.3.235 while.....	256
2.3.236 x.....	257

3. CMM-style commands supported by the debugger.....	260
3.1 CMM-style commands groups: All.....	261
3.1.1 Controlling breakpoints.....	261
3.1.2 Controlling data and display settings.....	262
3.1.3 Controlling images, symbols, and libraries.....	262
3.1.4 Controlling target execution and connections.....	263
3.1.5 Displaying the call stack and associated variables.....	263
3.1.6 Controlling the debugger and program information.....	263
3.1.7 Supporting commands.....	263
3.2 CMM-style commands listed in alphabetical order.....	264

3.2.1 CMM-style commands: break.....	265
3.2.2 CMM-style commands: break.delete.....	265
3.2.3 CMM-style commands: break.disable.....	266
3.2.4 CMM-style commands: break.enable.....	266
3.2.5 CMM-style commands: break.set.....	267
3.2.6 CMM-style commands: data.dump.....	267
3.2.7 CMM-style commands: data.load.binary.....	269
3.2.8 CMM-style commands: data.load.elf.....	269
3.2.9 CMM-style commands: data.set.....	270
3.2.10 CMM-style commands: go.....	272
3.2.11 CMM-style commands: help.....	272
3.2.12 CMM-style commands: print.....	273
3.2.13 CMM-style commands: register.set.....	274
3.2.14 CMM-style commands: system.down.....	274
3.2.15 CMM-style commands: system.up.....	275
3.2.16 CMM-style commands: var.frame.....	275
3.2.17 CMM-style commands: var.global.....	276
3.2.18 CMM-style commands: var.local.....	277
3.2.19 CMM-style commands: var.new.....	277
3.2.20 CMM-style commands: var.print.....	278
3.2.21 CMM-style commands: var.set.....	278
3.2.22 CMM-style commands: wait.....	279
4. GNU Free Documentation License Details.....	280
4.1 GNU Free Documentation License.....	280
4.2 ADDENDUM: How to use this License for your documents.....	286
Proprietary notice.....	287
Product and document information.....	289
Product status.....	289
Revision history.....	289
Conventions.....	291
Useful resources.....	293

1. Arm Debugger overview

Arm® Debugger is a tool that helps you find the causes of software bugs on Arm processor-based targets and Fixed Virtual Platform (FVP) targets. The Arm Debugger is integrated into the Arm Development Studio and Arm® Keil® MDK v6 products.

Versioning

Previous versions of Arm Debugger were available only in Arm Development Studio and this Debugger Command Reference document previously had the same version number as Arm Development Studio, for example, Arm Development Studio version 2023.1 has Arm Debugger version 2023.1.

The Arm Debugger is now available in Arm Keil MDK v6 and Arm Development Studio and has a new versioning scheme starting with version 6.1.0.

Additional Arm Debugger documentation

- For more details about using the Arm Debugger in Arm Development Studio, see [Arm Development Studio Getting Started Guide: Introduction to Arm Debugger](#).
- For more details about using the Arm Debugger in Arm Keil MDK v6, see [Arm Keil Studio Visual Studio Code Extensions User Guide: Arm Debugger extension](#).

2. Arm Debugger commands

Arm® Debugger commands are a comprehensive set of commands to debug embedded applications. This section provides an overview of the conformance and usage rules for Arm Debugger commands and describes how to use each of the commands.

2.1 Conformance and usage rules for Arm Debugger commands

This section provides topics that describe the conformance and usage rules for Arm® Debugger commands.

Related information

[Syntax of Arm Debugger commands](#) on page 13

[Special characters and environment variables in paths in Arm Debugger](#) on page 14

[Expressions in the Arm Debugger](#) on page 15

[printf\(\) style format string in Arm Debugger](#) on page 22

[Address space prefixes](#) on page 24

[Memory parameters](#) on page 26

2.1.1 Syntax of Arm Debugger commands

Arm® Debugger commands accept arguments and flags. A flag acts as an optional switch and is specified using a forward slash character. Where a command supports flags, the flags are described as part of the command syntax.

```
command [argument] [/<flag>]...
```



Note

- Commands are not case sensitive.
- When you specify an address as an argument to a command, you can also specify the [address space](#), for example `N:0x80000000`. If you do not specify the address space, Arm Debugger assumes the current address space.

In commands that use `/<flag>`, the position of `/<flag>` should generally be as shown in the command syntax. The commands you submit to the debugger must follow these 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 Arm Development Studio, you can open the

Development Studio perspective where you can use the menus, icons, and toolbars provided, or you can enter Arm Debugger commands in the **Commands** view.

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.

Some 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 providing an alias.

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.

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

2.1.2 Special characters and environment variables in paths in Arm Debugger

List of characters and variables that you can use for path shortcuts in Arm® Debugger commands.

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.



Note

Arm Development Studio does not resolve [Eclipse variables](#) when scripting or using the [Commands view](#).

However, they are resolved when you [export a debug configuration](#).

Related information

[set escapes-in-filenames](#) on page 191

2.1.3 Expressions in the Arm Debugger

Some Arm® Debugger commands accept expressions, which enable you to extend the operation of a command. Expressions can include binary mathematical expressions, references to module names, wildcards, or calls to functions.

Related information

[Accessing registers and debugger variables in Arm Debugger expressions](#) on page 15

[Accessing system registers by instruction encoding](#) on page 17

[Built-in functions in Arm Debugger expressions](#) on page 17

[Wildcards in Arm Debugger expressions](#) on page 19

[Globs in Arm Debugger expressions](#) on page 19

[Regular expressions in Arm Debugger](#) on page 19

[Regular expressions in the C expression parser in Arm Debugger](#) on page 20

[Scoping resolution operator in Arm Debugger](#) on page 21

2.1.3.1 Accessing registers and debugger variables in Arm Debugger expressions

In an Arm® Debugger expression you can access the content of registers and debugger variables by using the `$` character and the register name.

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 Symmetric MultiProcessing (SMP) systems.

\$vmid

Current Virtual Machine ID (VMID) for systems that support hypervisor / virtual machine debugging.

**Note**

- `$thread` is uniquely assigned by the debugger for the current context reported by the OS awareness plugin. If no OS awareness plugin is loaded, `$thread` tracks the current core, `$core`.
- `$pid` is assigned for the debugger for the current context by the OS awareness plugin. If no OS awareness plugin is loaded, `$pid` tracks the current core, `$core`.

Example: Register variable

The following command adds 4 to the content of R0 register and prints the result:

```
print 4+$R0
```

Related information

[Accessing system registers by instruction encoding](#) on page 17

[Built-in functions in Arm Debugger expressions](#) on page 17

[printf\(\) style format string in Arm Debugger](#) on page 22

[echo](#) on page 92

[set print](#) on page 195

[show print](#) on page 218

[append](#) on page 64

[break](#) on page 70

[thread](#) on page 235

[core](#) on page 82

[x](#) on page 256

[advance](#) on page 62

[About OS Awareness](#)

2.1.3.2 Accessing system registers by instruction encoding

You can access system registers by specifying the register instruction encoding. This feature enables access to system registers that are not explicitly supported in the debugger.



You must take care when specifying registers by instruction encoding, because the debugger does not check whether the register exists.

Example: Display register by instruction encoding

This command displays the contents of the register with an instruction encoding of `s3_1_c15_c2_0`:

```
output /x $S3_1_C15_C2_0
```

Example: Set register content by instruction encoding

This command sets the contents of the register with an instruction encoding of `s3_1_c15_c2_0`:

```
set $S3_1_C15_C2_0 = 0x80C2000
```

Related information

[Accessing registers and debugger variables in Arm Debugger expressions](#) on page 15

[newvar](#) on page 158

2.1.3.3 Built-in functions in Arm Debugger expressions

In an Arm® Debugger expression, you can use built-in functions to provide more functionality.

You can use the following built-in functions in Arm Debugger expressions:

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 strcmp(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: Conditional breakpoints

The following command sets a conditional breakpoint that stops when strings are identical.

```
break main.c:45 if strcmp(myVar, "10") == 0
```

Related information

[printf\(\) style format string in Arm Debugger](#) on page 22

[echo](#) on page 92

[set print](#) on page 195

[show print](#) on page 218

[append](#) on page 64

[break](#) on page 70

[thread](#) on page 235

[core](#) on page 82

[x](#) on page 256

2.1.3.4 Wildcards in Arm Debugger expressions

You can use wildcards to enhance your pattern matching in Arm® Debugger expressions.

You can use the following types of wildcard pattern matching:

- [Globs](#). This pattern is the default.
- [Regular expressions](#).

Use the Arm Debugger command `set wildcard-style` to change the default setting.

Related information

[set wildcard-style](#) on page 205

[show wildcard-style](#) on page 224

2.1.3.5 Globs in Arm Debugger expressions

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: Using globs

The following is a glob wildcard that lists all functions that start with m:

```
info functions m*
```

Related information

[set wildcard-style](#) on page 205

[show wildcard-style](#) on page 224

2.1.3.6 Regular expressions in Arm Debugger

Commands that support wildcards can use regular expressions.

The exact regular expression syntax supported is described in a book called *Mastering Regular Expressions*.

Example: Regular expressions

The following is a regular expression wildcard that lists all functions that start with m:

```
info functions m.*
```

Related information

[set wildcard-style](#) on page 205

[show wildcard-style](#) on page 224

Jeffrey E. F.Friedl, *Mastering Regular Expressions*. ISBN 0-596-52812-4

2.1.3.7 Regular expressions in the C expression parser in Arm Debugger

The C expression parser in Arm® 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:

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

```
expression =~ regular_expression
```

This example evaluates to 0 if the regular expression that uses `!~` 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).

Related information

[Jeffrey E. F. Friedl, Mastering Regular Expressions. ISBN 0-596-52812-4](#)

2.1.3.8 Scoping resolution operator in Arm Debugger

In Arm® Debugger, 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. You can use this to reference variables and functions in images, files, namespaces, or classes.

The following example references `image.axf` is created using `demo.c`:

```
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
"image.axf":main
"image.axf":demo.c::FILE_STATIC_VARIABLE
"demo.c":FILE_STATIC_VARIABLE
NAME_SPACE_OUTER::TEST_VAR
NAME_SPACE_OUTER::NAME_SPACE_INNER::TEST_VAR

```

If you set 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->valueptr_oc->value

```

2.1.4 printf() style format string in Arm Debugger

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. This command 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, OF 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 OF u

Results in a decimal integer. d indicates a signed integer. u indicates an unsigned integer.

h OF H

Results in a Hexadecimal character in lower or upper case.

x OF X

Results in an unsigned Hexadecimal character in lower or upper case.

o

Results in an octal integer.

c OF C

Results in a Unicode character in lower or upper case.

s

Results in a string.

b OF 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 2016:17:00"

Related information

[Expressions in the Arm Debugger](#) on page 15

[echo](#) on page 92

[set print](#) on page 195

[show print](#) on page 218

[append](#) on page 64

[break](#) on page 70

[thread](#) on page 235

[core](#) on page 82

[x](#) on page 256

2.1.5 Address space prefixes

Use address space prefixes in Arm® Debugger to refer to different address spaces. You can use these address space prefixes for various debugging activities.

Default

If no address space prefix is specified, then the debugger defaults to the current address space.

Syntax

```
{address_space_prefix}[parameter=<value>,parameter=<value>,...]:{address}
```

Parameters**address_space_prefix**

The address space prefix. Address spaces can vary on different targets. The availability of an address space depends on what architecture features are implemented, such as security extensions.

The following address space prefixes might be available for Armv7-based processors:

- **s**: This corresponds to the Secure address space.
- **H**: This corresponds to the hypervisor address space.
- **N**: This corresponds to the Non-secure address space.
- **SP**: This corresponds to Secure World physical memory.
- **NP**: This corresponds to Non-secure World physical memory.

The following address space prefixes might be available for Armv8-based processors when in the AArch32 execution state:

- **s**: This corresponds to the EL3, Secure EL1, and Secure EL0 translation regimes.
- **h**: This corresponds to the EL2 translation regime. This is a Non-secure address space.
- **n**: This corresponds to the Non-secure EL1 and Non-secure EL0 translation regimes.
- **sp**: This corresponds to Secure World physical memory.
- **np**: This corresponds to Non-secure World physical memory.

The following address space prefixes might be available for Armv8-based processors when in the AArch64 execution state:

- **EL3**: This corresponds to the EL3 translation regime. This is a secure address space.
- **EL2S**: This corresponds to the Secure EL2 translation regime.
- **EL2N**: This corresponds to the Non-secure EL2 translation regime.
- **EL1S**: This corresponds to the Secure EL1 and Secure EL0 translation regimes.
- **EL1N**: This corresponds to the Non-secure EL1 and Non-secure EL0 translation regimes.
- **sp**: This corresponds to Secure World physical memory.
- **np**: This corresponds to Non-secure World physical memory.

In addition to the prefixes listed for Armv8-based processors, the following address space prefixes might be available or different for Armv9-A-based processors that support the *Realm Management Extension* (RME). They all apply to the AArch64 execution state, unless otherwise specified.

- **EL3**: This corresponds to the EL3 translation regime. This is a root address space.
- **EL2RL**: This corresponds to the Realm EL2 translation regime.
- **EL1RL**: This corresponds to the Realm EL1 and EL0 translation regime.
- **RL**: This corresponds to the AArch32 Realm. It is similar to **n**: and **s**:.
- **RLP**: This corresponds to Realm physical memory.
- **RTP**: This corresponds to Root physical memory.

parameter

Optional. The parameter you want to specify.

When you are using an address space as part of an expression, you can use memory parameters to specify additional behavior. Use the [info memory-parameters](#) command to see the available parameters.

<value>

The value that you want to set for the parameter.

address

There address where you want to apply the operation.

Example: break command with address space prefix for Armv7

This example sets an execution breakpoint in the main function in the secure address space.

```
break S:main
```

Example: add-symbol-file command with address space prefix for Armv8

This example loads additional debug information into the secure physical address space.

```
add-symbol-file foo.axf SP:0
```

Example: x command with address space prefix for Armv8

This example displays the content of the memory at address 0x80000000 in the secure EL1 and ELO translation regimes.

```
x EL1S:0x80000000
```

Example: Address space parameters with the set command

```
set* ((int*) SP<verify=0>:0x8000)=0x1234
```

This command writes an integer, 0x1234, to the secure physical address, 0x8000, but does not verify the write.

Related information

[info memory-parameters](#) on page 121

[break](#) on page 70

[add-symbol-file](#) on page 61

[x](#) on page 256

[Set](#) on page 46

[About address spaces](#)

2.1.6 Memory parameters

When you are using an address space as part of an expression, you can use memory parameters to specify additional behavior. There are many commands where you can apply memory parameters.

Different address spaces support different parameters. Use the [info memory-parameters](#) command to see which parameters apply to an address space.



Note

Sometimes the `info memory-parameters` command returns parameters that are not implemented by your processor. See the documentation for your processor to find out which parameters it supports.

Syntax

```
{command} {address_space_prefix}<memory_parameter1=<value>,  
memory_parameter2=<value>>:{address}
```

Parameters

memory_parameter<n>=<value>

Can be either a single parameter pair or a comma-separated list of parameter pairs:

verify=<value>

When performing a write operation, the debugger must read back what was written and verify that it was written correctly.

Possible values are:

0 - Do not verify.

1 - Verify. This value is the default.

width=<value>

Specifies the width for the access.

Where <value> is one of 8, 16, 32, 40, or 64. If you do not specify a value for width, Arm® Debugger sets the value to 0, which enables the debugger to determine the access width.

use_image=<value>

When fetching data, specify from where the debugger reads data.

Possible values are:

0 - Read data from the target.

1 - Read data from the loaded image.

view=<option>

View data for a feature that is associated with the address space.

Where <option> is:

MemTag, which accesses the Allocation Tags associated with the address space.



Note

- Allocation Tags are only accessible if the core implements the *Memory Tagging Extension* (MTE) feature, FEAT_MTE, and memory tags are enabled for the address space.
- This parameter is only available for the AArch64 address spaces.

stages=<value>

Specify the *Memory Management Unit* (MMU) translation stage to disable for a physical address.

The only <value> available for use is 1, which disables stage 1 and treats the address as an *Intermediate Physical Address* (IPA).

Example: Applying view=MemTag to the print command

```
print /x *(EL3:0xe000f12f)           # Print the data in hexadecimal at the
                                     # address in the address space.
print /x *(EL3<view=MemTag>:0xe000f12f) # Print, in hexadecimal, the Allocation
                                     # Tag associated with the address in
                                     # the address space.
```

Example: Disabling verify to the memory set_typed command

Write a 128-bit unsigned integer to the specified address, but do not check that the integer is written correctly:

```
memory set_typed N<verify=0>:0x00000000E000F110 (unsigned __int128)
(0x05050505050505050505050505050505)
```

Related information

[info memory-parameters](#) on page 121

[Address space prefixes](#) on page 24

2.2 Arm Debugger commands listed in groups

Displays all the commands in functional groups according to specific tasks.

Related information

[Breakpoints and watchpoints](#) on page 29

[Execution control](#) on page 31

[Tracing](#) on page 33

[Scripts](#) on page 33

[Call stack](#) on page 34

[Operating System](#) on page 35

[Files](#) on page 36

[Data](#) on page 37

[Memory group](#) on page 39

[Cache](#) on page 40

[Registers](#) on page 40

[peripheral](#) on page 40

[mmu](#) on page 41

[mmu list](#) on page 41

[mpu](#) on page 42
[mpu list](#) on page 42
[Display](#) on page 43
[Information](#) on page 43
[log](#) on page 45
[Set](#) on page 46
[set elf](#) on page 48
[show](#) on page 48
[show elf](#) on page 50
[flash](#) on page 51
[Support](#) on page 51

2.2.1 Breakpoints and watchpoints

List of all the Arm® Debugger commands that enable you to control the starting and stopping of the debugger using breakpoints and watchpoints.

awatch

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

break

Sets an execution breakpoint at a specific location.

break-script

Assigns a script file to a specific breakpoint. The script executes when the breakpoint is triggered.

break-set-property

Updates the properties of an existing breakpoint.

break-stop-on-threads, break-stop-on-cores

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

break-stop-on-vmid

Applies an existing hardware breakpoint to a *Virtual Machine* (VM).

clear

Deletes a breakpoint at a specific location.

clearwatch

Deletes a watchpoint at a specific location.

condition

Sets a stop condition for a specific breakpoint or watchpoint.

delete breakpoints

Deletes one or more breakpoints or watchpoints.

disable breakpoints

Disables one or more breakpoints or watchpoints.

enable breakpoints

Enables one or more breakpoints or watchpoints by number.

hbreak

Sets a hardware execution breakpoint at a specific location.

ignore

Sets the ignore counter for a breakpoint or watchpoint condition.

info breakpoints, info watchpoints

Displays information about the status of all breakpoints and watchpoints.

info breakpoints capabilities

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

info watchpoints capabilities

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

resolve

Re-evaluates the specified breakpoints or watchpoints and those with addresses that can be resolved are set.

rwatch

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

set breakpoint

Controls the automatic behavior of breakpoints and watchpoints.

show breakpoint

Displays the breakpoint and watchpoint behavior settings.

silence

Disables the printing of stop messages for a specific breakpoint.

tbreak

Sets an execution breakpoint at a specific location and deletes the breakpoint when it is hit.

thbreak

Sets a hardware execution breakpoint at a specific location and deletes the breakpoint when it is hit.

unsilence

Enables the printing of stop messages for a specific breakpoint.

watch

Sets a watchpoint for a data symbol. The debugger stops the target when the memory at the specified address is written.

watch-set-property

Updates the properties of an existing watchpoint.

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

Related information

[info capabilities](#) on page 112

2.2.2 Execution control

List of all the Arm® Debugger commands that enable you to control the starting and stopping of the debugger.

advance

Sets a temporary breakpoint at the specified address and calls the debugger `continue` command. Use the `advance` command to halt execution at a particular point in your code, for example a specific function, source code line number, or instruction memory address.

continue

Continues running the target.

core apply

Switches control to a specific processor to execute a debugger command and then switches back to the original state.

core

Displays information about the current processor.

finish

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

handle

Controls the handler settings for one or more signals or exceptions.

info handle, info signals

Displays information about the handling of signals or processor exceptions.

info threads

Displays information about the available threads.

interrupt, stop

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

next

Steps through an application at the source level stopping at the first instruction of each source line but stepping over all function calls.

nexti

Steps through an application at the instruction level but stepping over all function calls.

nexts

Steps through an application at the source level stopping at the first instruction of each source statement but stepping over all function calls.

reset

Performs a reset on the target.

run

Starts running the target.

set blocking-run-control

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

set debug-from

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

set step-mode

Controls the default behavior of the `step` and `steps` commands.

show blocking-run-control

Displays the setting for blocking run control operations such as stepping and running.

show debug-from

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

show step-mode

Displays the step setting for functions without debug information.

start

Sets a temporary breakpoint, calls the debugger `run` command, and then deletes the temporary breakpoint when it is hit. By default, the temporary breakpoint is set at the address of the global function `main()`.

step

Steps through an application at the source level stopping on the first instruction of each source line including stepping into all function calls.

stepi

Steps through an application at the instruction level including stepping into all function calls.

steps

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.

thread apply

Switches control to a specific thread to execute a debugger command and then switches back to the original state.

thread

Displays information about the current thread.

wait

Instructs the debugger to wait until the target stops.

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

2.2.3 Tracing

List of all the Arm® Debugger commands that can be used to capture trace.

snapshot

Generate a snapshot of target data that can be used with the Arm Development Studio Snapshot Viewer.

trace start

Starts the trace capture on the specified trace capture device.

trace stop

Stops the trace capture on the specified trace capture device.

trace clear

Clears the trace on the specified trace capture device.

trace list

Lists the trace capture devices and trace sources.

trace info

Displays details about trace capture devices and trace sources.

trace dump

Dumps raw trace data to a directory, along with target trace configuration metadata, from a trace capture device or a trace source.

trace report

Produces a trace report, containing the decoded trace data, for the currently selected core.

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

2.2.4 Scripts

List of all the Arm® Debugger commands that can be used to control the debugger using script files.

define

Enables you to derive new user-defined commands from existing commands.

document

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

newvar

Declares and initializes a new debugger convenience variable or register alias.

end

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

if

Enables you to write scripts that conditionally execute debugger commands.

source

Loads and runs a script file to control and debug your target.

while

Enables you to write scripts with conditional loops that execute debugger commands.

usecase help

Displays help for a use case script.

usecase list

Lists use case scripts.

usecase run

Runs a use case script.

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

2.2.5 Call stack

List of all the Arm® Debugger commands that display information about the call stack and others that control the current position in the call stack.

down

Moves and displays the current frame pointer down the call stack towards the bottom frame.

down-silently

Moves the current frame pointer down the call stack towards the bottom frame.

frame

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

info frame

Displays stack frame information at the selected position.

info stack, backtrace, where

Displays a numbered list of the calling stack frames including the function names and source line numbers.

select-frame

Moves the current frame pointer in the callstack.

set backtrace

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

show backtrace

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

up

Moves and displays the current frame pointer up the call stack towards the top frame.

up-silently

Moves the current frame pointer up the call stack towards the top frame.

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

2.2.6 Operating System

List of all the Arm® Debugger commands that enable you to debug applications running on a target with an *Operating System* (OS).

sharedlibrary

Loads symbols from shared libraries.

nosharedlibrary

Discards all loaded shared library symbols.

info os

Displays the current state of the OS support. If OS support is enabled, also lists all available OS data tables.

info os-log

Displays the contents of the OS log buffer for connections that support this feature.

info os-modules

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

info os-version

Displays the version of the OS for connections that support this feature.

info processes

Displays information about the user space processes.

info sharedlibrary

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

info threads

Displays information about the available threads.

set auto-solib-add

Controls the automatic loading of shared library symbols.

set os

Controls OS settings in the debugger. An OS-aware connection must be established before you can use this command.

set solib-search-path

Specifies additional directories to search for shared library symbols.

set stop-on-solib-events

Controls whether the debugger stops execution when a shared object is loaded or unloaded.

set sysroot, set solib-absolute-prefix

Specifies the system root directory to search for shared library symbols.

show auto-solib-add

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

show os

Displays the OS control settings.

show solib-search-path

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

show stop-on-solib-events

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

show sysroot, show solib-absolute-prefix

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

thread apply

Switches control to a specific thread to execute a debugger command and then switches back to the original state.

thread

Displays information about the current thread.

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

2.2.7 Files

List of Arm® Debugger commands that enable you to control the loading and unloading of executable images on to a target and debug information into the debugger.

add-symbol-file

Loads additional debug information into the debugger.

append

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

cd

Changes the current working directory.

directory, set directories

Defines additional directories to search for source files.

discard-symbol-file

Discards debug information relating to a specific file.

dump

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

file, symbol-file

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

info files, info target

Displays information about the loaded image and symbols.

info sources

Displays the names of the source files used in the current image being debugged.

load

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

loadfile

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.

pwd

Displays the current working directory.

reload-symbol-file

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

restore

Reads data from a file and writes it to memory.

set substitute-path

Modifies the search paths used by the debugger when it executes any of the commands that look up and display source code.

show directories

Displays the list of directories to search for source files.

show substitute-path

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

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

2.2.8 Data

List of all the Arm® Debugger commands that enables you to display source code, expressions, variables, functions, classes, memory, and other data.

disassemble

Displays the disassembly for the function surrounding a specific address or the disassembly for a specific address range.

info address

Displays the location of a symbol.

info classes

Displays C++ class names.

info functions

Displays the name and data types for all functions.

info locals

Displays all local variables for the current stack frame.

info members

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

info symbol

Displays the symbol name at a specific address.

info variables

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

list

Displays lines of source code surrounding the current or specified location.

set listsize

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

set variable, set

Evaluates an expression and assigns the result to a variable, register or memory.

show listsize

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

whatis

Displays the data type of an expression.

x

Displays the content of memory at a specific address.

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

Related information

[info handle](#), [info signals](#) on page 117

2.2.9 Memory group

List of all the Arm® Debugger commands that controls memory accesses and displays information about specific memory regions.

append

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

assemble

Writes assembler instructions to memory.

delete memory

Deletes one or more user-defined memory regions.

disable memory

Disables one or more user-defined memory regions.

disassemble

Displays the disassembly for the function surrounding a specific address or the disassembly for a specific address range.

dump

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

enable memory

Enables one or more user-defined memory regions.

info memory

Displays the currently defined memory regions.

info memory-parameters

Displays the memory parameters applicable to an address space.

memory

Defines a memory region and specifies its attributes and size.

memory auto

Resets the memory regions to the default target settings and discards all user-defined regions.

memory debug-cache

Controls the caching by the debugger for all memory regions.

memory fill

Writes a specific pattern of bytes to memory.

memory set

Writes to memory.

memory set_typed

Writes a list of values to memory.

restore

Reads data from a file and writes it to memory.

x

Displays the content of memory at a specific address.

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

2.2.10 Cache

List of all the Arm® Debugger commands that provide information on the available caches.

cache flush

Flushes the caches of the current CPU.

cache list

Lists the caches and related information available for the current core. The output is implementation defined.

cache print

Provides a structured view of the cache data in the current core. The output is implementation defined.

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

2.2.11 Registers

List of all the Arm® Debugger commands that provide register information.

info all-registers

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

info registers

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

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

Related information

[Accessing registers and debugger variables in Arm Debugger expressions](#) on page 15

2.2.12 peripheral

List of all the Arm® Debugger commands for managing peripheral register groups.

peripheral add

Creates a peripheral register group containing the registers from a register frame.

peripheral delete

Deletes peripheral register groups.

peripheral show

Lists available peripherals, components, register frames, and registers.

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

2.2.13 mmu

List of all the Arm® Debugger commands that provide information on the Memory Management Unit.

mmu dirty-state

Displays the in-memory dirty state structures.

mmu list dirty-state

Lists the dirty state structures with associated parameters that are available in the target.

mmu list tables

Lists the available translation tables and their associated parameters.

mmu list translations

Lists the available translations and their associated parameters.

mmu list memory-maps

Lists the available memory maps and their associated parameters.

mmu print

Prints the contents of a translation table.

mmu translate

Performs translations between virtual and physical addresses.

mmu memory-map

Prints the memory map.

set mmu use-cache-for-phys-reads

Instructs the debugger to, where possible, ensure that the translation table entries it reads from physical memory are coherent with the contents of data caches.

show mmu use-cache-for-phys-reads

Displays the MMU setting that controls the coherency between translation table memory reads and cache data.

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

2.2.14 mmu list

`mmu list` commands in Arm® Debugger.

`mmu list dirty-state`

Lists the dirty state structures with associated parameters that are available in the target.

`mmu list memory-maps`

Lists the available memory maps and their associated parameters.

`mmu list tables`

Lists the available translation tables and their associated parameters.

`mmu list translations`

Lists the available translations and their associated parameters.

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

2.2.15 mpu

List of all the Arm® Debugger commands that provide information on the Memory Protection Unit.

`mpu list tables`

Lists the available translation tables and their associated parameters.

`mpu list memory-maps`

Lists the available memory maps and their associated parameters.

`mpu print`

Prints the contents of a translation table.

`mpu memory-map`

Prints the memory map.

`set idau-region`

Specifies the *Implementation Defined Attribution Unit* (IDAU) region parameters for each memory range.

`show idau-region`

Displays the currently specified IDAU region parameters.

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

2.2.16 mpu list

`mpu list` commands in Arm® Debugger.

`mpu list tables`

Lists the available translation tables and their associated parameters.

mpu list memory-maps

Lists the available memory maps and their associated parameters.

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

2.2.17 Display

List of all the Arm® Debugger commands that enable you to display specific output on the command-line.

echo

Displays only textual strings.

output

Displays only the result of an expression.

print, inspect

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.

set print

Controls the current debugger print settings.

show print

Displays the debugger print settings.

x

Displays the content of memory at a specific address.

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

2.2.18 Information

List of all the Arm® Debugger commands that enables you to display information about breakpoints, watchpoints, running processors, variables, functions, classes, registers, memory regions, stack frames, and other data.

info address

Displays the location of a symbol.

info all-registers

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

info breakpoints, info watchpoints

Displays information about the status of all breakpoints and watchpoints.

info breakpoints capabilities

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

info watchpoints capabilities

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

info capabilities

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

info classes

Displays C++ class names.

info cores

Displays information about the running processors.

info files, info target

Displays information about the loaded image and symbols.

info flash

Displays information about the flash devices on the current target.

info frame

Displays stack frame information at the selected position.

info functions

Displays the name and data types for all functions.

info-gcs

Displays a table of *Guarded Control Stack* (GCS) entries.

info inst-sets

Displays the available instruction sets.

info locals

Displays all local variables for the current stack frame.

info members

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

info memory

Displays the currently defined memory regions.

info memory-parameters

Displays the memory parameters applicable to an address space.

info os

Displays the current state of the *Operating System* (OS) support. If OS support is enabled, also lists all available OS data tables.

info os-log

Displays the contents of the OS log buffer for connections that support this feature.

info os-modules

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

info os-version

Displays the version of the OS for connections that support this feature.

info overlays

Displays information about the currently loaded overlays.

info processes

Displays information about the user space processes.

info registers

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

info semihosting

Displays semihosting information.

info sharedlibrary

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

info handle, info signals

Displays information about the handling of signals or processor exceptions.

info sources

Displays the names of the source files used in the current image being debugged.

info stack, backtrace, where

Displays a numbered list of the calling stack frames including the function names and source line numbers.

info symbol

Displays the symbol name at a specific address.

info threads

Displays information about the available threads.

info variables

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

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

2.2.19 log

List of all the Arm® Debugger commands that enable you to control runtime messages from the debugger.

log config

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

log file

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

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

2.2.20 Set

List of all the Arm® Debugger commands that enable you to control the default debugger settings.

set

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

set arm

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

set auto-solib-add

Controls the automatic loading of shared library symbols.

set backtrace

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

set blocking-run-control

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

set breakpoint

Controls the automatic behavior of breakpoints and watchpoints.

set case-insensitive-source-matching

Controls the case sensitivity of debugger file matching operations.

set cde-coprocessors

Specify the coprocessors that are associated with the *Arm Custom Datapath Extension* (CDE).

set debug-agent

Sets an internal configuration parameter for the debug agent.

set debug-from

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

set directories

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

set dtsl-options

Sets a parameter in the DTSL configuration.

set dtsl-temporary-directory

Specifies the path for the temporary directory to store trace data.

set elf cache-uninitialized-sections

Controls whether the debugger caches uninitialized sections.

set elf load-segments-at-p_paddr

Controls whether the specified load offset is + `p_paddr` or + `p_vaddr` when loading segments of ELF images to the target.

set elf zero-extra-segment-bytes

Enables zeroing of bytes from `p_filesz` to `p_memsz` when loading segments of ELF images to the target.

set endian

Specifies the byte order for use by the debugger.

set escape-strings

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

set escapes-in-filenames

Controls the use of special characters in paths.

set idau-region

Specifies the *Implementation Defined Attribution Unit* (IDAU) region parameters for each memory range.

set listsize

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

set mmu use-cache-for-phys-reads

Instructs the debugger to, where possible, ensure that the translation table entries it reads from physical memory are coherent with the contents of data caches.

set os

Controls *Operating System* (OS) settings in the debugger. An OS-aware connection must be established before you can use this command.

set overlays enabled

Enables or disables overlay support.

set print

Controls the current debugger print settings.

set semihosting

Controls the semihosting settings in the debugger.

set solib-search-path

Specifies additional directories to search for shared library symbols.

set step-mode

Controls the default behavior of the `step` and `steps` commands.

set stop-on-solib-events

Controls whether the debugger stops execution when a shared object is loaded or unloaded.

set substitute-path

Modifies the search paths used by the debugger when it executes any of the commands that look up and display source code.

set sysroot, set solib-absolute-prefix

Specifies the system root directory to search for shared library symbols.

set trust-ro-sections-for-opcodes

Controls whether the debugger can read opcodes from read-only sections of images on the host workstation rather than from the target itself.

set variable

Evaluates an expression and assigns the result to a variable, register, or memory.

set wildcard-style

Specifies the type of wildcard pattern matching you can use for examining the contents of strings.

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

2.2.21 set elf

`set elf` commands in Arm® Debugger.

set elf cache-uninitialized-sections

Controls whether the debugger caches uninitialized sections.

set elf load-segments-at-p_paddr

Controls whether the specified load offset is + `p_paddr` or + `p_vaddr` when loading segments of ELF images to the target.

set elf zero-extra-segment-bytes

Enables zeroing of bytes from `p_filesz` to `p_memsz` when loading segments of ELF images to the target.

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

2.2.22 show

List of all the Arm® Debugger commands that enable you to view the default debugger settings.

show

Displays the debugger settings.

show architecture

Displays the architecture of the target.

show arm

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

show auto-solib-add

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

show backtrace

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

show blocking-run-control

Displays the setting for blocking run control operations such as stepping and running.

show breakpoint

Displays the breakpoint and watchpoint behavior settings.

show case-insensitive-source-matching

Displays the case sensitivity setting for the debugger file matching operations.

show cde-coprocessors

Displays the encoding associated with each coprocessor.

show debug-agent

Displays the value of an internal configuration parameter for the debug agent.

show debug-from

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

show directories

Displays the list of directories to search for source files.

show dtsl-options

Displays the value of a parameter in the DTSL configuration.

show dtsl-temporary-directory

Displays the current path for the temporary directory which stores trace data.

show elf cache-uninitialized-sections

Displays the debugger setting that controls whether uninitialized sections are cached.

show elf load-segments-at-p_paddr

Displays the debugger setting that controls the location for loading segments of ELF images.

show elf zero-extra-segment-bytes

Displays the debugger setting that controls zeroing of bytes when loading segments of ELF images to the target.

show endian

Displays the byte order setting in use by the debugger.

show escape-strings

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

show escapes-in-filenames

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

show listsize

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

show idau-region

Displays the currently specified *Implementation Defined Attribution Unit* (IDAU) region parameters.

show mmu use-cache-for-phys-reads

Displays the MMU setting that controls the coherency between translation table memory reads and cache data.

show os

Displays the *Operating System* (OS) control settings.

show print

Displays the debugger print settings.

show semihosting

Displays the semihosting settings in the debugger.

show solib-search-path

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

show step-mode

Displays the step setting for functions without debug information.

show stop-on-solib-events

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

show substitute-path

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

show sysroot, show solib-absolute-prefix

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

show trust-ro-sections-for-opcodes

Displays the 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.

show version

Displays the version number of the debugger.

show wildcard-style

Displays the wildcard style for pattern matching.

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

2.2.23 show elf

`show elf` commands in Arm® Debugger.

show elf cache-uninitialized-sections

Displays the debugger setting that controls whether uninitialized sections are cached.

show elf load-segments-at-p_paddr

Displays the debugger setting that controls the location for loading segments of ELF images.

show elf zero-extra-segment-bytes

Displays the debugger setting that controls zeroing of bytes when loading segments of ELF images to the target.

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

2.2.24 flash

List of all the Arm® Debugger commands that controls flash accesses and displays information about specific flash devices.

flash erase-device

Erases the memory on a specified flash device.

flash erase-image-sectors

Erases all sectors of flash memory in the specified image.

flash load

Loads sections from an image into one or more flash devices.

flash load-multiple

Simultaneously load multiple flash image sections from multiple images, to one or more flash devices.

info flash

Displays information about the flash devices on the current target.

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



Note

To use this command you must check that flash device support is available for your target. If it is not available, you must write your own flash algorithm for this command to work. For details on how to do this, see [Flash programming](#). In Arm Development Studio, the `<armds_installation_directory>/examples/Bare-metal_examples_Armv7.zip/flash_algo-STM32F10x` directory provides an example of how to write a flash algorithm.

2.2.25 Support

List of all the miscellaneous Arm® Debugger commands.

define

Enables you to derive new user-defined commands from existing commands.

help

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

info capabilities

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

info inst-sets

Displays the available instruction sets.

pause

Pauses the execution of a script for a specified period of time.

preprocess

Displays the preprocessed expression, not the evaluated expression.

quit, exit

Quits the debugger session.

set arm

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

set endian

Specifies the byte order for use by the debugger.

set semihosting

Controls the semihosting settings in the debugger.

shell

Runs a shell command within the debug session.

show architecture

Displays the architecture of the target.

show arm

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

show semihosting

Displays the semihosting settings in the debugger.

show version

Displays the version number of the debugger.

show endian

Displays the byte order setting in use by the debugger.

stdin

Specifies semihosting input requested by application code.

snapshot

Generate a snapshot of target data that can be used with the Arm Development Studio Snapshot Viewer.

unset

Modifies the current debugger settings.

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

2.3 Arm Debugger commands listed in alphabetical order

Describes the available Arm® Debugger commands.

Table 2-1: Arm Debugger commands

Debugger command	Description
add-symbol-file	Loads additional debug information into the debugger.
advance	Sets a temporary breakpoint at the specified address and calls the debugger <code>continue</code> command.
append	Reads data from memory or the result of an expression and appends it to an existing file.
assemble	Writes assembler instructions to memory.
awatch	Sets a watchpoint for a data symbol.
backtrace	Displays a numbered list of the calling stack frames including the function names and source line numbers.
break	Sets an execution breakpoint at a specific location.
break-script	Assigns a script file to a specific breakpoint.
break-set-property	Updates the properties of an existing breakpoint.
break-stop-on-cores	Applies an existing breakpoint to one or more processors.
break-stop-on-threads	Applies an existing breakpoint to one or more threads.
break-stop-on-vmid	Applies an existing hardware breakpoint to a Virtual Machine.
cache flush	Flushes the caches of the current CPU.
cache list	Lists the caches and related information available for the current core.
cache print	Provides a structured view of the cache data in the current core.
cd	Changes the current working directory.
clear	Deletes a breakpoint at a specific location.
clearwatch	Deletes a watchpoint at a specific location.
condition	Sets a stop condition for a specific breakpoint or watchpoint.
continue	Continues running the target.
core apply	Switches control to a specific processor to execute a debugger command and then switches back to the original state.
core	Displays information about the current processor.
define	Derives new user-defined commands from existing commands.
delete breakpoints	Deletes one or more breakpoints or watchpoints.
delete memory	Deletes one or more user-defined memory regions.
directory	Defines additional directories to search for source files.
disable breakpoints	Disables one or more breakpoints or watchpoints.
disable memory	Disables one or more user-defined memory regions.
disassemble	Displays the disassembly for the function surrounding a specific address or the disassembly for a specific address range.
discard-symbol-file	Discards debug information relating to a specific file.

Debugger command	Description
document	Adds integrated help for a new user-defined command.
down	Moves and displays the current frame pointer down the call stack towards the bottom frame.
down-silently	Moves the current frame pointer down the call stack towards the bottom frame.
dump	Reads data from memory or the result of an expression and writes it to a file.
echo	Displays only textual strings.
enable breakpoints	Enables one or more breakpoints or watchpoints by number.
enable memory	Enables one or more user-defined memory regions.
end	Terminates conditional blocks when using the <code>define</code> , <code>if</code> , and <code>while</code> commands.
exit	Quits the debugger session.
file	Loads debug information from an image into the debugger and records the entry point address for future use by the <code>run</code> and <code>start</code> commands.
finish	Continues running the device to the next instruction after the selected stack frame finishes.
flash erase-device	Erases the memory on a specified flash device.
flash erase-image-sectors	Erases all sectors of flash memory in the specified image.
flash load	Loads sections from an image into one or more flash devices.
flash load-multiple	Load multiple images on to your target.
frame	Sets the current frame pointer in the call stack and also displays the function name and source line number for the specified frame.
gcs-print	Displays a table of <i>Guarded Control Stack</i> (GCS) entries.
handle	Controls the handler settings for one or more signals or exceptions.
hbreak	Sets a hardware execution breakpoint at a specific location.
help	Displays help information for a specific command or a group of commands listed according to specific debugging tasks.
if	Allows you to write scripts that conditionally execute debugger commands.
ignore	Sets the ignore counter for a breakpoint or watchpoint condition.
info address	Displays the location of a symbol.
info all-registers	Displays the name and content of grouped registers for the current stack frame.
info-breakpoints	Displays information about the status of all breakpoints and watchpoints.
info breakpoints capabilities	Displays a list of parameters that you can use with breakpoint commands for the current connection.
info capabilities	Displays a list of capabilities for the target device that is currently connected to the debugger.
info classes	Displays C++ class names.
info cores	Displays information about the running processors.
info files	Displays information about the loaded image and symbols.

Debugger command	Description
info flash	Displays information about the flash devices on the current target.
info frame	Displays stack frame information at the selected position.
info functions	Displays the name and data types for all functions.
info-gcs	Displays a table of GCS entries.
info-handle	Displays information about the handling of signals or processor exceptions.
info inst-sets	Displays the available instruction sets.
info locals	Displays all local variables for the current stack frame.
info members	Displays the name and data types for all class member variables that are accessible in the function corresponding to the selected stack frame.
info memory	Displays the currently defined memory regions.
info memory-parameters	Displays the memory parameters applicable to an address space.
info os	Displays the current state of the <i>Operating System</i> (OS) support.
info os-log	Displays the contents of the Operating System log buffer for connections that support this feature.
info os-modules	Displays a list of loadable kernel modules for connections that support this feature.
info os-version	Displays the version of the Operating System for connections that support this feature.
info overlays	Displays information about the currently loaded overlays.
info processes	Displays information about the user space processes.
info registers	Displays the name and content of all application level registers for the current stack frame.
info semihosting	Displays semihosting information.
info sharedlibrary	Displays the names of the loaded shared libraries, the base address, and whether the debug symbols of the shared libraries are loaded or not.
info-signals	Displays information about the handling of signals or processor exceptions.
info sources	Displays the names of the source files used in the current image being debugged.
info-stack	Displays a numbered list of the calling stack frames including the function names and source line numbers.
info symbol	Displays the symbol name at a specific address.
info target	Displays information about the loaded image and symbols.
info threads	Displays information about the available threads.
info variables	Displays the name and data types for all global and static variables.
info-watchpoints	Displays information about the status of all breakpoints and watchpoints.
info watchpoints capabilities	Displays a list of parameters that you can use with watchpoint commands for the current connection.
inspect	Displays the output of an expression and also records the result in a new debugger variable.
interrupt, stop	Interrupts the target and stops the application if it is running.

Debugger command	Description
<code>list</code>	Displays lines of source code surrounding the current or specified location.
<code>load</code>	Loads an image on to the target and records the entry point address for future use by the <code>run</code> and <code>start</code> commands.
<code>loadfile</code>	Loads debug information into the debugger, an image on to the target and records the entry point address for future use by the <code>run</code> and <code>start</code> commands.
<code>log config</code>	Specifies the type of logging configuration to output runtime messages from the debugger.
<code>log file</code>	Specifies an output file to receive runtime messages from the debugger.
<code>memory</code>	Defines a memory region and specifies its attributes and size.
<code>memory auto</code>	Resets the memory regions to the default target settings and discards all user-defined regions.
<code>memory debug-cache</code>	Controls the caching by the debugger for all memory regions.
<code>memory fill</code>	Writes a specific pattern of bytes to memory.
<code>memory set</code>	Writes to memory.
<code>memory set_typed</code>	Writes a list of values to memory.
<code>mmu dirty-state</code>	Displays the in-memory dirty state structures.
<code>mmu list dirty-state</code>	Lists the dirty state structures with associated parameters that are available in the target.
<code>mmu list memory-maps</code> , <code>mpu list memory-maps</code>	Lists the available memory maps and their associated parameters.
<code>mmu list tables</code> , <code>mpu list tables</code>	Lists the available translation tables and their associated parameters.
<code>mmu list translations</code>	Lists the available translations and their associated parameters.
<code>mmu memory-map</code> , <code>mpu memory-map</code>	Prints the memory map.
<code>mmu print</code> , <code>mpu print</code>	Prints the contents of a translation table.
<code>mmu translate</code>	Performs translations between virtual and physical addresses.
<code>newvar</code>	Declares and initializes a new debugger convenience variable or register alias.
<code>next</code>	Steps through an application at the source level stopping at the first instruction of each source line but stepping over all function calls.
<code>nexti</code>	Steps through an application at the instruction level but stepping over all function calls.
<code>nexts</code>	Steps through an application at the source level stopping at the first instruction of each source statement but stepping over all function calls.
<code>nosharedlibrary</code>	Discards all loaded shared library symbols.
<code>output</code>	Displays only the result of an expression.
<code>pause</code>	Pauses the execution of a script for a specified period of time.
<code>peripheral add</code>	Creates a peripheral register group containing the registers from a register frame.
<code>peripheral delete</code>	Deletes peripheral register groups.
<code>peripheral show</code>	Lists available peripherals, components, register frames, and registers.
<code>preprocess</code>	Displays the preprocessed expression, not the evaluated expression.

Debugger command	Description
<code>print</code>	Displays the output of an expression and also records the result in a new debugger variable.
<code>pwd</code>	Displays the current working directory.
<code>quit</code>	Quits the debugger session.
<code>reload-symbol-file</code>	Reloads debug information from an already loaded image into the debugger using the same settings as the original load operation.
<code>reset</code>	Performs a reset on the target.
<code>resolve</code>	Re-evaluates the specified breakpoints or watchpoints and those with addresses that can be resolved are set.
<code>restore</code>	Reads data from a file and writes it to memory.
<code>run</code>	Starts running the target.
<code>rwatch</code>	Sets a watchpoint for a data symbol.
<code>select-frame</code>	Moves the current frame pointer in the call stack.
<code>set arm</code>	Controls the behavior of the debugger when selecting the instruction set for disassembly and setting breakpoints.
<code>set auto-solib-add</code>	Controls the automatic loading of shared library symbols.
<code>set backtrace</code>	Controls the default behavior when using the <code>info stack</code> command.
<code>set blocking-run-control</code>	Controls whether run control operations such as stepping and running are blocked until the target stops or released immediately.
<code>set breakpoint</code>	Controls the automatic behavior of breakpoints and watchpoints.
<code>set case-insensitive-source-matching</code>	Controls the case sensitivity of debugger file matching operations.
<code>set cde-coprocessors</code>	Specify the coprocessors that are associated with the Arm <i>Custom Datapath Extension</i> (CDE).
<code>set debug-agent</code>	Sets an internal configuration parameter for the debug agent.
<code>set debug-from</code>	Specifies the address of the temporary breakpoint for subsequent use by the <code>start</code> command.
<code>set-directories</code>	Defines additional directories to search for source files.
<code>set dtsl-options</code>	Sets a parameter in the DTSL configuration.
<code>set dtsl-temporary-directory</code>	Specifies the path for the temporary directory to store trace data.
<code>set elf cache-uninitialized-sections</code>	Controls whether the debugger caches uninitialized sections.
<code>set elf load-segments-at-p_paddr</code>	Controls whether the specified load offset is + <code>p_paddr</code> or + <code>p_vaddr</code> when loading segments of ELF images to the target.
<code>set elf zero-extra-segment-bytes</code>	Enables zeroing of bytes from <code>p_filesz</code> to <code>p_memsz</code> when loading segments of ELF images to the target.
<code>set endian</code>	Specifies the byte order for use by the debugger.
<code>set escape-strings</code>	Controls how special characters in strings are printed on the debugger command-line.
<code>set escapes-in-filenames</code>	Controls the use of special characters in paths.
<code>set idau-region</code>	Specifies the <i>Implementation Defined Attribution Unit</i> (IDAU) region parameters for each memory range.
<code>set listsize</code>	Modifies the default number of source lines that the <code>list</code> command displays.

Debugger command	Description
<code>set mmu use-cache-for-phys-reads</code>	Instructs the debugger to, where possible, ensure that the translation table entries it reads from physical memory are coherent with the contents of data caches.
<code>set os</code>	Controls operating system settings in the debugger.
<code>set overlays enabled</code>	Enables or disables overlay support.
<code>set print</code>	Controls the current debugger print settings.
<code>set semihosting</code>	Controls the semihosting settings in the debugger.
<code>set-solib-absolute-prefix</code>	Specifies the system root directory to search for shared library symbols.
<code>set solib-search-path</code>	Specifies additional directories to search for shared library symbols.
<code>set step-mode</code>	Controls the default behavior of the <code>step</code> and <code>steps</code> commands.
<code>set stop-on-solib-events</code>	Controls whether the debugger stops execution when a shared object is loaded or unloaded.
<code>set substitute-path</code>	Modifies the search paths used by the debugger when it executes any of the commands that look up and display source code.
<code>set-sysroot</code>	Specifies the system root directory to search for shared library symbols.
<code>set trust-ro-sections-for-opcodes</code>	Controls whether the debugger can read opcodes from read-only sections of images on the host workstation rather than from the target itself.
<code>set variable, set</code>	Evaluates an expression and assigns the result to a variable, register, or memory address.
<code>set wildcard-style</code>	Specifies the type of wildcard pattern matching you can use for examining the contents of strings.
<code>sharedlibrary</code>	Loads symbols from shared libraries.
<code>shell</code>	Runs a shell command in the debug session.
<code>show</code>	Displays the debugger settings.
<code>show architecture</code>	Displays the architecture of the target.
<code>show arm</code>	Displays the instruction set settings in use by the debugger for disassembly and setting breakpoints.
<code>show auto-solib-add</code>	Displays the automatic setting for use when loading shared library symbols.
<code>show backtrace</code>	Displays the behavior settings for use with the <code>info stack</code> command.
<code>show blocking-run-control</code>	Displays the setting for blocking run control operations such as stepping and running.
<code>show breakpoint</code>	Displays the breakpoint and watchpoint behavior settings.
<code>show case-insensitive-source-matching</code>	Displays the case sensitivity setting for the debugger file matching operations.
<code>show cde-coprocessors</code>	Displays the encoding associated with each coprocessor.
<code>show debug-agent</code>	Displays the value of an internal configuration parameter for the debug agent.
<code>show debug-from</code>	Displays the setting for the expression that is used by the <code>start</code> command to set a temporary breakpoint.
<code>show directories</code>	Displays the list of directories to search for source files.

Debugger command	Description
<code>show dtsl-options</code>	Displays the value of a parameter in the DTSL configuration.
<code>show dtsl-temporary-directory</code>	Displays the current path for the temporary directory which stores trace data.
<code>show elf cache-uninitialized-sections</code>	Displays the debugger setting that controls whether uninitialized sections are cached.
<code>show elf load-segments-at-p_paddr</code>	Displays the debugger setting that controls the location for loading segments of ELF images.
<code>show elf zero-extra-segment-bytes</code>	Displays the debugger setting that controls zeroing of bytes when loading segments of ELF images to the target.
<code>show endian</code>	Displays the byte order setting in use by the debugger.
<code>show escape-strings</code>	Displays the setting for controlling how special characters in strings are printed on the debugger command line.
<code>show escapes-in-filenames</code>	Displays the setting for controlling the use of special characters in paths.
<code>show idau-region</code>	Displays the currently specified IDAU region parameters.
<code>show listsize</code>	Displays the number of source lines that the <code>list</code> command displays.
<code>show mmu use-cache-for-phys-reads</code>	Displays the MMU setting that controls the coherency between translation table memory reads and cache data.
<code>show os</code>	Displays the OS control settings.
<code>show print</code>	Displays the debugger print settings.
<code>show semihosting</code>	Displays the semihosting settings in the debugger.
<code>show-solib-absolute-prefix</code>	Displays the system root directory in use by the debugger when searching for shared library symbols.
<code>show solib-search-path</code>	Displays the search paths in use by the debugger when searching for shared libraries.
<code>show step-mode</code>	Displays the step setting for functions without debug information.
<code>show stop-on-solib-events</code>	Displays the debugger setting that controls whether execution stops when shared library events occur.
<code>show substitute-path</code>	Displays the search path substitution rules in use by the debugger when searching for source files.
<code>show-sysroot</code>	Displays the system root directory in use by the debugger when searching for shared library symbols.
<code>show trust-ro-sections-for-opcodes</code>	Displays the 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.
<code>show version</code>	Displays the version number of the debugger.
<code>show wildcard-style</code>	Displays the wildcard style for pattern matching.
<code>silence</code>	Disables the printing of stop messages for a specific breakpoint.
<code>snapshot</code>	Generate a snapshot of target data that can be used with the Arm Development Studio <i>Snapshot Viewer</i> .
<code>source</code>	Loads and runs a script file to control and debug your target.
<code>start</code>	Sets a temporary breakpoint, calls the debugger <code>run</code> command, and then deletes the temporary breakpoint when it is hit.
<code>stdin</code>	Specifies semihosting input requested by application code.

Debugger command	Description
<code>step</code>	Steps through an application at the source level stopping on the first instruction of each source line including stepping into all function calls.
<code>stepi</code>	Steps through an application at the instruction level including stepping into all function calls.
<code>steps</code>	Steps through an application at the source level stopping on the first instruction of each source statement including stepping into all function calls.
<code>stop</code>	<code>stop</code> is an alias for <code>interrupt</code> .
<code>symbol</code>	Loads debug information from an image into the debugger and records the entry point address for future use by the <code>run</code> and <code>start</code> commands.
<code>tbreak</code>	Sets an execution breakpoint at a specific location and deletes the breakpoint when it is hit.
<code>thbreak</code>	Sets a hardware execution breakpoint at a specific location and deletes the breakpoint when it is hit.
<code>thread</code>	Displays information about the current thread.
<code>thread apply</code>	Switches control to a specific thread to execute a debugger command and then switches back to the original state.
<code>trace clear</code>	Clears the trace on the specified trace capture device.
<code>trace dump</code>	Dumps raw trace data to a directory, along with target trace configuration metadata, from a trace capture device or a trace source.
<code>trace info</code>	Displays details about trace capture devices and trace sources.
<code>trace list</code>	Lists the trace capture devices and trace sources.
<code>trace report</code>	Produces a trace report, containing the decoded trace data, for the currently selected core.
<code>trace start</code>	Starts trace capture on the specified trace capture device.
<code>trace stop</code>	Stops trace capture on the specified trace capture device.
<code>unset</code>	Modifies the current debugger settings.
<code>unsilence</code>	Enables the printing of stop messages for a specific breakpoint.
<code>up</code>	Moves and displays the current frame pointer up the call stack towards the top frame.
<code>up-silently</code>	Moves the current frame pointer up the call stack towards the top frame.
<code>usecase help</code>	Displays help for a use case script.
<code>usecase list</code>	Lists use case scripts.
<code>usecase run</code>	Runs a use case script.
<code>wait</code>	Instructs the debugger to wait until the target stops.
<code>watch</code>	Sets a watchpoint for a data symbol.
<code>watch-set-property</code>	Updates the properties of an existing watchpoint.
<code>whatis</code>	Displays the data type of an expression.
<code>where</code>	Displays a numbered list of the calling stack frames including the function names and source line numbers.

Debugger command	Description
while	Allows you to write scripts with conditional loops that execute debugger commands.
x	Displays the content of memory at a specific address.

2.3.1 add-symbol-file

Loads additional debug information into the debugger.

Syntax

```
add-symbol-file {filename} [offset] [-s section address]...
```

Alias:

```
add {filename} [offset] [-s section address]...
```

Parameters

filename

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



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 in 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.

s

For relocatable objects, this specifies the address to which a section was relocated.

section

Specifies the name of the relocated section.

address

Specifies the address of the section. This can be either an address or an expression that evaluates to an address. You can also specify the address space.

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

Example: Loading symbols

```

add-symbol-file myFile.axf           # Load symbols with no offset
add-symbol-file myLib.so             # Load/pend symbols for shared library
add-symbol-file myModule.ko         # Load/pend symbols for Linux kernel
                                     # module
add-symbol-file myFile.axf 0x2000    # Load symbols with offset of 0x2000
                                     # from the base address
add-symbol-file relocate.o -s .text 0x1000 -s .data 0x2000
                                     # Load symbols from relocate.o with
                                     # section .text relocated to 0x1000 and
                                     # section .data relocated to 0x2000
add-symbol-file vmlinux N:0          # Load symbols for the AArch32
                                     # non-secure address space, with zero
                                     # offset
add-symbol-file vmlinux EL2N:0x4080000000
                                     # Load symbols for the AArch64 EL2N
                                     # address space, with offset
                                     # 0x4080000000

```

Related information

[Files](#) on page 36

2.3.2 advance

Sets a temporary breakpoint at the specified address and calls the debugger `continue` command.

Syntax

```

advance [-p] [filename:]{line_num}
advance [-p] [filename:]{function}
advance [-p] [filename:]{label}
advance [-p] *<address>
advance +<offset> | -<offset>

```

Alias:

```

adv [-p] [filename:]{line_num}
adv [-p] [filename:]{function}
adv [-p] [filename:]{label}
adv [-p] *<address>
adv +<offset> | -<offset>

```

Parameters

-p

Creates pending breakpoints for unrecognized locations.

By default, specifying an unrecognized breakpoint location (for example, a non-existent function name) results in an error.

The `-p` option creates pending breakpoints for unrecognized locations instead. This is useful when debugging shared libraries. Shared libraries are loaded on demand, so locations are unrecognized until the library is loaded. For more information, see [Pending breakpoints and watchpoints](#).



If you want to debug a shared library, you must load debug symbols from the shared library as well as the application itself. For more information, see [About debugging shared libraries](#).

filename

Sets a temporary breakpoint on a function, label, or line number in the specified source file.

Functions and labels are usually unique, so the debugger can identify the breakpoint location from the name alone.

However, if you have ambiguous function or label names in your source code, for example static functions named `myfunc` in both `file_a.c` and `file_b.c`, use the filename to identify the precise function. For example, advance `file_a.c:myfunc`.

line_num

Sets a breakpoint at the specified line number in the source file `filename`.

If no `filename` is specified, the debugger assumes the source file containing the current location.

function

Sets a breakpoint on the specified function name.

label

Sets a breakpoint on the specified assembly label.



You can only set breakpoints on labels that are present in the executable image. Toolchains might not preserve all symbol names in the final image by default. For example, when using Arm® Compiler 5 you must specify either the `KEEP` assembler directive or the `armasm --keep` option to retain local symbols. For more details, see the [KEEP directive](#) and [--keep command-line option](#) `armasm` documentation.

***<address>**

Sets a breakpoint at the specified address. Specify either an address (for example advance `*0x8000024c`) or an expression that evaluates to an address (for example advance `*$R4+64` or advance `*$PC+256`). For more information about expressions, see [Expressions in the Arm Debugger](#).

{+<offset> | -<offset>}

Sets a breakpoint on the source code line offset from the current location by the specified amount.

Operation

Use the `advance` command to halt execution at a particular point in your code, for example a specific function, source code line number, or instruction memory address.

Execution continues until it hits the temporary breakpoint (or until execution halts for another reason, for example the end of the program is reached). Temporary breakpoints are deleted when hit.

The `advance` command returns control as soon as the target is running. You can use the `wait` command to block the debugger from returning control until, for example, the application completes or a breakpoint is hit. This is useful if you are scripting Arm Debugger commands and do not want subsequent commands to run until after the breakpoint has been reached.

Example: Setting breakpoints and resuming execution

```
advance func1           # Set a temporary breakpoint at func1, then resume
                        # execution.
advance -p lib.c:foo     # Set a temporary breakpoint at foo in lib.c, then
                        # resume execution. If foo is not yet loaded (for
                        # example, if it is in a shared library), the debugger
                        # creates a pending breakpoint.
```

Related information

[Execution control](#) on page 31

[continue](#) on page 80

[wait](#) on page 251

[Expressions in the Arm Debugger](#) on page 15

[About debugging shared libraries](#)

[Pending breakpoints and watchpoints](#)

2.3.3 append

Reads data from memory or the result of an expression and appends it as binary to an existing binary file.

Syntax

```
append [binary] memory {filename} {start_address} {end_address | +<size>}
append [binary] value {filename} {expression}
```

Alias:

```
ap [binary] m {filename} {start_address} {end_address | +<size>}
ap [binary] v {filename} {expression}
```

Parameters

binary

Specifies that the output format is binary, which is the default. This value is not required and is only included here for compatibility with older versions of the debugger.

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: Reading data from memory and appending to a binary file

```
append memory myFile.bin 0x8000 0x8FFF # Append content of memory 0x8000-0x8FFF
                                         # to binary file myFile.bin
append value myFile.bin myArray        # Append content of myArray
                                         # to binary file myFile.bin
```

Related information

[Files](#) on page 36

[Memory group](#) on page 39

[Expressions in the Arm Debugger](#) on page 15

2.3.4 assemble

Writes assembler instructions to memory.

The debugger performs inline assembly of the instructions between the `assemble` and `end` commands, using the specified instruction set, and then writes them to the specified memory location.

Syntax

```
assemble {address} [InstructionSet] # comment
[Instruction]                       ; comment
...
end                                # comment
```

Parameters**address**

Specifies the address to write the first instruction to. Subsequent instructions are written to following memory.

InstructionSet

Specifies the instruction set to assemble to. This can be:

- ARM
- Thumb
- A32

- T32
- A64

You can only specify an instruction set that is available for the processor. If you do not specify the instruction set, it defaults to the instruction set state at the specified address.

Instruction

Assembler instruction to write to memory. You can specify multiple instructions. Each instruction must be on a separate line.

You can also specify supported directives. The supported directives are:

- ARM
- THUMB
- CODE32
- CODE16
- A64
- DCB
- DCD
- DCDU
- DCDO
- DCFD
- DCFDU
- DCFS
- DCFSU
- DCI
- DCQ
- DCQU
- DCW
- DCWU



The syntax for the instructions and directives follows the Arm assembly language syntax.

end

Specifies the end of the `assemble` command. The list of assembler instructions are written to memory only when you enter `end`.

comment

For comments after an `assemble` or `end` command, use the hash `#` character at the beginning of your comment.

For comments after an instruction or directive, use the semicolon `;` character at the beginning of your comment.

Restrictions

The `assemble` command does not change the processor state. You must ensure that the processor is in the correct state to execute the new instructions.

Operation

This command is useful for making small changes to your code without recompiling. For larger code changes or to make use of a wider set of assembler directives you must use the standalone assembler tool provided by your compiler toolchain.

Example: Writing Arm instructions to memory

```
assemble $pc ARM      # Assemble the following Arm instructions
    ADD r1,r2,r3      ; Write the A32 ADD instruction to address $PC
    SUB r2,r3,r4      ; Write the A32 SUB instruction to address $PC+4
    DMB               ; Write Data Memory Barrier to $PC+8
    THUMB             ; Assemble the following THUMB instructions
    MOVS r0,#10        ; Write T32 move instruction to $PC+12
end                   # End of the assemble command
```

Example: Writing directives to memory

```
assemble 0x00008000 # Assemble the following directives
    DCB 0,1,2,3      ; Write four bytes to 0x00008000
    DCD 7,8          ; Write two words to 0x00008004 and 0x00008008
end                 # End of the assemble command
```

Related information

[Memory group](#) on page 39

[Arm Compiler armasm User Guide](#)

[Syntax of source lines in assembly language](#)

2.3.5 awatch

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

Syntax

```
awatch [-d] [-p] [-w <width> | -m <mask> | -s <size>] [-f] {[filename:]symbol |
* <address>} [vmid <number>] [if <condition>]
```

Alias:

```
aw [-d] [-p] [-w <width> | -m <mask> | -s <size>] [-f] {[filename:]symbol |
*<address>} [vmid <number>] [if <condition>]
```

Parameters

-d

Creates the watchpoint disabled.

-p

Specifies whether or not the resolution of an unrecognized watchpoint location results in a pending watchpoint being created.

-w <width>

Specifies the width to watch at the given address, in bits. Accepted values are: 8, 16, 32, and 64 if supported by the target. This option is optional.

The width defaults to:

- 32 bits for an address.
- The width corresponding to the type of the symbol or expression, if entered.

-m <mask>

This option is only available for Arm®v6-M, Armv7-M, Armv8-M, Armv8-R, Armv8-A, and Armv9-A hardware.

Specifies a mask that represents a memory range. The mask value is the number of least significant bits that will be masked. For example, a value of 5 would watch a range of 32 bytes.



The range could be widened if the watched address is not specified as a power of 2.

-s <size>

This option is only available for Armv6-M, Armv7-M, Armv8-M, Armv8-R, Armv8-A, and Armv9-A hardware.

Specifies the size of a memory area to watch. For Armv6-M and Armv7-M, if the <size> and *<address> are not specified so that the range is exactly coverable by an address mask, you must set the -f option.

-f

This option is only available for Armv6-M, Armv7-M, Armv8-M, Armv8-R, Armv8-A, and Armv9-A hardware.

You must set this parameter if the hardware does not support arbitrary ranges and the range described by *<address> and <size> is not exactly coverable by an address mask. If set, a

mask is used that is inclusive of the range specified by `<size>`, but may extend beyond that range. If the mask extends beyond the range specified by `<size>`, the Arm Debugger could halt if there are accesses outside of the intended range.

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 option can be either an address or an expression that evaluates to an address.

vmid `<number>`

Specifies the Virtual Machine ID (VMID) to apply the watchpoint to. This option can be either an integer or an expression that evaluates to an integer. Applicable only on targets which support hypervisor / virtual machine debugging.

if `<condition>`

Specifies the condition which must evaluate to true at the time the watchpoint is triggered for the target to stop. You can create several conditional watchpoints, but when a conditional watchpoint is enabled, no other watchpoints (regardless of whether they are conditional) can be enabled.

Operation

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.

Watchpoints are supported on single memory addresses for all processors, on hardware and models. For M-class Arm processor hardware, watchpoints are also supported on structs and arrays, and arbitrary memory ranges.

The availability of watchpoints depends on your target. In the case of Linux application debug using gdbserver, the availability of watchpoints also depends on the Linux kernel version and configuration.

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.

Example: Setting watchpoints to stop on read or write

```
awatch myVar1           # Set a read/write watchpoint on myVar1
awatch *0x80D4          # Set a read/write watchpoint on address
                        # 0x80D4
awatch myVar1 if myVar1 == 2 # Set a read/write watchpoint on myVar1 which
                        # is hit only if myVar1 evaluates to 2
awatch myVar1 if ($LR & 0xFF) == 0x12 # Set a read/write watchpoint on myVar1 which
                        # is hit only if ($LR & 0xFF) evaluates
                        # to 0x12 when myVar1 is accessed
```

```
awatch -s 200 *0x80D4      # On Armv6-M, Armv7-M, Armv8-A, and
                           # Armv8-R hardware, set a read/write
                           # watchpoint on address 0x80D4 with a
                           # range of 200 bytes
awatch -m 5 *0x8000        # On Armv6-M, Armv7-M, Armv8-A, and
                           # Armv8-R hardware, set a read/write
                           # watchpoint on address 0x8000 with a
                           # range of 32 bytes
```

Related information

[watch](#) on page 251

[rwatch](#) on page 175

[clearwatch](#) on page 79

[Breakpoints and watchpoints](#) on page 29

2.3.6 backtrace

`backtrace` is an alias for [info stack and where](#).

2.3.7 break

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.

Syntax

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

Alias:

```
b [-d] [-p] [[filename:] location | *<address>][[thread | core] number...] [if
expression]
```

Parameters

-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.

numberSpecifies 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 *Program Counter* (PC).

Operation

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 in 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.

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

Example: Setting a breakpoint

```
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
```

```
break function1 if x>0      # my_File.c
                             # Set conditional breakpoint that stops execution
                             # when x>0
break *0x80000000 if $thread==32 # Set conditional breakpoint that stops execution
                             # when thread ID is 32.
break *0x80000000 if $pid==928  # Set conditional breakpoint that stops execution
                             # when process ID is 928.
```

Related information

[hbreak](#) on page 104

[tbreak](#) on page 232

[thbreak](#) on page 233

[resolve](#) on page 172

[clear](#) on page 78

[Breakpoints and watchpoints](#) on page 29

[Expressions in the Arm Debugger](#) on page 15

2.3.8 break-script

Assigns a script file to a specific breakpoint. The script executes when the breakpoint is triggered.

Syntax

```
break-script {number} [filename]
```

Parameters

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.

Restrictions

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 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: Running script when breakpoint triggered

The following command runs myScript.ds when breakpoint 1 is triggered.

```
break-script 1 myScript.ds
```

Related information

[Breakpoints and watchpoints](#) on page 29

2.3.9 break-set-property

Updates the properties of an existing breakpoint.

Syntax

```
break-set-property {number} {property}
```

Alias:

```
break-se {number} {property}
```

Parameters

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.

property

Specifies the property to set. The valid properties are:

if[expression]

Specifies an expression that is evaluated when the breakpoint is hit. If the value of the expression evaluates to true, then the debugger stops the target, otherwise execution resumes. If no expression is specified then the breakpoint condition is deleted.

core[id]

The current core ID. You can use `info cores`, `info processes`, or `info threads` to display the ID numbers.

thread[id]

The current thread ID. You can use `info cores`, `info processes`, or `info threads` to display the ID numbers.

Other target-dependent properties

This command supports other types of `property` depending on your target. Use the [info breakpoints capabilities](#) command to display a list of `property` types that you can use for the current connection.

Example: Updating if statement in a breakpoint

The following command updates the 'if' property of breakpoint 4 with the breakpoint only hit if myVar1 evaluates to 2.

```
break-set-property 4 if myVar1 == 2
```

2.3.10 break-stop-on-cores

`break-stop-on-cores` is an alias for [break-stop-on-threads](#).

2.3.11 break-stop-on-threads, break-stop-on-cores

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

Syntax

```
break-stop-on-threads {number} [id]...  
break-stop-on-cores {number} [id]...
```

Parameters

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: Applying breakpoint to thread

The following command applies breakpoint 1 to thread 2.

```
break-stop-on-threads 1 2
```

Example: Applying breakpoint to multiple threads

The following command applies breakpoint 4 to threads 9 and 11.

```
break-stop-on-threads 4 9 11
```

Example: Applying breakpoint to processors

The following command applies breakpoint 4 to all processors.

```
break-stop-on-cores 4
```

Related information

[Breakpoints and watchpoints](#) on page 29

2.3.12 break-stop-on-vmid

Applies an existing hardware breakpoint to a Virtual Machine (VM).

Syntax

```
break-stop-on-vmid {number} [vmid]
```

Parameters

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: Applying hardware breakpoint to a vmid

The following command applies hardware breakpoint 1 to vmid 2.

```
break-stop-on-vmid 1 2
```

Related information

[Breakpoints and watchpoints](#) on page 29

2.3.13 cache flush

Flushes the caches of the current CPU. This might affect the caches of the other CPUs depending on the cache hierarchy. The precise behavior is implementation defined.



Note

This command might be slow depending on the size of the caches and the available flush methods.

Syntax

```
cache flush
```

Alias:

```
cache f
```

Parameters

None.

Example: Flushing the cache

The following command flushes the caches of the current CPU.

```
cache flush
```

Related information

[Cache](#) on page 40

[About debugging MMUs](#)

2.3.14 cache list

Lists the caches and related information available for the current core. The output is implementation defined.

Syntax

```
cache list
```

Alias:

```
cache l
```

Parameters

None.

Restrictions

The availability of the command and the available caches depend on the specific device that the debugger is connected to.

Example: Listing cache information

```
cache list      # Lists the available caches and views. An example output is:
                  L1D:
                   L1 data cache, size=32k, views: [tags, tlb]
                   ...
                  L1I:
```

```
L1 instruction cache, size=2k, views: [tags, tlb]
...
```

Related information

[Cache](#) on page 40

2.3.15 cache print

Provides a structured view of the cache data in the current core. The output is implementation defined.

Syntax

```
cache print {cache} [view]... [/x]
```

Alias:

```
cache p {cache} [view]... [/x]
```

Parameters

cache

Specifies the cache name.

view

Specifies the view name for the selected cache. For each cache, views provide access to different sets of data, or data presented in different formats.

/x

Specifies that invalid cache data lines are not printed.

Restrictions

The availability of the command and the available caches depend on the specific device that the debugger is connected to.

Example: Printing a structured view of the cache data

```
cache print L1D          # Prints L1 data cache. An example output is:
                           tags:
                           ...
                           tlb:
                           ...

cache print L1D tags      # Prints L1 data cache. An example output is:
                           tags:
                           ...
```

Related information

[Cache](#) on page 40

2.3.16 cd

Changes the current working directory.

Syntax

```
cd {dir}
```

Parameters

dir

Specifies the directory.

Example: Changing working directory

The following command changes the current working directory to `\usr\source`.

```
cd "\usr\source"
```

Related information

[Files](#) on page 36

2.3.17 clear

Deletes a breakpoint at a specific location.

Syntax

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

Parameters

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>}

is a 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: Clearing breakpoints

```
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
```

Related information

[break](#) on page 70

[hbreak](#) on page 104

[tbreak](#) on page 232

[thbreak](#) on page 233

[resolve](#) on page 172

[Breakpoints and watchpoints](#) on page 29

2.3.18 clearwatch

Deletes a watchpoint at a specific location.

Syntax

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

Parameters**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: Deleting a watchpoint

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

Related information

[watch](#) on page 251

[rwatch](#) on page 175

[awatch](#) on page 67

[Breakpoints and watchpoints](#) on page 29

2.3.19 condition

Sets a stop 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]
```

Alias:

```
cond {number} [expression]
```

Parameters

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: Setting break conditions

The following command sets break condition `myVar<5` for breakpoint number 1.

```
condition 1 myVar<5
```

Related information

[Breakpoints and watchpoints](#) on page 29

2.3.20 continue

Continues running the target.



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]
```

Alias:

```
c [count]  
fg [count]
```

Parameters

count

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

Example: Continue running target

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

Related information

[Execution control](#) on page 31

[advance](#) on page 62

2.3.21 core apply

Switches control to a specific processor to execute a 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

```
core apply [all|id] {command}
```

Alias:

```
core a [all|id] {command}
```

Parameters

all

Specifies all processors.

id

Specifies the unique processor number. You can use `info processes` or `info threads` to display the `id` numbers.

command

Specifies the debugger command that you want to execute.

If `all` is specified then the command executes on each processor successively before switching back.

Example: Switching control to execute a command

The following command cycles through all core and print address in PC register (hexadecimal).

```
core apply all print /x $pc
```

Related information

[Execution control](#) on page 31

[Operating System](#) on page 35

2.3.22 core

Displays information about the current processor.

Syntax

```
core [id]
```

Parameters

id

Specifies the unique processor number.

If `id` is not specified, then the debugger switches control to the current processor before displaying information. You can use `info cores` or `info processes` to display the `id` numbers.

If `id` is specified, then the debugger switches control to that processor before displaying the information. Registers and call stacks are associated with a particular processor. This means that switching context also switches the registers and call stack to those belonging to the current processor.

Outputs

This command displays:

- The unique `id` number assigned by the debugger.
- The processor state. For example, `stopped` or `running`.
- The current stack frame, including function names and source line numbers.



`core` and `thread` are aliases, but give different outputs.

Example: Setting current processor

The following command sets the current processor to number 2.

```
core 2
```

Related information

[Execution control](#) on page 31

[Operating System](#) on page 35

[Expressions in the Arm Debugger](#) on page 15

2.3.23 `define`

Derives new user-defined commands from existing commands.

User-defined commands accept arguments separated by whitespace.

Syntax

```
define {cmd}  
...  
end
```

Parameters

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. You can use arguments by using `$arg0...$argn`, or `$argv` for all arguments.



Existing built in commands cannot be redefined.

Example: Define `add-args` command to print sum of first 3 arguments

```
define add-args
```

```
print $arg0+$arg1+$arg2
end
```

Example: Define echo-all command to echo all arguments

```
define echo-all
    echo $argv
end
```

Related information

[Scripts](#) on page 33

2.3.24 delete breakpoints

Deletes one or more breakpoints or watchpoints.

Syntax

```
delete [breakpoints] {number}...
```

Alias:

```
d {number...}
```

Parameters

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. Multiple numbers must be separated by spaces.



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: Deleting 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 breakpoints $      # Delete breakpoint whose number is in the
                           # most recently created debugger variable
```

Related information

[set breakpoint](#) on page 181

[disable breakpoints](#) on page 86

[info breakpoints](#), [info watchpoints](#) on page 111

[info capabilities](#) on page 112

[Breakpoints and watchpoints](#) on page 29

2.3.25 delete memory

Deletes one or more user-defined memory regions.

Syntax

```
delete memory {number}...
```

Alias:

```
d mem {number}...
```

Parameters

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: Deleting memory regions

```
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
```

Related information

[Memory group](#) on page 39

2.3.26 directory, set directories

Defines 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.

The default directories for searching are:

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

Syntax

```
directory [path...]
set directories [path...]
```

Alias:

```
dir
```

Parameters

path

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

Multiple directories can be specified but must be separated with either:

- a space
- a colon (Unix)
- a semi-colon (Windows).

Example: Adding directories to the search list

```
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
```

Related information

[Files](#) on page 36

[Set](#) on page 46

2.3.27 disable breakpoints

Disables one or more breakpoints or watchpoints.

Syntax

```
disable [breakpoints] [number...]
```

Alias:

```
dis [number...]
```

Parameters

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. Multiple numbers must be separated by spaces.



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.

Example: Disabling breakpoints

```
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
```

Related information

[set breakpoint](#) on page 181

[delete breakpoints](#) on page 84

[info breakpoints](#), [info watchpoints](#) on page 111

[info capabilities](#) on page 112

[Breakpoints and watchpoints](#) on page 29

2.3.28 disable memory

Disables one or more user-defined memory regions.

Syntax

```
disable memory {number}...
```

Alias:

```
dis mem {number}...
```

Parameters

number

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

Example: Disabling 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
```

Related information

[Memory group](#) on page 39

2.3.29 disassemble

Displays the disassembly for the function surrounding a specific address or the disassembly for a specific address range.

Syntax

```
disassemble [address [address | +<size>]]
```

Parameters

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: Displaying disassembly

```
disassemble          # Display disassembly from current program counter
disassemble 0xC0040AC0 # Display disassembly from 0xC0040AC0
disassemble 0x8140 0x8157 # Display disassembly for address range 0x8140-0x8157
disassemble 0x8140 +0x18 # Display disassembly for address range 0x8140-0x8157
```

Related information

[Memory group](#) on page 39

2.3.30 discard-symbol-file

Discards debug information relating to a specific file.

Syntax

```
discard-symbol-file {filename}
```

Alias:

```
disc {filename}
```

Parameters

filename

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



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: Discarding symbols relating to a 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
```

2.3.31 document

Adds integrated help for a new user-defined command.

Syntax

```
document {cmd}
...
end
```

Parameters

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.

Example: Documenting a new command

The following code provides 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
```

Related information

[Scripts](#) on page 33

2.3.32 down

Moves and displays 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]
```

Alias:

```
do [offset]
```

Parameters

offset

Specifies a frame offset from the current frame pointer in the call stack. If no `offset` is specified then the default is 1.

Example: Moving the pointer down the call stack and display information

```
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
```

Related information

[Call stack](#) on page 34

2.3.33 down-silently

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]
```

Alias:

```
down-s [offset]
```

Parameters

offset

Specifies a frame offset from the current frame pointer in the call stack. If no `offset` is specified then the default is 1.

Example: Moving the pointer down the call stack

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

Related information

[Call stack](#) on page 34

2.3.34 dump

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

Syntax

```
dump [format] memory {filename} [-r] {start_address} {end_address | +<size>}
dump [format] value {filename} [-r] {expression}
```

Parameters

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 to write to. Specify `-r` to overwrite an existing file.

-r

Use this option with `filename` to overwrite an existing 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: Write content of memory 0x8000-0x8FFF to binary file myFile.bin

```
dump memory myFile.bin 0x8000 0x8FFF
```

Example: Write contents of myArray to Motorola 32-bit file myFile.m32

```
dump srec value myFile.m32 &myArray
```

Related information

[Memory group](#) on page 39

2.3.35 echo

Displays only textual strings.

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}
```

Parameters**string**

Specifies a string of characters.

Example: Displaying text strings

```
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
```

Related information

[Display](#) on page 43

[Expressions in the Arm Debugger](#) on page 15

2.3.36 enable breakpoints

Enables one or more breakpoints or watchpoints by number.

Syntax

```
enable [breakpoints] [number...]
```

Alias:

```
en [number...]
```

Parameters

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.

Example: Enabling breakpoints or watchpoints

```
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
```

2.3.37 enable memory

Enables one or more user-defined memory regions.

Syntax

```
enable memory {number}...
```

Alias:

```
en m {number}...
```

Parameters

number

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

Example: Enabling user-defined memory regions

```
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
```

Related information

[Memory group](#) on page 39

2.3.38 end

Terminates conditional blocks when using the `define`, `if`, and `while` commands.

Syntax

```
end
```

Parameters

None.

Example: Defining a while loop containing commands to conditionally execute

The following code assumes `myVar` is a variable in the application code.

```
while myVar<10
  step
  wait
  x
  set myVar++
end
```

Related information

[Scripts](#) on page 33

2.3.39 exit

`exit` is an alias for [quit](#).

2.3.40 file, symbol-file

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. The debug information is loaded when required by the debugger.

If you want to append debug information instead of replacing it, you can use the `add-symbol-file` command.



Note

This command does not set the PC register.

Syntax

```
file [filename] [offset] [-s section address]...  
symbol-file [filename] [offset] [-s section address]...
```

Parameters

filename

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

offset

Specifies the offset that is added to all addresses in 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 loaded the specified library then the request is pended until the library is loaded and the offset can be determined.

-s

For relocatable objects, this specifies the address to which a section was relocated.

section

Specifies the name of the relocated section.

address

Specifies the address of the section. This can be either an address or an expression that evaluates to an address. You can also specify the address space.

Example: Loading debug information

```
file "myFile.axf"           # Load debug information on demand
file "images\myFile.axf"   # Load debug information on demand
file                       # Discard debug information
file "myFile.axf" -s .text 0x1000 -s .data 0x2000
                           # Load debug information on demand with
                           # section .text relocated to 0x1000 and
                           # section .data relocated to 0x2000
file "vmlinux" N:0          # Load debug information for the AArch32
                           # non-secure address space, with zero offset
file "vmlinux" EL2N:0x4080000000 # Load debug information for the AArch64 EL2N
                           # address space with offset 0x4080000000
```

Related information

[Files](#) on page 36

2.3.41 finish

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

Syntax

```
finish [n]
```

Alias:

```
fin [n]
```

Parameters

n

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

Example: Continue running a device

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

2.3.42 flash erase-device

Erases the memory on a specified flash device.

Syntax

```
flash erase-device {device}
```


Parameters

device

Specifies the name of a device. You can find the name of a flash device using the `info flash` command, for example:

```
info flash
LPC55S69JBD100:cm33_core0-0 [High Efficiency Arm Cortex-M33-based
  Microcontroller]
regions: 0x0-0x97FFF
parameters: address: 0x00000000
            algorithm: C:/Arm/Packs/NXP/LPC55S69_DFP/13.0.0/arm/LPC55XX_640.FLM
            coreName: Cortex-M33_0
            deviceName: LPC55S69JBD100:cm33_core0
```

Example: Erasing flash memory

The following command erases the memory on the LPC55S69JBD100:cm33_core0-0 flash device.

```
flash erase-device LPC55S69JBD100:cm33_core0-0
```

Related information

[flash erase-image-sectors](#) on page 97

[info flash](#) on page 115

2.3.43 flash erase-image-sectors

Erases all sectors of flash memory in the specified image. The image can be across multiple flash devices.

Syntax

```
flash erase-image-sectors {filename}
```

Parameters

filename

Specifies the image file, and includes the full or relative path to the file.

Restrictions

An image must be a valid `.axf` file.

Example: Erase all sectors of flash memory

The following command Erases all sectors of flash memory used by `primes.axf`.

```
flash erase-image-sectors primes.axf
```

Related information

[flash erase-device](#) on page 96

[info flash](#) on page 115

2.3.44 flash load

Loads sections from an image into one or more flash devices.



Note

To use this command you must check that flash device support is available for your target. If this support is not available, you must write your own flash algorithm for this command to work. For details on how to write your own flash algorithm in Arm® Development Studio, see [File-based Flash Programming in Arm Development Studio](#). In Arm Development Studio, the <armds_installation_directory>/examples/Bare-metal_examples_Armv7.zip/flash_algo-STM32F10x directory provides an example of how to write a flash algorithm.

Syntax

```
flash load {filename} [device [:parameter=<value>]...]...
```

Parameters

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: Loading section from an image onto a flash device

```
flash load "foo.axf"           # Loads the file to flash
flash load "foo.axf" MainFlash:ramAddress=0x20000100,ramSize=0xFF00
                                # Load the file to flash and override parameters
```

Related information

[flash load-multiple](#) on page 98

[flash](#) on page 51

2.3.45 flash load-multiple

Load multiple images on to your target.



To use this command you must check that flash device support is available for your target. If this support is not available, you must write your own flash algorithm for this command to work. For details on how to do this in Arm® Development Studio, see [File-based Flash Programming in Arm Development Studio](#). In Arm Development Studio, the <armds_installation_directory>/examples/Bare-metal_examples_Armv7.zip/flash_algo-STM32F10x directory provides an example of how to write a flash algorithm.

Syntax

```
flash load-multiple {"image"}[device_parameters] "another_image"[device_parameters]
```

Parameters

image and another_image

Specify the absolute or relative path to a valid .axf image file.

device_parameters

Specifies optional device parameter information. If you do not provide device parameter information, the image is loaded on to all flash devices on the target, using the default values for each device.

device_parameters uses the following syntax:

```
@device_name:parameter1=<value>,parameter2=<value>|  
@device_name:parameter3=<value>
```

The device parameters vary between different targets. Use `info flash` to find out which device parameters are available on your target.



You must not have any spaces between an image and its associated device parameters. Spaces are used to indicate a new image.

Restrictions

- You must specify one or more images when you use this command.
- Image paths can be relative or absolute.
- If there is an overlap between the memory address ranges that are specified in the images, an error is reported in the console.

Operation

When loading multiple images on to your target, before it writes the images to flash memory, `flash load-multiple` checks that there are no overlaps in the memory address ranges that are specified in the image files. Arm recommends using this option instead of running the `flash load` command multiple times, to prevent data corruption. With the `flash load` command, the debugger does not check for memory overlap and you might accidentally overwrite the existing data.

The following example shows the operation of `flash load-multiple` when loading multiple images onto the target:

```
flash load-multiple "image1.axf"@MainFlash "image2.axf"@OtherFlash:ramAddress=0x2  
image3.axf
```

First, the debugger checks that there are no overlaps between the memory address ranges that are specified in each image. If there is an overlap, an error is reported in the console and the command finishes without loading the images.

If there is no overlap, this command does the following:

- Loads `image1.axf` on to the `MainFlash` device only, using the default values for this device.
- Loads `image2.axf` on to the `OtherFlash` device only, starting at RAM address `0x2`.
- Loads `image3.axf` on to all available devices, using the default values for each device.

The following example shows the operation of `flash load-multiple` when loading multiple images onto the target and using device parameters to override default values:

```
flash load-multiple image4.axf image5.axf@Device2|Device3  
image6.axf@Device4:size=512,type=2
```

If there are no overlaps in the memory address ranges, this command does the following:

- Loads `image4.axf` on to all available devices, using the default values for each device.
- Loads `image5.axf` on to `Device2` and `Device3` only, using the default values for both devices.
- Loads `image6.axf` on to `Device4` only, and overrides the default values for `size` and `type`.

Related information

[flash load](#) on page 98

[info flash](#) on page 115

2.3.46 frame

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]
```

Alias:

```
f [number]
```

Parameters

number

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

Example: Displaying information for stack frames

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

Related information

[Call stack](#) on page 34

2.3.47 gcs print, info gcs

Displays a table of *Guarded Control Stack* (GCS) entries.

Syntax

```
gcs print [count] [address]
info gcs [count] [address]
```

Alias:

```
gcs p [count] [address]
i gcs [count] [address]
```

Parameters

count

Specifies the number of records to display. If not specified, five records are displayed.

address

Specifies an expression that evaluates to an address. In normal use, the evaluated address should be an entry in the GCS.

If you do not specify `address`, the GCS entries in the current EL are displayed. `GCSPR_ELn` points to these entries. In this case, the register and address used are printed in the output, above the table of entries.

Outputs

The output is a table of GCS entries. For example:

Record	Type	Field	Value
#0	Cap (Valid)	-	0x0000000000000001
#1	Exception	LR	0x0000000000000120
		SPSR	0x00000000ABCD1234
		ELR	0x0000000000000110
#2	Procedure Call	LR	0x0000000000000100
#3	Procedure Call	LR	0x0000000000000200
#4	Procedure Call	LR	0x0000000000000300

In the table:

- Record #0 represents the bottom of the GCS stack, that is the last value added.
- Type indicates the GCS entry type.
- Field and value show the data items associated with each GCS entry.
- The exception entry header 0x0000000000000009 is omitted from the table

Restrictions

If you use both parameters, you must specify `count` before `address`.

Example: Displaying GCS entries

```

info gcs                               # Displays 5 records from the currently active GCS
gcs print 10                           # Displays 10 records from the currently active GCS
gcs print EL1N:0x80000000              # Displays 5 records from the GCS at address
                                       # 0x80000000 in the EL1N address space
info gcs 8 EL3:0x10000000              # Displays 8 records from the GCS at address
                                       # 0x10000000 in the EL3 address space
gcs print $GCSPR_EL0                   # Displays 5 records from the GCS at the address
                                       # held in the GCSPR_EL0 register
gcs print gcs_stack                     # Displays 5 records from the GCS at the address
                                       # held in the gcs_stack symbol

```

2.3.48 handle

Controls the handler settings for one or more signals or exceptions. The default handler settings depend 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 handle](#), [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}...
```

Alias:

```
ha [name]...{keyword}...
```

Parameters

name

Specifies the signal or processor exception name.

keyword

Specifies the following keywords:

noprint

Disables the print property so the occurrence of an event is not indicated at all. Using the `noprint` keyword implies the properties of the `nostop` keyword as well.

nostop

Disables the stop property so the occurrence of an event does not stop execution.

print

Enables the print property. The debugger prints a message and continues execution when the event occurs. When using `gdbserver` the debugger can only print if `stop` is enabled.

stop

Enables the stop and print properties. The debugger stops execution and prints a message when the event occurs. Using the `stop` keyword implies the properties of the `print` keyword as well.

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

Example: Handler settings

```
handle SVC stop      # When an SVC exception occurs, stop execution and
                     # print a message.
handle IRQ print     # When an IRQ exception occurs, print a message, but
                     # continue execution.
handle IRQ noprint   # When an IRQ exception occurs, do not print a message.
handle noprint nostop # Do not stop execution at any event and do not print
```

```
# a message.
```

Related information

[Execution control](#) on page 31

[info handle](#), [info signals](#) on page 117

2.3.49 hbreak

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.

Syntax

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

Alias:

```
hb [-d] [-p] [[filename:]location | *<address>] [[thread|core] <number>...] [vmid
<vmid>] [context <contextid>] [if <expression>]
```

Parameters

-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 <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.

context <contextid>

Specifies the context ID to apply the breakpoint to. This can be either an integer or an expression that evaluates to an integer. You can only use the context parameter if your hardware supports it and your application makes use of the CONTEXTIDR register.

if <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.

Operation

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 is processor limited. If you run out of hardware breakpoints, then delete or disable one that you no longer use.



Note

Breakpoints that are set in a shared object or kernel module become pending when the shared object or kernel module is not loaded.

You can use **info breakpoints capabilities** to display a list of parameters that you can use with breakpoint commands for the current connection.

Example: Setting a hardware execution breakpoint

```
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 at address of  
                             # function1() when x>0  
hbreak context 257 0x80000000 # Set conditional breakpoint at address 0x80000000  
                             # that stops when CONTEXTIDR=257
```

Related information

[break](#) on page 70

[tbreak](#) on page 232

[thbreak](#) on page 233

[resolve](#) on page 172

[clear](#) on page 78

[Breakpoints and watchpoints](#) on page 29

2.3.50 help

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

Syntax

```
help [command | group]
```

Alias:

```
h [command | group]
```

Parameters

command

Specifies an individual command.

group

Specifies a group name for specific debugging tasks:

group_breakpoints

Displays the breakpoint and watchpoint commands.

group_cache

Displays the cache 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_flash

Displays the flash commands.

group_info

Displays the program information commands.

group_log

Displays the message logging commands.

group_memory

Displays the commands that interact with memory.

group_mmu

Displays the MMU commands.

group_mpu

Displays the MPU commands.

group_os

Displays the operating system commands.

group_registers

Displays the register commands.

group_running

Displays the target execution and stepping group.

group_scripts

Displays the commands for use in script files.

group_set

Displays the set commands for debugger settings.

group_show

Displays the show commands for debugger settings.

group_stack, stack

Displays the call stack commands.

group_support

Displays the supporting commands.

Example: Displaying help information

```
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
```

2.3.51 if

Allows you to write scripts that conditionally execute debugger commands.

Syntax

```
if {condition}  
...  
else  
...  
end
```

Parameters

<condition>

Specifies a conditional expression. Follow the if statement with one or more debugger commands that execute when the expression evaluates to true.

Operation

Enter each debugger command on a new line. Use the optional `else` statement to specify debugger commands that only execute when <condition> evaluates to false.

Terminate the `if` command with the `end` command.

Example: Conditional execution

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

Related information

[Scripts](#) on page 33

2.3.52 ignore

Sets the ignore counter for a breakpoint or watchpoint condition.

Syntax

```
ignore {number} {count}
```

Parameters

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: Ignoring breakpoints

```
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
```

Related information

[Breakpoints and watchpoints](#) on page 29

2.3.53 info address

Displays the location of a symbol.

Syntax

```
info address {symbol}
```

Alias:

```
i ad {symbol}
```

Parameters

symbol

Specifies the symbol.

Example: Displaying the location of a symbol

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

2.3.54 info all-registers

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

Syntax

```
info all-registers [group] [/ [<grouping>] [<separator>]]
```

Alias:

```
i all-registers [group] [/ [<grouping>] [<separator>]]
```

Parameters

group

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

<grouping>

Specifies the number of digits in a group between separators when displaying values held in registers. Must be an integer with a value of 2, 3, 4, or 8.

If `<grouping>` is not specified and `<separator>` is specified, grouping defaults to 4.

<separator>

Specifies a separator character used when displaying values held in registers. Must be one of the following:

- `a` specifies a separator character of `'`
- `b` specifies a separator character of ```
- `u` specifies a separator character of `_`

If `<grouping>` is not specified and `<spacing>` is specified, the separator character defaults to `'`. If `<grouping>` and `<spacing>` are both specified, `<grouping>` must be specified first.

Operation

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 callee 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.

Optionally, you can make large values held in registers easier to read by displaying separators in the values.

Example: Displaying information for all registers

```
info all-registers      # Display info for all registers
info all-registers USR  # Display info for all user mode registers
info all-registers /3u  # Display info for all registers with an underscore
                       # separator after every third digit
```

Related information

[Registers](#) on page 40

[Information](#) on page 43

2.3.55 info breakpoints, info watchpoints

Displays information about the status of all breakpoints and watchpoints.

Syntax

```
info breakpoints
info watchpoints
```

Alias:

```
i b
i w
```

Parameters

None.

Operation

These commands set 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.

Example: Displaying status for all breakpoints and watchpoints

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

Related information

[set breakpoint](#) on page 181

[disable breakpoints](#) on page 86

[delete breakpoints](#) on page 84

[info capabilities](#) on page 112

[info breakpoints capabilities](#) on page 112

[info watchpoints capabilities](#) on page 136

[Breakpoints and watchpoints](#) on page 29

[Information](#) on page 43

2.3.56 info breakpoints capabilities

Displays a list of properties that you can use with breakpoint commands for the current connection. You can modify the properties of an existing breakpoint using [break-set-property](#).

Syntax

```
info breakpoints capabilities
```

Alias:

```
i b c
```

Parameters

None.

Example: Displaying list of available breakpoint properties

```
info breakpoints capabilities    # Display list of available breakpoint  
                                # properties for the current connection
```

Related information

[set breakpoint](#) on page 181

[disable breakpoints](#) on page 86

[delete breakpoints](#) on page 84

[info breakpoints](#), [info watchpoints](#) on page 111

[info capabilities](#) on page 112

[Breakpoints and watchpoints](#) on page 29

2.3.57 info capabilities

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
```

Alias:

```
i cap
```


Parameters

None.

Example: Displaying target device capabilities

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

Related information

[set breakpoint](#) on page 181

[disable breakpoints](#) on page 86

[delete breakpoints](#) on page 84

[info breakpoints](#), [info watchpoints](#) on page 111

[info breakpoints capabilities](#) on page 112

[info watchpoints capabilities](#) on page 136

[Breakpoints and watchpoints](#) on page 29

[Information](#) on page 43

2.3.58 info classes

Displays C++ class names.

Syntax

```
info classes [expression]
```

Alias:

```
i classes [expression]
```

Parameters

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: Displaying C++ class names

```
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)
```

Related information

[Information](#) on page 43

2.3.59 info cores

Displays information about the running 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
```

Alias:

```
i cores
```

Parameters

None.

Example: Displaying all processors

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

Related information

[Information](#) on page 43

2.3.60 info files

Displays information about the loaded image and symbols.

Syntax

```
info files
```

Alias:

```
i files
```

Parameters

None.

Example: Displaying information for loaded image and symbols

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

Related information

[Information](#) on page 43

[info target](#) on page 134

2.3.61 info flash

Displays information about the flash devices on the current target.

Syntax

```
info flash
```

Alias:

```
i flash
```

Parameters

None.

Operation

To use this command you must check that flash device support is available for your target. If this support is not available, you must write your own flash algorithm for this command to work. For details on how to do this in Arm® Development Studio, see [File-based Flash Programming in Arm Development Studio](#). In Arm Development Studio, the <armds_installation_directory>/examples/Bare-metal_examples_Armv7.zip/flash_algo-STM32F10x directory provides an example of how to write a flash algorithm.

Example: Display information for flash devices

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

Related information

[flash](#) on page 51

2.3.62 info frame

Displays stack frame information at the selected position.

- 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]
```

Alias:

```
i f [number]
```

Parameters

number

Specifies the frame number.

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

Example: Displaying stack frame information

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

Related information

[Call stack](#) on page 34

2.3.63 info functions

Displays the name and data types for all functions.

Syntax

```
info functions [expression]
```

Alias:

```
i fun [expression]
```

Parameters

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: Displaying the name and data types for all 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)

```

2.3.64 info gcs

`info gcs` is an alias for `gcs print`.

2.3.65 info handle, info signals

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 handle [name]
info signals [name]

```

Alias:

```

i handle [name]
i signals [name]

```

Parameters

name

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

Example: Displaying information about signals

```

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

```

Related information

[Execution control](#) on page 31

[handle](#) on page 102

[Information](#) on page 43

2.3.66 info inst-sets

Displays the available instruction sets.

Syntax

```
info inst-sets
```

Alias:

```
i inst-sets
```

Parameters

None.

Example: Displaying available instruction sets

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

Related information

[Information](#) on page 43

2.3.67 info locals

Displays all local variables for the current stack frame.

Syntax

```
info locals
```

Alias:

```
i locals
```

Parameters

None.

Example: Displaying local variables for current stack frame

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

Related information

[Information](#) on page 43

2.3.68 info members

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]
```

Alias:

```
i memb [expression]
```

Parameters

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.

Restrictions

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: Displaying class member variables accessible to a stack frame

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

Related information

[Information](#) on page 43

2.3.69 info memory

Displays the currently defined memory regions.

This command also shows the currently defined attributes for the memory regions. When you specify an address as an argument to a command, you can also specify the attributes defined for the memory region if needed.

Syntax

```
info memory
```

Alias:

```
i mem
```

Parameters

None.

Operation

You can define new memory regions using the `memory` command. To discover the additional set of attributes applicable for a region of address space, you can use the `info memory-parameters` command.

Example: info memory output from an Armv9-A target

Num	Enb	Low Addr	High Addr	Attributes
Description				
1:	y	SP:0x0000000000000000	SP:0xFFFFFFFFFFFFFFFF	rw, nocache, verify
Memory accessed using secure world physical addresses				
2:	y	S:0x00000000	S:0xFFFFFFFF	rw, nocache, verify
Memory accessed using secure world addresses				
3:	y	RTP:0x0000000000000000	RTP:0xFFFFFFFFFFFFFFFF	rw, nocache, verify
Memory accessed using root world physical addresses				
4:	y	RLP:0x0000000000000000	RLP:0xFFFFFFFFFFFFFFFF	rw, nocache, verify
Memory accessed using realm world physical addresses				
5:	y	RL:0x00000000	RL:0xFFFFFFFF	rw, nocache, verify
Memory accessed using realm world addresses				
6:	y	NP:0x0000000000000000	NP:0xFFFFFFFFFFFFFFFF	rw, nocache, verify
Memory accessed using normal world physical addresses				
7:	y	N:0x00000000	N:0xFFFFFFFF	rw, nocache, verify
Memory accessed using normal world addresses				
8:	y	EL3:0x0000000000000000	EL3:0xFFFFFFFFFFFFFFFF	rw, nocache, verify
Memory accessed using EL3 addresses				
9:	y	EL2S:0x0000000000000000	EL2S:0xFFFFFFFFFFFFFFFF	rw, nocache, verify
Memory accessed using secure world EL2 addresses				
10:	y	EL2RL:0x0000000000000000	EL2RL:0xFFFFFFFFFFFFFFFF	rw, nocache, verify
Memory accessed using EL2 realm world addresses				
11:	y	EL2N:0x0000000000000000	EL2N:0xFFFFFFFFFFFFFFFF	rw, nocache, verify
Memory accessed using normal world EL2 addresses				
12:	y	EL1S:0x0000000000000000	EL1S:0xFFFFFFFFFFFFFFFF	rw, nocache, verify
Memory accessed using EL1 secure world addresses				
13:	y	EL1RL:0x0000000000000000	EL1RL:0xFFFFFFFFFFFFFFFF	rw, nocache, verify
Memory accessed using EL1 realm world addresses				
14:	y	EL1N:0x0000000000000000	EL1N:0xFFFFFFFFFFFFFFFF	rw, nocache, verify
Memory accessed using EL1 normal world addresses				
15:	y	APB:0x00000000	APB:0xFFFFFFFF	rw, nobp, nohbp, nocache
APB bus accessed via AP_1				
16:	y	AHB:0x00000000	AHB:0xFFFFFFFF	rw, nobp, nohbp, nocache
AHB bus accessed via AP_0				

Example: info memory output from an Armv8-A target

Num	Enb	Low Addr	High Addr	Attributes
Description				
1:	y	SP:0x0000000000000000	SP:0xFFFFFFFFFFFFFFFF	rw, nocache, verify
Memory accessed using secure world physical addresses				


```

2: y S:0x00000000 S:0xFFFFFFFF rw, nocache, verify
    Memory accessed using secure world addresses
3: y NP:0x0000000000000000 NP:0xFFFFFFFFFFFFFFFF rw, nocache, verify
    Memory accessed using normal world physical addresses
4: y N:0x00000000 N:0xFFFFFFFF rw, nocache, verify
    Memory accessed using normal world addresses
5: y H:0x00000000 H:0xFFFFFFFF rw, nocache, verify
    Memory accessed via hypervisor address
6: y EL3:0x0000000000000000 EL3:0xFFFFFFFFFFFFFFFF rw, nocache, verify
    Memory accessed using EL3 addresses
7: y EL2N:0x0000000000000000 EL2N:0xFFFFFFFFFFFFFFFF rw, nocache, verify
    Memory accessed using normal world EL2 addresses
8: y EL1S:0x0000000000000000 EL1S:0xFFFFFFFFFFFFFFFF rw, nocache, verify
    Memory accessed using EL1 secure world addresses
9: y EL1N:0x0000000000000000 EL1N:0xFFFFFFFFFFFFFFFF rw, nocache, verify
    Memory accessed using EL1 normal world addresses
10: y AXI:0x0000000000000000 AXI:0xFFFFFFFFFFFFFFFF rw, nobp, nohbp, nocache,
    noverify AXI bus accessed via AP 1 (CSMEMAP_1)
ACE=0, CACHE=0, DOMAIN=3, MODE=0, PROT=3 AXI bus accessed via AP 1 (CSMEMAP_1)
11: y APB:0x00000000 APB:0xFFFFFFFF rw, nobp, nohbp, nocache,
    noverify APB bus accessed via AP 0 (CSMEMAP_0)
12: y AHB 1:0x00000000 AHB 1:0xFFFFFFFF rw, nobp, nohbp, nocache,
    noverify AHB bus accessed via AP 1 (CSMEMAP_3)
HPROT=67 AHB bus accessed via AP 1 (CSMEMAP_3)
13: y AHB 0:0x00000000 AHB 0:0xFFFFFFFF rw, nobp, nohbp, nocache,
    noverify AHB bus accessed via AP 0 (CSMEMAP_2)
HPROT=67 AHB bus accessed via AP 0 (CSMEMAP_2)

```

Example: info memory output from an Armv7-A target

Num	Enb	Low Addr	High Addr	Attributes	
1:	y	SP:0x0000000000	SP:0xFFFFFFFF	rw, nocache, verify	Memory
		accessed using secure world physical addresses			
2:	y	S:0x00000000	S:0xFFFFFFFF	rw, nocache, verify	Memory
		accessed using secure world addresses			
3:	y	NP:0x0000000000	NP:0xFFFFFFFF	rw, nocache, verify	Memory
		accessed using normal world physical addresses			
4:	y	N:0x00000000	N:0xFFFFFFFF	rw, nocache, verify	Memory
		accessed using normal world addresses			
5:	y	H:0x00000000	H:0xFFFFFFFF	rw, nocache, verify	Memory
		accessed via hypervisor address			
6:	y	APB:0x00000000	APB:0xFFFFFFFF	rw, nobp, nohbp, nocache, noverify	APB bus
		accessed via AP 1			
7:	y	AHB:0x00000000	AHB:0xFFFFFFFF	rw, nobp, nohbp, nocache, noverify	AHB bus
		accessed via AP 0			
8:	y	S:0x80000000	S:0x80001DCB	cache	[EXEC]C:
		\Arm_DS_Workspace\smp_primes_A15x2-CoreTile\primes.axf			
9:	y	S:0x80001DCC	S:0x80001E33	cache	[EXEC]C:
		\Arm_DS_Workspace\smp_primes_A15x2-CoreTile\primes.axf			
10:	y	S:0x80001E34	S:0x8000229F	cache	[EXEC]C:
		\Arm_DS_Workspace\smp_primes_A15x2-CoreTile\primes.axf			
11:	y	S:0x800022A0	S:0x8000429F	cache	
		[ARM_LIB_HEAP]C:\Arm_DS_Workspace\smp_primes_A15x2-CoreTile\primes.axf			
12:	y	S:0x800042A0	S:0x8000829F	cache	
		[ARM_LIB_STACK]C:\Arm_DS_Workspace\smp_primes_A15x2-CoreTile\primes.axf			
13:	y	S:0x800082A0	S:0x8000869F	cache	
		[IRQ_STACKS]C:\Arm_DS_Workspace\smp_primes_A15x2-CoreTile\primes.axf			
14:	y	S:0x80500000	S:0x805FFFFF	cache	
		[PAGETABLES]C:\Arm_DS_Workspace\smp_primes_A15x2-CoreTile\primes.axf			

Related information

[Memory group](#) on page 39

[Information](#) on page 43

2.3.70 info memory-parameters

Displays the memory parameters applicable to an address space.

Syntax

```
info memory-parameters
```

Alias:

```
i mem-params
```

Parameters

None.

Operation

When using the debugger to interact with target memory, you can specify the memory address using an expression. The debugger also allows other aspects of the memory operation to be controlled using extra parameters in the expression. Different address spaces support different parameters. You can use the `info memory-parameters` command to list the parameters applicable to an [address space](#).

Example: Displaying memory parameters for an address space

The following is an example of the output:

Address Space	Parameter	Description
N:	width	Specifies the access width used to perform the access, note that this is independent from the total amount of data read
	verify	Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written
	use_image	Fetch data reads from loaded images rather than the target
NP:	width	Specifies the access width used to perform the access, note that this is independent from the total amount of data read
	verify	Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written
	use_image	Fetch data reads from loaded images rather than the target
	stages	For a physical access specifies the number of MMU stages to leave enabled
S:	width	Specifies the access width used to perform the access, note that this is independent from the total amount of data read
	verify	Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written
	use_image	Fetch data reads from loaded images rather than the target
SP:	width	Specifies the access width used to perform the

	verify	access, note that this is independent from the total amount of data read Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written
	use_image	Fetch data reads from loaded images rather than the target
	stages	For a physical access specifies the number of MMU stages to leave enabled
<hr/>		
AHB_0:	width	Specifies the access width used to perform the access, note that this is independent from the total amount of data read
	verify	Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written
	use_image	Fetch data reads from loaded images rather than the target
	HPROT	Specify the HPROT (15 bits), specifying the protection bits in the AHB CSW register
<hr/>		
AHB_1:	width	Specifies the access width used to perform the access, note that this is independent from the total amount of data read
	verify	Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written
	use_image	Fetch data reads from loaded images rather than the target
	HPROT	Specify the HPROT (15 bits), specifying the protection bits in the AHB CSW register
<hr/>		
APB4	width	Specifies the access width used to perform the access, note that this is independent from the total amount of data read
	verify	Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written
	use_image	Fetch data reads from loaded images rather than the target
	PROT	Specify the PROT (15 bits), specifying the protection bits in the APB4 CSW register
	NSE	Specify the NSE (1 bit, default value 1, controls access to ROOT/REALM physical address spaces)
<hr/>		
AXI5	width	Specifies the access width used to perform the access, note that this is independent from the total amount of data read
	verify	Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written
	use_image	Fetch data reads from loaded images rather than the target
	DOMAIN	Specify the DOMAIN (2 bits, default value 3, the shareable transaction encoding for ACE)
	MODE	Specify the MODE (4 bits, default value 0, the mode of operation normal or barrier)
	NSE	Specify the NSE (1 bit, default value 1, controls access to ROOT/REALM physical address spaces)
	PROT	Specify the PROT (3 bits, default value 3, the protection encoding as AMBA AXI protocol describes)
	CACHE	Specify the CACHE (4 bits, default value 0, the cache encoding as AMBA AXI protocol describes)
<hr/>		
EL1N:	width	Specifies the access width used to perform the access, note that this is independent from the total amount of data read
	verify	Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written

	use_image	Fetch data reads from loaded images rather than the target
	view = {MemTag}	View data for a feature which is associated with this memory space. Only accessible when the feature is implemented.

EL1S:	width	Specifies the access width used to perform the access, note that this is independent from the total amount of data read
	verify	Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written
	use_image	Fetch data reads from loaded images rather than the target
	view = {MemTag}	View data for a feature which is associated with this memory space. Only accessible when the feature is implemented.

EL2N:	width	Specifies the access width used to perform the access, note that this is independent from the total amount of data read
	verify	Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written
	use_image	Fetch data reads from loaded images rather than the target
	view = {MemTag}	View data for a feature which is associated with this memory space. Only accessible when the feature is implemented.

EL3:	width	Specifies the access width used to perform the access, note that this is independent from the total amount of data read
	verify	Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written
	use_image	Fetch data reads from loaded images rather than the target
	view = {MemTag}	View data for a feature which is associated with this memory space. Only accessible when the feature is implemented.

H:	width	Specifies the access width used to perform the access, note that this is independent from the total amount of data read
	verify	Controls whether or not a write operation must verify the value written by reading the value back and comparing it to the value written
	use_image	Fetch data reads from loaded images rather than the target

Related information

[Memory group](#) on page 39

[Information](#) on page 43

[set variable, set](#) on page 204

[Address space prefixes](#) on page 24

[Memory parameters](#) on page 26

[About address spaces](#)

[About debugging caches](#)

2.3.71 info os

Displays the current state of the Operating System (OS) support. If OS support is enabled, also lists all available OS data tables. To print the contents of a data table, pass its name as an argument.

Syntax

```
info os [data-table]
```

Alias:

```
i os [data-table]
```

Parameters

data-table

Specifies the data table name.

Restrictions

A connection must be established with your target before you can use this command. You can use the `set os` command to control operating system support in the debugger.

Example: Displaying the state of OS support

```
info os          # Displays the current state of the OS support and lists all
                  # available OS data tables.
info os tasks    # Displays the contents of the 'tasks' data table, where
                  # 'tasks' is the name of an available data table.
```

Related information

[Operating System](#) on page 35

[Information](#) on page 43

2.3.72 info os-log

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.

Syntax

```
info os-log
```

Alias:

```
i os-log
```

Parameters

None.

Restrictions

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

Example: Display the OS log buffer

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

Related information

[Operating System](#) on page 35

[Information](#) on page 43

2.3.73 info os-modules

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

Syntax

```
info os-modules [-s]
```

Alias:

```
i os-modules [-s]
```

Parameters

-s

Displays the section information of the modules.

Restrictions

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

Example: Displaying OS module information

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

Related information

[Operating System](#) on page 35

[Information](#) on page 43

2.3.74 info os-version

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

Syntax

```
info os-version
```

Alias:

```
i os-version
```

Parameters

None.

Example: Displaying OS version

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

Related information

[Operating System](#) on page 35

[Information](#) on page 43

2.3.75 info overlays

Displays information about the currently loaded overlays. It shows the ID, the load address, exec address, and size for each overlay, and whether it is loaded or not.

Syntax

```
info overlays [functions]
```

Alias:

```
i overlays [functions]
```

Parameters

functions

Displays the details of functions in the overlay.

Example: Displaying information about overlays

```
info overlays      # Displays the details of overlays in the application.  
info overlays functions  # Displays the details of functions in each overlay.
```

2.3.76 info processes

Displays information about the 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
```

Alias:

```
i processes
```

Parameters

None.

Example: Displaying user space processes information

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

Related information

[Operating System](#) on page 35

[Information](#) on page 43

2.3.77 info registers

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

Syntax

```
info registers [register] [/[<grouping>][<separator>]]
```

Alias:

```
i r [register] [/[<grouping>][<separator>]]
```

Parameters

register

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

<grouping>

Specifies the number of digits in a group between separators when displaying values held in registers. Must be an integer with a value of 2, 3, 4, or 8.

If `<grouping>` is not specified and `<separator>` is specified, grouping defaults to 4.

<separator>

Specifies a separator character used when displaying values held in registers. Must be one of the following:

- `a` specifies a separator character of `'`
- `b` specifies a separator character of ```
- `u` specifies a separator character of `_`

If `<separator>` is not specified and `<grouping>` is specified, the separator character defaults to `'`. If `<separator>` and `<grouping>` are both specified, `<grouping>` must be specified first.

Operation

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 callee 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.

Optionally, you can make large values held in registers easier to read by displaying separators in the values.

Example: Displaying application level register information

```
info registers          # Display info for all application level registers
info registers pc       # Display info for PC register
info registers /3u      # Display info for all application level registers with an
                        # underscore separator after every third digit
```

Related information

[Registers](#) on page 40

[Information](#) on page 43

2.3.78 info semihosting

Displays semihosting information.

Syntax

```
info semihosting [server | clients | all]
```

Alias:

```
i semihosting [server | clients | all]
```

Parameters

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: Displaying semihosting information

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

Related information

[Information](#) on page 43

2.3.79 info sharedlibrary

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

Syntax

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

Alias:

```
i share [ /<order> ] [ /<sort_by> ] [ /<group> ]
```

Parameters

/<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.

Restrictions

- You must launch the debugger with the `--target_os` command-line option before you can use this feature. In Arm® Development Studio, this option is automatically selected when you connect to a target using `gdbserver`.
- This command is only supported for Linux application debug, for example, connections using `gdbserver`. It is not supported for Linux kernel debug, for example, connections using JTAG.

Example: Displaying shared library information

```
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
```

Related information

[Operating System](#) on page 35

[Information](#) on page 43

2.3.80 info signals

`info signals` is an alias for [info handle](#).

2.3.81 info sources

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
```

Alias:

```
i sources
```

Parameters

None.

Example: Displaying source file names

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

Related information

[Files](#) on page 36

[Information](#) on page 43

2.3.82 info stack, backtrace, where

Displays a numbered list of the calling stack frames including the function names and source line numbers.

Syntax

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

Alias:

```
i s [n | -n] [full]
bt [n | -n] [full]
```

Parameters

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.

Operation

You can use [set backtrace](#) to control the default call stack display settings.

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.

Example: Displaying call stacks

```
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
```

Related information

[Call stack](#) on page 34

[Information](#) on page 43

2.3.83 info symbol

Displays the symbol name at a specific address.

Syntax

```
info symbol {address}
```

Alias:

```
i sy {address}
```

Parameters

address

Specifies the address.

Example: Displaying symbol names

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

Related information

[Information](#) on page 43

2.3.84 info target

Displays information about the loaded image and symbols.

Syntax

```
info target
```

Alias:

```
i target
```

Parameters

None.

Example: Displaying loaded image and symbol information

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

Related information

[Information](#) on page 43

[info files](#) on page 114

2.3.85 info threads

Displays information about the available 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.

Syntax

```
info threads
```

Alias:

```
i threads
```

Parameters

None.

Restrictions

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

Example: Displaying thread information

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

Related information

[Operating System](#) on page 35

[Information](#) on page 43

2.3.86 info variables

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

Syntax

```
info variables [expression]
```

Alias:

```
i va [expression]
```

Parameters

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: Displaying 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)
```

Related information

[Information](#) on page 43

2.3.87 info watchpoints

`info watchpoints` is an alias for [info breakpoints](#).

2.3.88 info watchpoints capabilities

Displays a list of properties that you can use with watchpoint commands for the current connection. You can modify the properties of an existing watchpoint using [watch-set-property](#).

Syntax

```
info watchpoints capabilites
```

Alias:

```
i w c
```

Parameters

None.

Example: Displaying watchpoint properties

```
info watchpoints capabilities # Display list of available watchpoint
                             # properties for the current connection
```

Related information

[info breakpoints](#), [info watchpoints](#) on page 111

[info capabilities](#) on page 112

[Breakpoints and watchpoints](#) on page 29

2.3.89 inspect

inspect is an alias for [print](#).

2.3.90 interrupt, stop

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

Syntax

```
interrupt
stop
```

Parameters

None.

Example: Interrupting the application

```
interrupt          # Interrupt application.
stop               # Interrupt application.
```

Related information

[Execution control](#) on page 31

2.3.91 list

Displays lines of source code surrounding the current or specified location.

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.

Syntax

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

Alias:

```
l [[<filename>:]<location> | + | - | +<offset> | -<offset>]] [*<address>]
```

Parameters

<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.

Operation

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.

Example: Displaying source code surrounding a location

```
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
```

2.3.92 load

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

Syntax

```
load [<filename>][<offset>]
```

Parameters

<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 in the image.

Restrictions

- This command does not set the PC register.
- This command does not load debug information. To load debug information, you can use one of the following commands:
 - [add-symbol-file](#)
 - [file](#)
 - [loadfile](#)

Operation

The debugger supports and automatically recognizes the following file formats:

- 32-bit Arm ELF
- 64-bit Arm ELF
- Byte oriented hexadecimal (Verilog Memory Model)
- Intel Hex-32
- Motorola 32-bit (S-records)

Example: Loading an image

```
load "myFile.axf"           # Load image
load "images\myFile.axf"    # Load image
load myFile.axf 0x2000      # Load image with offset 0x2000
load "myV8File.axf" EL3:0x0 # Load image in the EL3 address space with offset 0x0
```

Related information

[Files](#) on page 36

[set elf load-segments-at-p_paddr](#) on page 188

2.3.93 loadfile

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.

The debug information is loaded when required by the debugger.

Syntax

```
loadfile <filename> [<offset>]
```

Alias:

```
lf <filename> [<offset>]
```

Parameters

<filename>

Specifies the image.

<offset>

Specifies the offset that is added to all addresses in the image.

Restrictions

This command does not set the PC register.

Example: Loading images

```
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 "myV8File.axf" EL3:0x0 # Load image in the EL3 address space with offset
                                # 0x0 and load debug information when required.
```

Related information

[Files](#) on page 36

2.3.94 log config

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

Syntax

```
log config <option>
```

Parameters

<option>

Specifies a predefined logging configuration or a user-defined logging configuration file:

off

Do not output runtime messages.

error

Output messages using the predefined ERROR level configuration. Reports error messages.

warn

Output messages using the predefined WARN level configuration. Reports error and warning messages.

info

Output messages using the predefined INFO level configuration. Reports errors, warnings, and other debugger messages. This is the default.

debug

Output messages using the predefined DEBUG level configuration. Reports more detailed messages than the `info` argument.

trace

Output messages using the predefined TRACE level configuration. Reports more detailed messages than the `debug` argument.

<filename>

Specifies a user-defined logging configuration file to customize the output of messages. The debugger supports log4j configuration files only.

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

Example: Displaying debug messages

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

Related information

[log](#) on page 45

[Log4j in Apache Logging Services](#)

2.3.95 log file

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

Syntax

```
log file [<filename>]
```

Parameters

<filename>

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

Example: Specifying an output file

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

Related information

[log](#) on page 45

2.3.96 memory

Defines a memory region and specifies its attributes and size.

Syntax

```
memory {start_address} {end_address | +<size>} [attributes]...
```

Alias:

```
mem {start_address} {end_address | +<size>} [attributes]...
```

Parameters

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.

unwind | nounwind

Controls whether the debugger should read from this area of memory when unwinding the stack.

By default, when unwinding the stack, the debugger accesses any area of memory marked as readable.

Operation

This command 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.

User-defined memory regions are higher-numbered, and they override lower-numbered memory regions. Use the `info memory` command to view the available memory regions.

Example: Defining a memory region

```
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)
memory 0 0xFFFFFFFF ro nobp    # Specify RO region 0-0xFFFFFFFF no breakpoints
```

Related information

[Memory group](#) on page 39

[info memory](#) on page 119

[disable memory](#) on page 87

[enable memory](#) on page 93

[delete memory](#) on page 85

2.3.97 memory auto

Resets the memory regions to the default target settings and discards all user-defined regions.

Syntax

```
memory auto
```

Alias:

```
mem auto
```

Parameters

None.

Example: Resetting default memory regions

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

Related information

[Memory group](#) on page 39

2.3.98 memory debug-cache

Controls the caching by the debugger for all memory regions. You can use `info mem` to display the caching attributes.

Syntax

```
memory debug-cache {option}
```

Alias:

```
mem debug-cache {option}
```

Parameters

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: Controlling debugger caching

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

Related information

[Memory group](#) on page 39

2.3.99 memory fill

Writes a specific pattern of bytes to memory.

Syntax

```
memory fill [verify=<flag>:]{start_address} {end_address | +<offset>} {fill_size}  
{pattern}
```

Alias:

```
mem fill [verify=<flag>:]{start_address} {end_address | +<offset>} {fill_size}  
{pattern}
```

Parameters

verify

Qualifies the address with a flag to specify whether the operation must perform a verify action or not. The values for <flag> are:

0

There is no need to verify whether the operation executed correctly.

1

The operation must verify whether it executed correctly. This is the default.

start_address

Specifies the start address for the region. This can be either an address or an expression that evaluates to an address.

For example:

```
memory fill EL1N<verify=0>:0x0 0xFFFFFFFF 4 0x12345678
```

If there is only one (anonymous) address space, then use:

```
memory fill <verify=0>:0x00xFFFFFFFF 4 0x12345678
```

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: Writing byte patterns to memory

```
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
```

Related information

[Memory group](#) on page 39

2.3.100 memory set

Writes to memory.

Syntax

```
memory set {address} {width} {expression}
memory set {address_space} [<memory_parameters>]:{address} {width} {expression}
```

Alias:

```
mem set {address} {width} {expression}
mem set {address_space} [<memory_parameters>]:{address} {width} {expression}
```

Parameters

address

Specifies an address where the first value is to be written. The address must be correctly aligned for the type of the specified expression. For example:

```
memory set 0x8000 32 0x1234
```

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.

Widths depend on the target, address region, and address alignment. In addition, some widths might not be supported. A width of 0 enables the debugger to determine the access width. The following are examples of widths that could be specified:

8 specifies 8-bit

16 specifies 16-bit

32 specifies 32-bit

64 specifies 64-bit

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.

address_space

The name of an address space (for example `N` or `EL3`) or a bus (for example `AXI` or `APB`). If the `address_space` is omitted, the current address space is used. For further details of address spaces see [Address space prefixes](#).

You can use [info memory](#) to show the size of all types of address space the debugger knows about on the target, and the memory parameters used for each address space.

memory_parameters

Allows you to set the memory bus parameters for an `address_space` that is a bus. For example:

```
memory set AXI<PROT=3,CACHE=1,DOMAIN=3,ACE=1,MODE=0,width=32,verify=0>:0x80000000
32 0x1234
```

**Note**

The memory bus `width` parameter is specified in bits and, if set, overrides the `width` parameter specified in the `memory set` command.

You can use [info memory-parameters](#) to show the available memory bus parameters for a target.

Example: Writing to memory

```
memory set 0x8000 0 "Hello"      # Writes a string to memory.
memory set 0x1008 0 0x1234        # Writes an integer to memory. This is the equivalent
                                # of *(int*)0x1008 = 0x1234.
memory set 0x1008 8 0x1234        # Writes 0x00001234 to address 0x1008 but forces the
                                # use of 4 writes of one byte each.
memory set EL1N:0x80000000 32 0x1234
                                # Writes 0x00001234 to address 0x80000000 on the EL1N
                                # address space.
memory set N<verify=1>:0x80000000 16 0x1234
                                # Writes 0x00001234 to address 0x80000000 on the
                                # Non-secure (N) address space and then verifies that
                                # the value was written correctly by reading it back.
memory set AXI<PROT=3,CACHE=1,DOMAIN=3,ACE=1,MODE=0,width=16>:0x80000000 32 0x5678
                                # Sets memory bus parameters for AXI access and writes
                                # 0x00005678 to address 0x80000000 on the AXI bus
                                # using a 16-bit access.
memory set AXI<PROT=2,NSE=1>:0x80000000 32 0x12345678
```

```

# Where the Realm Management Extension is implemented,
# selects the Realm Physical Address Space on the
# AXI-AP bus, and writes a value to an address using
# a 32-bit access.
memory set AHB<HPROT=3,width=32>:0x80000000 32 0x1234
# Sets memory bus parameters for AHB access and writes
# 0x00005678 to address 0x80000000 on the AHB bus
# using a 32-bit access.
memory set 0x80000000 16 (unsigned short)0x1234
# Writes the 16-bit value 0x1234 to address 0x80000000
# using a 16-bit access (in a little endian
# memory system).
memory set 0x80000000 8 (unsigned char)0x12
# Writes the 8-bit value 0x12 to address 0x80000000
# using an 8-bit access (in a little endian
# memory system).
memory set 0x1000 0 {(char)0x10,(char)0xFF,(char)1,(char)2,(char)3,(char)42}
# This command is equivalent to the following
# sequence of 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
# set variable *(char*)0x1005 = (char)42

```

Related information

[Memory group](#) on page 39

[Address space prefixes](#) on page 24

2.3.101 memory set_typed

Writes a list of values to memory.

Syntax

```
memory set_typed {address} {type} {expressions}
```

Alias:

```
mem set_typed {address} {type} {expressions}
```

Parameters

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.

Operation

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: Writing values to memory

```
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
```

Related information

[Memory group](#) on page 39

2.3.102 mmu dirty-state

Displays in-memory dirty state structures.

Syntax

```
mmu dirty-state {structure name} [param1=<value1> param2=<value2>...] [all|count
expression]
```

Alias:

```
mm d {structure name} [param1=<value1> param2=<value2>...] [all|count expression]
```

Parameters

structure name

The name of a dirty state structure, for example `EL1S_S2_HDBSSPROD_EL2`. You can display the available dirty state structures using [mmu list dirty-state](#).

param1=<value1> param2=<value2>...

Specifies zero or more dirty state structure parameters, each with a value used to interpret the structure. Each `param=value` pair must be separated by whitespace, but cannot contain whitespace. You can display the available dirty state structures and valid parameters using [mmu list dirty-state](#).

If you do not specify a parameter, then the structure is determined from the current target state.

all

Prints the entire dirty state structure from index 0 to the maximum configured size.

If you do not specify `count expression` or `all`, 10 entries are displayed. Entries start at, and include, the current index.

count expression

Specifies an expression, relative to the current index, that is used to determine how many entries to show. Counts can be negative to list entries before the INDEX. Counts that are greater than the specified structure size or lower than the zero index entry are truncated.

If you do not specify `count expression` or `all`, 10 entries are displayed. Entries start at, and include, the current index.

Outputs

The output from this command shows:

- The index base address, physical size, and current index entry with number of index entries.
- A table displaying the following:
 - **Entry** - The index entry number with > indicating the current index entry.
 - **IPA Range** - The *Intermediate Physical Address* (IPA) range for the index entry.
 - **Type** - The type of the index entry. Valid entries display the level, whether the entry is `block` or `page`, and the size. There are three other types of entry:
 - **RESERVED** shows that the index has a level that is not possible for the selected translation regime. For example, level -2 for 64-bit Virtual Memory System Architecture.
 - **INVALID** shows that the index has levels that are valid for the stage 2 translation regime of EL0 and EL1 but is inconsistent with the configuration of `VTCR_EL2.TOSZ` or `VSTCR_EL2.TOSZ`.
 - **UNKNOWN** shows that the state of the MMU tables cannot be determined for the index entry.
 - **Security State** - The security state of the index entry, `Secure`, `Non-secure`, or `Realm`.
 - **valid** - The validity of the index entry, `Y` for valid or `N` for invalid.

Restrictions

This command only applies to Arm®v9-A processors that implement hardware dirty state structures (FEAT_HDBSS and FEAT_HACDBS).

Example: Displaying all dirty state entries

The following command displays all dirty state entries for the `EL1S_S2_HDBSSPROD_EL2` structure.

```
mmu dirty-state EL1S_S2_HDBSSPROD_EL2 all
```

Base address = NP:0x80000000, Size = 4KB, Index = 4 / 63

Index	IPA Range	Type	Security State	Valid
0	0x00000000 - 0x3FFFFFFF	Level 1 Block (1GB)	Secure	Y
1	0x1C000000 - 0x1C1FFFFFF	Level 2 Block (2MB)	Non-secure	Y
2	0x2F000000 - 0x2F1FFFFFF	Level 2 Block (2MB)	Secure	N
3	0x3F000000 - 0x3F1FFFFFF	Level 2 Block (2MB)	Secure	N
4>	0x200000 - 0x200FFF	Level 3 Page (4KB)	Secure	Y
5	0x210000 - 0x210FFF	Level 3 Page (4KB)	Secure	Y
6	0x86000000 - 0x87FFFFFF	Level 2 Block (2MB)	Secure	Y
7	0x310000 - 0x310FFF	Level 3 Page (4KB)	Secure	Y
8	0xDC000000 - 0xDFFFFFFF	Level 1 Page (1GB)	Non-secure	N

9	0x320000-0x320FFF	Level 3 Page (4KB)	Secure	Y
10	0x1C000000 - 0x1C1FFFFFF	Level 2 Block (2MB)	Non-secure	Y
11	0xCEADCEEF	Level -1 RESERVED	Non-secure	Y
...				
62	0xFF300000	Level 0 INVALID	Secure	N
63	0xFF400000	Level 0 INVALID	Non-secure	N

Example: Displaying dirty-state entries after the current index

The following command displays 5 dirty state entries for the EL1S_S2_HDBSSPROD_EL2 structure starting with, and including, the current index.

```
mmu dirty-state EL1S_S2_HDBSSPROD_EL2 5
```

Base address = NP:0x80000000, Size = 4KB, Index = 4 / 63				
Index	IPA Range	Type	Security State	Valid
4>	0x200000-0x200FFF	Level 3 Page (4KB)	Secure	Y
5	0x210000-0x210FFF	Level 3 Page (4KB)	Secure	Y
6	0x86000000-0x87FFFFFF	Level 2 Block (4KB)	Secure	Y
7	0x310000-0x310FFF	Level 3 Page (4KB)	Secure	Y
8	0xDC000000-0xDFFFFFFF	Level 1 Page (1GB)	Non-secure	N

Example: Displaying dirty-state entries before the current index

The following command displays 10 dirty state entries for the EL1S_S2_HDBSSPROD_EL2 structure before, and including, the current index. Only 5 entries are displayed as the count goes below the zero index entry.

```
mmu dirty-state EL1S_S2_HDBSSPROD_EL2 -10
```

Base address = NP:0x80000000, Size = 4KB, Index = 4 / 63				
Entry	IPA Range	Type	Security State	Valid
0	0x00000000 - 0x3FFFFFFF	Level 1 Block (1GB)	Secure	Y
1	0x1C000000 - 0x1C1FFFFFF	Level 2 Block (2MB)	Non-secure	Y
2	0x2F000000 - 0x2F1FFFFFF	Level 2 Block (2MB)	Secure	N
3	0x3F000000 - 0x3F1FFFFFF	Level 2 Block (2MB)	Secure	N
4>	0x200000-0x200FFF	Level 3 Page (4KB)	Secure	Y

Example: Displaying dirty state entries using a parameter to interpret the structure

The following command displays 3 dirty state entries for the EL1S_S2_HDBSSPROD_EL2 structure using the HDBSSPROD_EL2 parameter to interpret the structure.

```
mmu dirty-state EL1S_S2_HDBSSPROD_EL2 HDBSSPROD_EL2=0x1 3
```

Base address = NP:0x80000000, Size = 4KB, Index = 1 / 63				
Index	IPA Range	Type	Security State	Valid
1>	0x1C000000 - 0x1C1FFFFFF	Level 2 Block (2MB)	Non-secure	Y
2	0x2F000000 - 0x2F1FFFFFF	Level 2 Block (2MB)	Secure	N
3	0x3F000000 - 0x3F1FFFFFF	Level 2 Block (2MB)	Secure	N

Related information

[mmu](#) on page 41

2.3.103 mmu list dirty-state

Lists the dirty state structures with associated parameters that are available in the target.

Syntax

```
mmu list dirty-state
```

Alias:

```
mm l d
```

Parameters

None.

Restrictions

This command only applies to Arm®v9-A processors that implement hardware dirty state structures (FEAT_HDBSS and FEAT_HACDBS).

Example: Listing dirty state structures

```
mmu list dirty-state

Available dirty state structures:
  EL1S_S2_HDBSSPROD_EL2
    parameters: HDBSSBR_EL2, HDBSSPROD_EL2
  EL1S_S2_HACDBSCONS_EL2
    parameters: HACDBSBR_EL2, HACDBSCONS_EL2
  EL1N_S2_HDBSSPROD_EL2
    parameters: HDBSSBR_EL2, HDBSSPROD_EL2
  EL1N_S2_HACDBSCONS_EL2
    parameters: HACDBSBR_EL2, HACDBSCONS_EL2
```

Related information

[mmu](#) on page 41

[mmu dirty-state](#) on page 150

2.3.104 mmu list memory-maps, mpu list memory-maps

Lists the available memory maps and their associated parameters.

Syntax

```
mmu list memory-maps
mpu list memory-maps
```

Parameters

None.

Example: Listing memory maps

```
mmu list memory-maps
Available memory maps:
  PL1S_S1
    parameters: S_SCTLR, S_TTBCR, S_TTBR0, S_TTBR1
  PL1N_S1
    parameters: N_TTBR1, N_TTBCR, N_SCTLR, N_TTBR0
mpu list memory-maps
Available memory maps:
  MPU
```

Related information

[mmu](#) on page 41[mpu](#) on page 42[mpu list](#) on page 42

2.3.105 mmu list tables, mpu list tables

Lists the available translation tables and their associated parameters.

Syntax

```
mmu list tables
mpu list tables
```

Parameters

None.

Example: Listing translation tables

```
mmu list tables
Available translation tables:
  PL1S_S1_TTBR0
    parameters: S_TTBCR, S_TTBR0, S_SCTLR
  PL1S_S1_TTBR1
    parameters: S_TTBCR, S_TTBR1, S_SCTLR
  PL1N_S1_TTBR0
    parameters: N_TTBCR, N_TTBR0, N_SCTLR
  PL1N_S1_TTBR1
    parameters: N_TTBCR, N_TTBR1, N_SCTLR

mpu list tables
Available translation tables:
  MPU_MPU_S
  MPU_MPU_NS
  MPU_SAU
  MPU_IDAU
```

Related information

[mmu](#) on page 41[mpu](#) on page 42[mpu list](#) on page 42

2.3.106 mmu list translations

Lists the available translations and their associated parameters.

Syntax

```
mmu list translations
```

Parameters

None.

Example: Listing translations

```
mmu list translations
Available address translations:
  PL1S_S1
    parameters: S_SCTLR, S_TTBCR, S_TTBR0, S_TTBR1
  PL1N_S1
    parameters: N_TTBR1, N_TTBCR, N_SCTLR, N_TTBR0
```

Related information

[mmu](#) on page 41

2.3.107 mmu memory-map, mpu memory-map

Prints the memory map.

Syntax

```
mmu memory-map [memory-map] [param1=<value1>]...
mpu memory-map [memory-map] [param1=<value1>]...
```

Parameters

memory-map

Specifies the memory map to print. If you do not specify a memory map, then the command prints the most relevant memory map.

param1=<value1>

Specifies a parameter and its value to govern the interpretation of the memory map. If you do not specify a required parameter, then it is determined from the current target state.

Example: Printing memory maps

```
mmu memory-map PL1S_S1 S_TTBR1=0x80000404A
Virtual Range      | Physical Range      | Type   | AP | C   | S   | X
-----
0x00000000-0x00007FFF | <unmapped>
0x00008000-0x00008FFF | 0x8DC4B000-0x8DC4BFFF | Normal | RO | True | True | True
0x00009000-0x00009FFF | 0x8DC4D000-0x8DC4DFFF | Normal | RO | True | True | True
0x0000A000-0x0000AFFF | 0x8DC69000-0x8DC69FFF | Normal | RO | True | True | True
```

0x0000B000-0x0000BFFF	0x8DC6B000-0x8DC6BFFF	Normal	RO	True	True	True
0x0000C000-0x0000CFFF	0x8DE2B000-0x8DE2BFFF	Normal	RO	True	True	True
0x0000D000-0x0000DFFF	0x8DC9E000-0x8DC9EFFF	Normal	RO	True	True	True
0x0000E000-0x0000EFFF	0x80EB0000-0x80EB0FFF	Normal	RO	True	True	True

mpu memory-map

Virtual Range		Physical Range		Type	SA	AP (Priv)
AP (Unpriv)	X	C	S			
0x00000000-0x1FFFFFFF		0x00000000-0x1FFFFFFF		Normal	SECURE	RW
RW	True	True	False			
0x20000000-0x3FFFFFFF		0x20000000-0x3FFFFFFF		Normal	SECURE	RW
RW	True	False	False			
0x40000000-0x5FFFFFFF		0x40000000-0x5FFFFFFF		Device-nGnRE	SECURE	RW
RW	False	False	True			
0x60000000-0x7FFFFFFF		0x60000000-0x7FFFFFFF		Normal	SECURE	RW
RW	True	False	False			
0x80000000-0x9FFFFFFF		0x80000000-0x9FFFFFFF		Normal	SECURE	RW
RW	True	True	False			
0xA0000000-0xDFFFFFFF		0xA0000000-0xDFFFFFFF		Device-nGnRE	SECURE	RW
RW	False	False	True			
0xE0000000-0xE00FFFFF		0xE0000000-0xE00FFFFF		Device-nGnRnE	SECURE	RW
RW	False	False	True			
0xE0100000-0xFFFFFFFF		0xE0100000-0xFFFFFFFF		Device-nGnRE	SECURE	RW
RW	False	False	True			

Related information

[mmu](#) on page 41

[mpu](#) on page 42

2.3.108 mmu print, mpu print

Prints the contents of a translation table.

Syntax

```
mmu print [table] [param1=<value1>]...  
mpu print [table] [param1=<value1>]...
```

Parameters

- table**
Specifies the translation table to print. If you do not specify a table, the command prints all tables for the current translation regime.
- param1=<value1>**
Specifies a parameter and its value to govern the interpretation of the table. If you do not specify a required parameter, then it is determined from the current target state.

Operation

Printing translation tables might be slow on some targets because it might involve a full traversal of the translation tables on the target.

Example: Printing translation tables

```
mmu print PL1S_S1_TTBR0
```

SP:0x80F15000

Input Address	Type	Next Level	Output Address	Properties
+ 0x00000000	TTBR0	SP:0x0080500000		
- 0x00000000	Fault (x704)			
- 0x2C000000	Section		SP:0x002C000000	NS=0, nG=0, S=0
- 0x2C100000	Fault (x1343)			
- 0x80000000	Section		SP:0x0080000000	NS=0, nG=0, S=1
- 0x80100000	Fault (x2047)			
+ 0xFFFFFFFF	TTBR1	SP:0x009082C300		

mpu print

Base	Limit	Type	Properties
+ MPU (Secure) 0=0x0, MAIR 1=0x0			ENABLE=0, HFNMIENA=0, PRIVDEFENA=0, MAIR
- 0x00000000	0x00000000	Region	SH=0, AP=0, XN=0, AttrIndex=0, EN=0
- 0x00000000	0x00000000	Region	SH=0, AP=0, XN=0, AttrIndex=0, EN=0
- 0x00000000	0x00000000	Region	SH=0, AP=0, XN=0, AttrIndex=0, EN=0
- 0x00000000	0x00000000	Region	SH=0, AP=0, XN=0, AttrIndex=0, EN=0
- 0x00000000	0x00000000	Region	SH=0, AP=0, XN=0, AttrIndex=0, EN=0
- 0x00000000	0x00000000	Region	SH=0, AP=0, XN=0, AttrIndex=0, EN=0
+ MPU (Non-Secure) 0=0x0, MAIR 1=0x0			ENABLE=0, HFNMIENA=0, PRIVDEFENA=0, MAIR
- 0x00000000	0x00000000	Region	SH=0, AP=0, XN=0, AttrIndex=0, EN=0
- 0x00000000	0x00000000	Region	SH=0, AP=0, XN=0, AttrIndex=0, EN=0
- 0x00000000	0x00000000	Region	SH=0, AP=0, XN=0, AttrIndex=0, EN=0
- 0x00000000	0x00000000	Region	SH=0, AP=0, XN=0, AttrIndex=0, EN=0
- 0x00000000	0x00000000	Region	SH=0, AP=0, XN=0, AttrIndex=0, EN=0
- 0x00000000	0x00000000	Region	SH=0, AP=0, XN=0, AttrIndex=0, EN=0
+ SAU			ALLNS=0, ENABLE=0
- 0x00000000	0x00000000	Region	NSC=0, ENABLE=0
- 0x00000000	0x00000000	Region	NSC=0, ENABLE=0
- 0x00000000	0x00000000	Region	NSC=0, ENABLE=0
- 0x00000000	0x00000000	Region	NSC=0, ENABLE=0
- 0x00000000	0x00000000	Region	NSC=0, ENABLE=0
- 0x00000000	0x00000000	Region	NSC=0, ENABLE=0
+ IDAU			[Not Configured]

Related information

[mmu](#) on page 41

[mpu](#) on page 42

2.3.109 mmu translate

Performs translations between virtual and physical addresses.

It translates either:

- From a virtual address to a physical address.
- From a physical address to one or more virtual addresses.

Syntax

```
mmu translate {address} [translation] [param1=<value1>]...
```

Parameters

{address}

Specifies the address to translate. If this is a virtual address then a virtual to physical address translation is performed. If this is a physical address then a physical to virtual address translation is performed.

translation

Specifies the translation to perform.

param1=<value1>

Specifies a parameter and its value to govern the interpretation of the table. If you do not specify a required parameter, then it is determined from the current target state.

Operation

Physical to virtual address translation might be slow on some targets because it might involve a full traversal of the translation tables on the target.

Example: Translating between addresses

```
mmu translate 0x00008000 PL1S_S1 S_TTBR1=0x80000404A
SP:0x80F15000

mmu translate SP:0x80F15000
Address SP:0x80F15000 maps to
0x00008000
0x80F15000
```

Related information

[mmu](#) on page 41

2.3.110 newvar

Declares and initializes a new debugger convenience variable or register alias.

Syntax

```
newvar [global] $<name>[=<initial_value>]
newvar /r [group]$<name>=<initial_value>
newvar /d $<symbol>
```

Parameters

global

Specifies that the variable has global scope. If global is not specified, then the variable is accessible only within its enclosing lexical scope. For lexical scope:

- 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.

- 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.

/r

Defines a register alias. A register alias always has global scope.

/d

Deletes the specified variable, register alias, or register group.

group

The register group for the register alias. If the specified group does not already exist, it is created. If a group is not specified, the register alias is added to the `Alias` group.

<name>

Specifies the name of the new variable or register alias. The name must be a valid C identifier but prefixed with \$.

<symbol>

Specifies a symbol that resolves to a variable, register, or register group to delete.

<initial_value>

Specifies the initial value of the variable.

- For variables with no initial value specified, the variable defaults to an integer of 0.
- For register aliases, <initial_value> must be a register name, the instruction encoding for a system register, or an address for the register. Specifying the <initial_value> as an instruction encoding works with hardware but not with models.



When the <initial_value> for a register alias is an instruction encoding or memory address, Arm® Debugger does not check whether these values are correct.

Example: Saving the number of a breakpoint as a variable

The following code defines a new command that runs to an address using a hardware breakpoint.

```
define advance_hw
  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
```

Example: Specifying a register alias by instruction encoding

The following command creates the `newreg` register alias for a register with an instruction encoding of `$s3_0_c4_c3_1`, and adds the variable to the `MyGroup` group. The alias is shown in use in other commands.

```

newvar /r $System::$MyGroup::$newreg=$S3_0_C4_C3_1 # Define a register alias.
output /x $System::$MyGroup::$newreg               # Display the value of the
                                                    # register using the alias.
set $System::$MyGroup::$newreg = 0x88              # Modify the value of the
                                                    # register using the alias.
newvar /d $System::$MyGroup::$newreg               # Delete the register alias.

```

Example: Specifying a register alias by address

The following command creates the `newregaddr` register alias for a register at an address, and adds the variable to the default `Alias` group. The alias is shown in use in other commands.

```

newvar /r $newregaddr=*0xE000EF34 # Define a register alias.
output /x $newregaddr             # Display the value using the alias.
set $newregaddr = 0x88            # Modify the value using the alias.

```

Related information

[Accessing system registers by instruction encoding](#) on page 17

[Scripts](#) on page 33

2.3.111 next

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]
```

Alias:

```
n [count]
```

Parameters

count

Specifies the number of source lines to execute. If not set, defaults to 1.

Operation

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

Example: Stepping through source lines

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

Related information

[step](#) on page 229

[stepi](#) on page 230

[steps](#) on page 231

[nexti](#) on page 161

[nexts](#) on page 162

[Execution control](#) on page 31

2.3.112 nexti

Steps through an application at the instruction level but stepping over all function calls.

Syntax

```
nexti [count]
```

Alias:

```
ni [count]
```

Parameters

count

Specifies the number of instructions to execute. If not set, defaults to 1.

Operation

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

Example: Stepping through instructions

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

Related information

[step](#) on page 229

[stepi](#) on page 230

[steps](#) on page 231

[next](#) on page 160

[nexts](#) on page 162

[Execution control](#) on page 31

2.3.113 nexts

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]
```

Alias:

```
ns [count]
```

Parameters

count

Specifies the number of source statements to execute. If not set, defaults to 1.

Operation

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

Example: Stepping through source statements

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

Related information

[step](#) on page 229

[stepi](#) on page 230

[steps](#) on page 231

[next](#) on page 160

[nexti](#) on page 161

[Execution control](#) on page 31

2.3.114 nosharedlibrary

Discards all loaded shared library symbols.

Syntax

```
nosharedlibrary
```

Parameters

None.

Operation

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

Example: Discarding loaded shared library symbols

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

Related information

[Operating System](#) on page 35

2.3.115 output

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}
```

Parameters

<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: Outputting an expression

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

Example: Outputting matrix expressionsThe following examples show access to the tiles available with *Scalable Matrix Extension*, SME2:

```
output $ZA2H_S.F32[0][0] # Print the zeroth element of the zeroth row of
                        # tile ZA2
output $ZA2H_S.F32[0]    # Print all elements of the zeroth row of tile ZA2
output $ZA2V_S.F32[0]    # Print all elements of the zeroth column of
                        # tile ZA2
output /x $ZA0H_S.F32[$w13] # Print the w13th row of ZA0 in hexadecimal
output $ZA2H_S.F32        # Print the whole tile ZA2 as rows
output $ZA2V_S.F32        # Print the whole tile ZA2 as columns
```

Related information[Display](#) on page 43**2.3.116 pause**

Pauses the execution of a script for a specified period of time.

Syntax

```
pause {number} [ms|s]
```

Parameters**number**

Specifies the period of time.

ms

Specifies the time in milliseconds. This is the default.

s

Specifies the time in seconds.

Example: Pausing script execution

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

Related information

[Support](#) on page 51

2.3.117 peripheral add

Creates a peripheral register group containing the memory-mapped registers from a register frame.

Syntax

```
peripheral add {id <group>} {type <frame>} [address <expression>]
```

Alias:

```
pe a {id <group>} {type <frame>} [address <expression>]
```

Parameters

id

Specifies the full path to the register group to create. Groups in the path that do not exist are created. If the target has registers for more than one execution state, for example both AArch64 and AArch32 registers, the root of the path is the peripherals register set. The group holds the registers from the register frame.

type

Specifies the type of the register <frame> containing the registers to add to the register group. Use [peripheral show](#) to list the supported register frames.

address

Specifies an <expression> that provides the base address of the register frame. If not provided, and there is a default base address available, then the default is used.

Example: Creating register groups

The following commands create the \$GIC::\$GICR0_RD_base and \$GIC::\$GICR0_SGI_base register groups.

```
peripheral add id $GIC::$GICR0_RD_base type "RD_base" address "$IMP_CBAR+0x100000"
peripheral add id $GIC::$GICR0_SGI_base type "SGI_base" address "$IMP_CBAR+0x110000"
```

Related information

[peripheral delete](#) on page 166

2.3.118 peripheral delete

Deletes peripheral register groups.

Syntax

```
peripheral delete {<group_path>}
```

Alias:

```
pe d {<group_path>}
```

Parameters

<group_path>

Specifies the peripheral register group or groups to delete. You must include the path to the group. If a partial group path is provided, all register groups below <group_path> are deleted.

Example: Deleting peripheral register groups

```
peripheral delete $GIC::$GICR0_RD_base    # Delete the $GICR0_RD_base
                                           # peripheral register group
peripheral delete $GIC                     # Delete all peripheral register
                                           # groups below $GIC
```

Related information

[peripheral add](#) on page 165

[peripheral show](#) on page 166

2.3.119 peripheral show

Lists available peripherals, components, register frames, and registers.

Syntax

```
peripheral show [<peripheral>] [<component>] [<frame>]
```

Alias:

```
pe s [<peripheral>] [<component>] [<frame>]
```

Parameters

<peripheral>

Lists components and register frames for the specified peripheral.

<component>

Lists register frames for the specified peripheral component.

<frame>

Lists registers for the specified peripheral register frame.

No parameters

If you do not specify any arguments, all peripherals are listed.

Example: Listing all peripherals

```
>peripheral show
GICv4
GICv3
```

Example: Listing components in a peripheral

```
>peripheral show GICv4
GICD
  Dist_base
  MSI_base
GICR
  RD_base
  SGI_base
  VLPI_base
GITS
  GITS_ctrl
  GITS_translation
GICC
  GICC
GICV
  GICV
GICH
  GICH
```

Related information

[peripheral add](#) on page 165

[peripheral delete](#) on page 166

2.3.120 preprocess

Displays the preprocessed expression, not the evaluated expression.

Syntax

```
preprocess [expression]
```

Parameters**expression**

Is a C/C++ preprocessor expression.

Operation

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

As an example of the `preprocess` operation, 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
```

Related information

[Support](#) on page 51

2.3.121 `print`, `inspect`

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]
```

Alias:

```
p [/<flag>] [expression]
ins [/<flag>] [expression]
```

Parameters

<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: Printing an expression

```
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)
```

Example: Printing matrix expressions

The following examples show access to the tiles available with *Scalable Matrix Extension*, SME2:

```
print $ZA2H_S.F32[0][0] # Print the zeroth element of the zeroth row of
                        # tile ZA2
print $ZA2H_S.F32[0]    # Print all elements of the zeroth row of tile ZA2
print $ZA2V_S.F32[0]    # Print all elements of the zeroth column of
                        # tile ZA2
print /x $ZA0H_S.F32[$w13] # Print the w13th row of ZA0 in hexadecimal
print $ZA2H_S.F32        # Print the whole tile ZA2 as rows
print $ZA2V_S.F32        # Print the whole tile ZA2 as columns
```

Related information

[Display](#) on page 43

2.3.122 pwd

Displays the current working directory.

Syntax

```
pwd
```

Parameters

None.

Example: Displaying the current working directory

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

Related information

[Files](#) on page 36

2.3.123 quit, exit

Quits the debugger session.

Syntax

```
quit  
exit
```

Alias:

```
q
```

Parameters

None.

Example: Quitting the session

```
quit      # Quit debugger session
```

Related information

[Support](#) on page 51

2.3.124 reload-symbol-file

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]
```

Parameters

filename

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

Example: Reloading debug information

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

Related information

[Files](#) on page 36

2.3.125 reset

Performs a reset on the target. The exact behavior of the `reset` command depends 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.

Syntax

```
reset [key]
```

Parameters

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

Restarts a Linux application, for Linux Application debug sessions via gdbserver.

system

Asserts the AIRCR.SYSRESETREQ bit when the target has this register and bit available. This is intended to force a large system reset of all major components except for debug.

vector

Asserts the AIRCR.VECTRESET bit when the target has this register and bit available. This is intended as a localised processor reset.

hardware

General hardware reset that is not specific to a processor.

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

Restrictions

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

Example: Resetting a target

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

Related information

[Execution control](#) on page 31

2.3.126 resolve

Re-evaluates the specified breakpoints or watchpoints and those with addresses that can be resolved are set. Unresolved addresses remain pending.

Syntax

```
resolve [number] ...
```

Parameters

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: Resolving breakpoints and watchpoints

```

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

```

Related information

[break](#) on page 70

[hbreak](#) on page 104

[tbreak](#) on page 232

[thbreak](#) on page 233

[clear](#) on page 78

[Breakpoints and watchpoints](#) on page 29

2.3.127 restore

Reads data from a file and writes it to memory. The debugger supports and automatically recognizes the following file formats:

- Intel Hex-32
- Motorola 32-bit (S-records)
- Byte oriented hexadecimal (Verilog Memory Model)
- 32-bit Arm ELF
- 64-bit Arm ELF

Binary files are also supported but require the `binary` parameter to be set.

Syntax

```

restore {filename} [binary] [offset [start_address [end_address | +<size>]]]

```

Parameters

filename

Specifies the file.

binary

Specifies binary format. This is only required for binary files, because the debugger does not automatically recognize them.

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: Reading file data and writing it to memory

```
restore myFile.bin binary 0x200      # Write the 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 write
                                     # the content between 0x8000-0x8FFF
```

Related information

[Files](#) on page 36

[Memory group](#) on page 39

[set elf load-segments-at-p_paddr](#) on page 188

2.3.128 run

Starts running the target.

Syntax

```
run [args]
```

Alias:

```
r [args]
```

Parameters

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.

Operation

This command operates as follows:

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.



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.

Example: Running a device

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

2.3.129 rwatch

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

Syntax

```
rwatch [-d] [-p] [-w <width> | -m <mask> | -s <size>] [-f] {[filename:]symbol | *<address>} [vmid <number>] [if <condition>]
```

Parameters

-d

Creates the watchpoint disabled.

-p

Specifies whether or not the resolution of an unrecognized watchpoint location results in a pending watchpoint being created.

-w <width>

Specifies the width to watch at the given address, in bits. Accepted values are: 8, 16, 32, and 64 if supported by the target. This option is optional.

The width defaults to:

- 32 bits for an address.
- The width corresponding to the type of the symbol or expression, if entered.

-m <mask>

This option is only available for Arm®v6-M, Armv7-M, Armv8-M, Armv8-R, Armv8-A, and Armv9-A hardware.

Specifies a mask that represents a memory range. The mask value is the number of least significant bits that will be masked. For example, a value of 5 would watch a range of 32 bytes.



The range could be widened if the watched address is not specified as a power of 2.

-s <size>

This option is only available for Armv6-M, Armv7-M, Armv8-M, Armv8-R, Armv8-A, and Armv9-A hardware.

Specifies the size of a memory area to watch. If the <size> and *<address> are not specified so that the range is exactly coverable by an address mask, you must set the -f option.

-f

This option is only available for Armv6-M, Armv7-M, Armv8-M, Armv8-R, Armv8-A, and Armv9-A hardware.

You must set this parameter if the hardware does not support arbitrary ranges and the range described by *<address> and <size> is not exactly coverable by an address mask. If set, a mask is used that is inclusive of the range specified by <size>, but may extend beyond that range. If the mask extends beyond the range specified by <size>, the Arm Debugger could halt if there are accesses outside of the intended range.

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 option can be either an address or an expression that evaluates to an address.

vmid <number>

Specifies the Virtual Machine ID (VMID) to apply the watchpoint to. This option can be either an integer or an expression that evaluates to an integer. Applicable only on targets which support hypervisor / virtual machine debugging.

if <condition>

Specifies the condition which must evaluate to true at the time the watchpoint is triggered for the target to stop. You can create several conditional watchpoints, but when a conditional watchpoint is enabled, no other watchpoints (regardless of whether they are conditional) can be enabled.

Operation

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.

Watchpoints are supported on single memory addresses for all processors, on hardware and models. For M-class Arm processor hardware, watchpoints are also supported on structs and arrays, and arbitrary memory ranges.

The availability of watchpoints depends on your target. In the case of Linux application debug using gdbserver, the availability of watchpoints also depends on the Linux kernel version and configuration.

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.

Example: Setting a read watchpoint

```

rwatch myVar1           # Set read watchpoint on myVar1
rwatch *0x80D4          # Set read watchpoint on address 0x80D4
rwatch myVar1 if myVar1 == 2 # Set read watchpoint on myVar1 which
                           # is hit only if myVar1 evaluates to 2
rwatch myVar1 if ($LR & 0xFF) == 0x12 # Set read watchpoint on myVar1 which
                           # is hit only if ($LR & 0xFF) evaluates
                           # to 0x12 when myVar1 is accessed
rwatch -s 200 *0x80D4    # On Armv6-M, Armv7-M, Armv8-A, and
                           # Armv8-R hardware, set a read
                           # watchpoint on address 0x80D4 with
                           # a range of 200 bytes
rwatch -m 5 *0x8000      # On Armv6-M, Armv7-M, Armv8-A, and
                           # Armv8-R hardware, set a read
                           # watchpoint on address 0x8000 with
                           # a range of 32 bytes

```

Related information

[watch](#) on page 251

[clearwatch](#) on page 79

[awatch](#) on page 67

[Breakpoints and watchpoints](#) on page 29

2.3.130 select-frame

Moves the current frame pointer in the call stack.

Syntax

```
select-frame {number}
```

Parameters

number

Specifies the frame number.

Operation

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.

Example: Moving the frame pointer in the stack

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

Related information

[Call stack](#) on page 34

2.3.131 set arm

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

Syntax

```
set arm {option}
```

Parameters

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.

Operation

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

Example: Controlling debugger behavior

```
set arm force-mode t32          # Force the use of T32
set arm fallback-mode a32      # When force-mode is auto, use A32
                                # if no debug information is available
```

Related information

[Set](#) on page 46

[Support](#) on page 51

2.3.132 set auto-solib-add

Controls the automatic loading of shared library symbols.

Syntax

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

Parameters

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.

Operation

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

Example: Controlling automatic loading of shared library symbols

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

Related information

[Operating System](#) on page 35

[Set](#) on page 46

2.3.133 set backtrace

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

Syntax

```
set backtrace {option}
```

Parameters

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.

past-main [on | off]

Specifies whether the stack trace shows the frames before `main()`:

on

Stack trace shows the frames before `main()`.

off

Stack trace stops at `main()`. If `main()` does not exist, the stack trace behaves as if `past-main` is set to `on`.

The default is `off`.

Example: Controlling info stack default behavior

```
set backtrace limit 10      # Limit the call stack display to 10 frames
```

```
set backtrace limit 0      # No limit, display the entire call stack
set backtrace past-main on # Stack trace shows the frames before main()
```

Related information

[Call stack](#) on page 34

[Set](#) on page 46

2.3.134 set blocking-run-control

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}
```

Parameters

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 in a script.

This is the default.

Example: Controlling run control operations blocking

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

Related information

[Execution control](#) on page 31

[Set](#) on page 46

2.3.135 set breakpoint

Controls the automatic behavior of breakpoints and watchpoints.

Syntax

```
set breakpoint [option]
```

Parameters

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.



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

skipmode

Controls whether to skip all non-temporary breakpoints and watchpoints:

off

Disables skip mode. This is the default.

on

Enables skip mode.



In skip mode, temporary breakpoints set using `tbreak`, `thbreak`, `set debug-from`, or `advance` are not skipped.

Example: Controlling the automatic behavior of breakpoints and watchpoints

```

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

```

Related information

[disable breakpoints](#) on page 86

[delete breakpoints](#) on page 84

[info breakpoints](#), [info watchpoints](#) on page 111

[info capabilities](#) on page 112

[Breakpoints and watchpoints](#) on page 29

[Set](#) on page 46

2.3.136 set case-insensitive-source-matching

Controls the case sensitivity of debugger file matching operations.

Syntax

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

Parameters

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: Controlling debugger file matching case sensitivity

```
# 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
```

Related information

[Set](#) on page 46

2.3.137 set cde-coprocessors

Specify the coprocessors that are associated with the *Arm Custom Datapath Extension* (CDE).

Default

By default, coprocessors are associated with the *General CoProcessor* (GCP) encoding.

Syntax

```
set cde-coprocessors {coprocessor=<type>}[, coprocessor=<type>[, ...]]
```

Parameters

coprocessor

Specifies the coprocessor. Valid values are P0 to P7.

<type>

Specifies the encoding pattern the coprocessor supports. The types are:

CDE

Support the CDE encoding pattern.

GCP

Support the GCP encoding pattern.

Operation

You can provide multiple coprocessor associations in a single command, separated by commas.

Example: specify a single coprocessor association

```
set cde-coprocessors p1=cde           # Associates coprocessor P1 with the
                                       # CDE. All other coprocessors default
                                       # to the general coprocessor encoding.
```

Example: specify multiple coprocessor associations

```
set cde-coprocessors p1=cde, p4=gcp, p5=cde # Associates coprocessor P1 and P5 with
                                              # the CDE.
```

Related information

[show cde-coprocessors](#) on page 210

2.3.138 set debug-agent

Sets an internal configuration parameter for the debug agent. The available parameters depend on the debug agent, such as the DSTREAM family of devices or gdbserver.

Syntax

```
set debug-agent {name} {value}
```

Parameters

name

Specifies the name of the parameter to set.

value

Specifies the value of the parameter. Values depend on the parameter being set. An error is reported if the value is not valid.

Example: Setting a debug agent internal configuration parameter

```
set debug-agent UserOut_P1 1

# Set value of USER OUT pin1 to 1.
# This parameter is available for DSTREAM connections.
```

Related information

[Set](#) on page 46

2.3.139 set debug-from

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}
```

Parameters**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: Specifying the address of a temporary breakpoint used by start

```
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
```

Related information

[Execution control](#) on page 31

[Set](#) on page 46

2.3.140 set directories

`set directories` is an alias for [directory](#).

2.3.141 set dtsl-options

Sets a parameter in the DTSL configuration.

Syntax

```
set dtsl-options {name} {value}
```

Parameters

name

Specifies a name of the parameter to set.

value

Specifies the value of the parameter. Values depend on the parameter being set. An error is reported if the value is not valid.

Example: Setting a DTSL configuration parameter

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

Related information

[Set](#) on page 46

2.3.142 set dtsl-temporary-directory

Specifies the path for the temporary directory to store trace data.

Syntax

```
set dtsl-temporary-directory {path}
```

Parameters

path

Specifies the location of your temporary directory, for example, c:my_temp_dir.

To clear the setting and revert to the default system directory, enter an empty string:

```
set dtsl-temporary-directory ""
```

Restrictions

This command can only set the path to an existing directory location. You must create the directory before using this command.

Operation

In Arm® Development Studio, you can also use the **Arm DS Preferences** dialog box to set trace data temporary directory. To do this:

1. In Arm Development Studio, select **Window > Preferences**.
2. Browse to **Arm DS > Debugger > Trace**.
3. Select the **Use custom directory for temporary trace data files** option.
4. Enter or **Browse** the path to your temporary directory.

Example: Specifying the trace data temporary directory path

```
set dtsl-temporary-directory C:\my_temp_dir      # Set DTSL temporary directory path
                                                  # as C:\my_temp_dir.
```

Related information

[Set](#) on page 46

2.3.143 set elf cache-uninitialized-sections

Controls whether the debugger caches uninitialized sections.

Syntax

```
set elf cache-uninitialized-sections {off | on}
```

Parameters

off

Disables caching of uninitialized sections.

on

Enables caching of uninitialized sections. This is the default.

Operation

After the symbols for an image are loaded, the debugger by default marks regions corresponding to ELF sections as cacheable if:

- The section has `sht_type` that is set to one of:
 - `SHT_PROGBITS`
 - `SHT_INIT_ARRAY`
 - `SHT_FINI_ARRAY`
 - `SHT_PREINIT_ARRAY`
 - `SHT_NOBITS`.
- The `SHF_ALLOC` flag in `sh_flags` is set for the section.

This can result in uninitialized sections, or volatile regions of the address space, for example peripherals, being set to cacheable by default. To overcome this problem, you can use `set elf cache-uninitialized-sections off` to disable the debugger from caching such ELF sections.

Example: Controlling caching of uninitialized sections

```
set elf cache-uninitialized-sections off      # Disable caching of uninitialized
                                              # sections
```

Related information

[Set](#) on page 46

2.3.144 set elf load-segments-at-p_paddr

Controls whether the specified load offset is + `p_paddr` or + `p_vaddr` when loading segments of ELF images to the target.

Syntax

```
set elf load-segments-at-p_paddr {off | on}
```

Parameters

off

Loads to the specified load offset + `p_vaddr`.

on

Loads to the specified load offset + `p_paddr`. This is the default.

Operation

When loading segments of ELF images to the target, by default, the debugger loads to the specified load offset + `p_paddr` specified in the ELF Program Header for that segment. If you want the debugger to load to the specified load offset + `p_vaddr`, then disable `elf load-segments-at-p_paddr`.



Note

The ELF Program Header for the corresponding segment specifies the `p_vaddr`.

Example: Controlling the specified load offset

```
set elf load-segments-at-p_paddr off      # Loads to the specified load offset
                                              # + p_vaddr
```

Related information

[Set](#) on page 46

2.3.145 set elf zero-extra-segment-bytes

Enables zeroing of bytes from `p_filesz` to `p_memsz` when loading segments of ELF images to the target.

Syntax

```
set elf zero-extra-segment-bytes {off | on}
```

Parameters

off

Disables zeroing. This is the default.

on

Enables zeroing of the region from `p_filesz`

Operation

When loading segments of ELF images to the target, by default, the debugger only writes `p_filesz` bytes to the target. If `p_filesz` is less than `p_memsz`, and you want the debugger to pad the region from `p_filesz` to `p_memsz` with zero then enable `elf zero-extra-segment-bytes`.



Note

The ELF Program Header for the corresponding segment specifies the `p_filesz`.

Example: Controlling zeroing of bytes

```
set elf zero-extra-segment-bytes on      # Enable zeroing from p_filesz to p_memsz
```

Related information

[Set](#) on page 46

2.3.146 set endian

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}
```

Parameters

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 the Arm®v6 architecture (data is big endian and code is little endian).

big

Specifies big endian mode.

little

Specifies little endian mode.

Example: Specifying the byte order

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

Related information

[Set](#) on page 46

[Support](#) on page 51

2.3.147 set escape-strings

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

Syntax

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

Parameters**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: Controlling if special characters are printed

```
set escape-strings on
output "Say \"hello\""
"Say \"hello\""
set escape-strings off
output "Say \"hello\""
"Say "hello"")
```

Related information

[Set](#) on page 46

2.3.148 set escapes-in-filenames

Controls the use of special characters in paths.

Syntax

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

Parameters

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: Controlling special characters in paths

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

Related information

[Set](#) on page 46

2.3.149 set idau-region

Specifies the *Implementation Defined Attribution Unit* (IDAU) region parameters for each memory range. Targets with Arm®v8-M Security Extension can provide an IDAU which constrains security attribution for an address in an **IMPLEMENTATION DEFINED** manner. To instruct Arm Debugger to take the IDAU into consideration, you must specify the IDAU region using the `set idau-region` command.

Syntax

```
set idau-region {region_number} [base_address] [limit_address] [region_type]
```

Parameters

region_number

Specifies the number of the IDAU region.

To delete an existing IDAU region, specify the region number without any additional parameters.

base_address

Specifies the base address of the IDAU region.

limit_address

Specifies the last address of the IDAU region.

region_type

Specifies the type of security attribution that is provided by the IDAU region. The types are:

EXEMPT

Specifies if the region is exempt from security attribution.

SECURE

Specifies if the region is a secure region.

SECURE_NSC

Specifies if the region is a Non-secure Callable (NSC) memory region.

NON_SECURE

Specifies if the region is a Non-secure memory region.

Example: Setting the IDAU region parameters

```

set idau-region 10 0x80000000 0x8000ffff SECURE # Set IDAU region 10, with base
                                                    # address 0x80000000, limit address
                                                    # 0x8000ffff, and specify the
set idau-region 10                               # region as SECURE.
                                                    # Delete IDAU region 10

```

2.3.150 set listsize

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

Syntax

```
set listsize {n}
```

Parameters

n

Specifies the number of source lines.

Example: Setting listing size

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


Related information

[Set](#) on page 46

2.3.151 set mmu use-cache-for-phys-reads

Instructs the debugger to, where possible, ensure that the translation table entries it reads from physical memory are coherent with the contents of data caches.

Syntax

```
set mmu use-cache-for-phys-reads {off | on}
```

Parameters

off

Does not ensure coherency between physical memory reads and data caches. This is the default.

on

Ensures coherency between physical memory reads and data caches.

Example: Ensuring coherent physical memory reads

```
set mmu use-cache-for-phys-reads on      # Ensure coherent physical memory reads
```

Related information

[mmu](#) on page 41

[Set](#) on page 46

[About debugging MMUs](#)

2.3.152 set os

Controls operating system (OS) settings in the debugger. An OS-aware connection must be established before you can use this command.

Syntax

```
set os {option}
```

Parameters

option

Specifies additional options:

enabled

Controls OS support:

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.

kernel-stack-size <bytes>

Specifies the number of bytes to use for the stack size.

log-capture

Controls logging to the console:

off

Disables OS log capture and printing of Linux kernel `dmesg` logs to the console. This is the default.

on

Enables OS log capture and printing to the console.

**Note**

This option automatically checks the connection state and, if required, stops the target before changing this setting.

physical-address

Specifies the physical address of where the kernel is loaded.

read-all-threads-on-stop

Controls the OS reading of threads:

off

Disables OS reading of threads when the target is stopped. This is the default.

on

Enables OS reading of threads when the target is stopped.

Example: Controlling OS settings

```
set os log-capture on           # Enable OS log capture and printing to the
                                # console
set os enabled off             # Disable OS support in the debugger
set os physical-address 0x80080000 # Specifies the physical address of where
                                # the kernel is loaded as 0x80080000
```

Related information

[Operating System](#) on page 35

[Set](#) on page 46

2.3.153 set overlays enabled

Enables or disables overlay support. The default setting is auto.

Syntax

```
set overlays enabled [on | off | auto]
```

Parameters

on

Enables overlay support.

off

Disables overlay support.

auto

If the required symbols are present in an image during load time, automatically enables overlay support. This is the default.

Example: Setting overlay support

```
set overlays enabled on       # Enable overlay support
set overlays enabled off      # Disable overlay support
set overlays enabled auto     # Enable overlay support if overlay symbols
                                # are detected
```

2.3.154 set print

Controls the current debugger print settings.

Syntax

```
set print {option}
```

Parameters

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: Controlling print settings

```
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

Related information

[Display](#) on page 43[Set](#) on page 46[Expressions in the Arm Debugger](#) on page 15[printf\(\) style format string in Arm Debugger](#) on page 22

2.3.155 set semihosting

Controls the semihosting settings in the debugger. Semihosting is used to communicate input and output requests from application code to the host running the debugger.

**Warning**

Semihosting enables code on the target device to access resources on the debugger host, which can expose sensitive data on the host. Before you use semihosting, you must assess and mitigate the security implications.

By default, if semihosting is enabled by the debugger, the debugger prevents semihosting from accessing files and executing system commands on the debugger host. If required, you can use the `set semihosting policy` command to enable these operations.

Syntax

```
set semihosting {option}
```

Parameters

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 | off | on}

Controls semihosting operations:

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.

You might have to configure semihosting addresses before you enable semihosting. For example:

```
set semihosting top-of-memory address
set semihosting enabled on
```

policy {block | warn | allow}

Controls semihosting policy for the following operations on the host:

- Opening, renaming, and deleting files
- Running system commands

block

The operations are not allowed and an error is displayed for each attempted operation. This is the default.

warn

The operations are allowed but a warning is displayed for each operation.

allow

The operations are allowed.

vector

Allows you to specify the semihosting trap mechanism to use on your target.

ADDR <trap_address>

Specifies a breakpoint address for the vector catch. This instructs the debugger to set a breakpoint at the specified address. When the breakpoint is hit, the debugger takes control to perform the semihosting operation.

SVC

Uses svc vector catch to trap semihosting operations.

UNDEF

Uses UNDEF vector catch to trap semihosting operations.

SVC+UNDEF

Uses SVC+UNDEF vector catch to trap semihosting operations.



- On M-Profile targets, this command produces an error since semihosting is implemented using a BKPT instruction on these targets.
- On Arm®v7 A or R profiles and classic Arm targets, you can use svc, UNDEF, SVC+UNDEF, or the ADDR <trap_address> options to switch between vector catch operations.
- On AArch64 (Arm®v8-A or Armv9-A) targets, use ADDR <trap_address> to enable instruction breakpoint based semihosting.

Restrictions

The semihosting settings only apply if:

- The target supports semihosting
- The debugger is handling semihosting

You cannot change the semihosting settings while the target is running.

Example: Control semihosting settings

```
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
set semihosting vector svc        # Set the semihosting vector catch to SVC
set semihosting vector ADDR 0x800 # Set the semihosting vector catch to 0x800
set semihosting policy block      # Disallow semihosting from running system
                                   # commands or opening/renaming/deleting
                                   # files on the host
```

Related information

[Set](#) on page 46

[Support](#) on page 51

[Using semihosting to access resources on the host computer](#)

2.3.156 set solib-absolute-prefix

`set solib-absolute-prefix` is an alias for [set sysroot](#).

2.3.157 set solib-search-path

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` command to display the current settings.

Syntax

```
set solib-search-path [path]...
```

Parameters

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.

Multiple directories can be specified but must be separated with either:

- a colon (Unix)
- a semi-colon (Windows).

Operation

Before you can use this feature, you must launch the debugger with the `--target_os` command-line option. In Arm® Development Studio, this option is automatically selected when you connect to a target using `gdbserver`.

Example: Specifying directories for shared library symbol searches

```
set solib-search-path "\usr\lib"      # Specify search directory
set solib-search-path "/lib":"/My Lib" # Specify two search directories (Unix)
```

Related information

[Operating System](#) on page 35

[Set](#) on page 46

2.3.158 set step-mode

Controls the default behavior of the `step` and `steps` commands.

Syntax

```
set step-mode {step-over | stop | step-until-source}
```

Parameters

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: Specifying the default behavior of step commands

```
set step-mode step-over          # Step over a function call and stop.  
                                # Otherwise stop
```

Related information

[Execution control](#) on page 31

[Set](#) on page 46

2.3.159 set stop-on-solib-events

Controls whether the debugger stops execution when a shared object is loaded or unloaded.

Syntax

```
set stop-on-solib-events {off | on}
```

Parameters

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.

Operation

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

Example: Stopping execution when a shared object is loaded or unloaded

```
set stop-on-solib-events on      # Stop execution when event occurs
```

Related information

[Operating System](#) on page 35

[Set](#) on page 46

2.3.160 set substitute-path

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}
```

Parameters

path1

Specifies the existing search path.

path2

Specifies the replacement search path.

Example: Modifying search paths

```
set substitute-path "\src" "\My Src"      # Substitute "\src" with "\My Src"
```

Related information

[Files](#) on page 36

[Set](#) on page 46

2.3.161 set sysroot, set solib-absolute-prefix

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.

Syntax

```
set sysroot {path}
set solib-absolute-prefix {path}
```

Parameters

path

Specifies the system root directory.

Operation

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

Example: Setting system root directory

```
set sysroot "/mySystem"      # Set system root directory "/mySystem"
set solib-absolute-prefix "/mySystem" # Set system root directory "/mySystem"
```

Related information

[Operating System](#) on page 35

[Set](#) on page 46

2.3.162 set trust-ro-sections-for-opcodes

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}
```

Parameters

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.



Note

The Linux kernel often contains self-modifying code.

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. This is the default.

Example: Enabling reading of opcodes

```
set trust-ro-sections-for-opcodes on      # Enable reading opcodes from host
```

Related information

[Set](#) on page 46

2.3.163 set variable, set

Evaluates an expression and assigns the result to a variable, register, or memory address.

Syntax

```
set variable expression
```

Alias:

```
set var expression
```

Parameters**expression**

Specifies an expression and assigns the result to a variable, register, or memory address.

Example: Evaluating an expression

```
set variable myVar=10          # Assign 10 to variable myVar
set variable $PC=0x8000        # Set Program Counter to address 0x8000
set variable $CPSR.N=0         # Clear N bit of CPSR
set variable *0x8000=1         # Write 0x1 (one byte) to address 0x8000
set *(int*)0x8000=1            # Write 0x00000001 (four bytes) to 0x8000
set strcpy((char*)0x8000,"My String") # Assign string to address 0x8000
set memcpy((void *)0x80000100, (void *)0x80000000,8) # Copy 8 bytes from 0x80000000 to
                                                                    # 0x80000100
set memcpy((void *)0x80000000, "Hello",4) # Copy the first 4 characters of the
                                                                    # "Hello" string to 0x80000000
```

Example: Evaluating SME2 expressions

The following examples show access to the tiles available with *Scalable Matrix Extension*, SME2.

```
set $AARCH64::$SME::$Tiles::$ZA2H_S.F32[0][0] = 12.34
                                                                    # Set the zeroth element of the zeroth row of
                                                                    # tile ZA2 to 12.34. This can be abbreviated to:
                                                                    # set $ZA2H_S.F32[0][0] = 12.34
set $ZA0V_S.F32[$x5][$w1] = 12.34 # Set the w1th element of the x5th
                                                                    # column of tile ZA0 to 12.34
```

Related information

[Set](#) on page 46

[Built-in functions in Arm Debugger expressions](#) on page 17

2.3.164 set wildcard-style

Specifies the type of wildcard pattern matching you can use for examining the contents of strings.

Syntax

```
set wildcard-style {glob | regex}
```

Parameters

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: Specifying the type of wildcard pattern matching

```
set wildcard-style regex      # Use regular expression pattern matching
```

Related information

[Set](#) on page 46

2.3.165 sharedlibrary

Loads symbols from shared libraries. It can only load symbols for shared libraries that are already loaded by the application.

Syntax

```
sharedlibrary [expression]
```

Alias:

```
share [expression]
```

Parameters

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.

Operation

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

Example: Loading symbols from shared libraries

```
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).
```

Related information

[Operating System](#) on page 35

2.3.166 shell

Runs a shell command in the debug session. The command is launched in the working directory. You can use [pwd](#) to display the working directory.

Syntax

```
shell {cmd}
```

Parameters

cmd

Specifies the command and associated arguments.

Example: Running a shell command

```
shell dir              # On Windows, list files in directory.
shell cat my_script.ds # On Linux, list contents of my_script.ds file.
```

Related information

[Support](#) on page 51

2.3.167 show

Displays the debugger settings.

Syntax

```
show
```

Parameters

None.

Example: Showing debugger settings

```
show          # Display debugger settings.
```

Related information

[show](#) on page 48

2.3.168 show architecture

Displays the architecture of the target.

Syntax

```
show architecture
```

Parameters

None.

Example: Displaying the target architecture

```
show architecture    # Display target architecture.
```

Related information

[show](#) on page 48

[Support](#) on page 51

2.3.169 show arm

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

Syntax

```
show arm {option}
```

Parameters

option

Specifies additional options:

force-mode

Display the current force-mode behavior.

fallback-mode

Display the current fallback-mode behavior.

Example: Displaying the instruction set settings

```
show arm                # Display the instruction set settings.  
show arm force-mode     # Display the force-mode setting.
```

Related information

[show](#) on page 48

[Support](#) on page 51

2.3.170 show auto-solib-add

Displays the automatic setting for use when loading shared library symbols. You can use the `set auto-solib-add` command to modify this setting.

Syntax

```
show auto-solib-add
```

Parameters

None.

Operation

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

Example: Displaying automatic setting for loading library symbols

```
show auto-solib-add     # Display automatic setting for loading  
                        # shared library symbols.
```

Related information

[Operating System](#) on page 35

[show](#) on page 48

2.3.171 show backtrace

Displays the behavior settings for use with the `info stack` command. You can use the [set backtrace](#) command to modify these settings.

Syntax

```
show backtrace {option}
```

Parameters

option

Specifies additional options:

limit

Displays the limit when listing the call stack.

past-main

Shows whether the backtrace stops at `main()` or continues past `main()`.

Example: Displaying the call stack limit

```
show backtrace limit      # Display current call stack limit.
```

Related information

[Call stack](#) on page 34

[show](#) on page 48

2.3.172 show blocking-run-control

Displays the setting for blocking run control operations such as stepping and running. You can use the `set blocking-run-control` command to modify this setting.

Syntax

```
show blocking-run-control
```

Parameters

None.

Example: Displaying the run control setting

```
show blocking-run-control      # Display run control setting.
```

Related information

[Execution control](#) on page 31

[show](#) on page 48

2.3.173 show breakpoint

Displays the breakpoint and watchpoint behavior settings. You can use the `set breakpoint` commands to modify these settings.

Syntax

```
show breakpoint {option}
```

Parameters

option

Specifies additional options:

auto-hw

Displays the automatic breakpoint selection setting. This sets the type of breakpoint to use for the `break` command.

skipmode

Displays the breakpoint and watchpoint skipmode setting.

Example: Displaying the breakpoint and watchpoint behavior settings

```
show breakpoint auto-hw      # Display automatic breakpoint selection setting.
show breakpoint skipmode     # Display breakpoint and watchpoint skipmode setting.
```

Related information

[show](#) on page 48

2.3.174 show case-insensitive-source-matching

Displays the 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
```

Parameters

None.

Example: Displaying the case sensitivity setting

```
show case-insensitive-source-matching      # Display case sensitivity setting.
```

Related information

[show](#) on page 48

2.3.175 show cde-coprocessors

Displays the encoding associated with each coprocessor. Coprocessors are associated with either the *General CoProcessor* (GCP) encoding, or the *Custom Datapath Extension* (CDE) encoding.

Syntax

```
show cde-coprocessors
```

Parameters

None.

Example: Displaying the coprocessor encodings

In the following example, only the P0 coprocessor is associated with the CDE encoding. The other coprocessors are associated with the GCP encoding.

```
show cde-coprocessors
> P0 = CDE, P1 = GCP, P2 = GCP, P3 = GCP, P4 = GCP, P5 = GCP, P6 = GCP,
P7 = GCP
```

Related information

[show](#) on page 48

[set cde-coprocessors](#) on page 183

2.3.176 show debug-agent

Displays the value of an internal configuration parameter for the debug agent. You can use the `set debug-agent` command to modify this setting. The available parameters depend on the debug agent, such as `DSTREAM` or `gdbserver`.

Syntax

```
show debug-agent [name]
```

Parameters

name

Specifies the parameter to display.

Example: Displaying the debug agent configuration parameters

```
show debug-agent      # Display all debug agent configuration parameters.
```

Related information

[show](#) on page 48

2.3.177 show debug-from

Displays the 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
```

Parameters

None.

Example: Displaying the expression used by start command

```
show debug-from      # Display expression used by start command.
```

Related information

[Execution control](#) on page 31

[show](#) on page 48

2.3.178 show directories

Displays the list of directories to search for source files. You can use the `directory` command to modify this list.

Syntax

```
show directories
```

Alias:

```
show dir
```

Parameters

None.

Example: Displaying the list of search paths

```
show directories      # Display list of search paths.
```

Related information

[Files](#) on page 36

[show](#) on page 48

2.3.179 show dtssl-options

Displays the value of a parameter in the DTSL configuration. You can use the `set dtssl-options` command to modify this setting.

Syntax

```
show dtssl-options [name]
```

Parameters

name

Specifies the parameter to display.

Example: Displaying the DTSL configuration parameters

```
show dtssl-options      # Display all DTSL configuration parameters.
```

Related information

[show](#) on page 48

2.3.180 show dtssl-temporary-directory

Displays the current path for the temporary directory which stores trace data. You can modify the temporary directory path using the `set dtssl-temporary-directory` command.

Syntax

```
show dtssl-temporary-directory
```

Parameters

None.

Example: Displaying the current trace data temporary directory path

```
show dtssl-temporary-directory      # Shows the current trace data temporary  
                                     # directory path.
```

Related information

[show](#) on page 48

2.3.181 show elf cache-uninitialized-sections

Displays the debugger setting that controls whether uninitialized sections are cached.

Syntax

```
show elf cache-uninitialized-sections
```

Parameters

None.

Example: Displaying whether uninitialized sections are cached

```
show elf cache-uninitialized-sections      # Display whether uninitialized sections
                                           # are cached
```

Related information

[show](#) on page 48

2.3.182 show elf load-segments-at-p_paddr

Displays the debugger setting that controls the location for loading segments of ELF images.

Syntax

```
show elf load-segments-at-p_paddr
```

Parameters

None.

Example: Displaying whether ELF segment loaded at offset p_paddr

```
show elf load-segments-at-p_paddr      # Displays whether ELF segment
                                         # loading at offset p_paddr is
                                         # enabled or disabled (ELF segments
                                         # loaded at offset p_vaddr).
```

Related information

[show](#) on page 48

2.3.183 show elf zero-extra-segment-bytes

Displays the debugger setting that controls zeroing of bytes when loading segments of ELF images to the target.

Syntax

```
show elf zero-extra-segment-bytes
```

Parameters

None.

Example: Displaying if the zeros written when loading segments of ELF images

```
set elf zero-extra-segment-bytes      # Display whether the debugger writes zeros
                                         # if p_filesz is smaller than p_memsz.
```

Related information

[show](#) on page 48

2.3.184 show endian

Displays the byte order setting in use by the debugger. You can use the `set endian` command to modify this setting.

Syntax

```
show endian
```

Parameters

None.

Example: Displaying the byte order setting

```
show endian      # Display byte order setting.
```

Related information

[show](#) on page 48

[Support](#) on page 51

2.3.185 show escape-strings

Displays the 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
```

Parameters

None.

Example: Displaying the settings for printing special characters

```
show escape-strings      # Display setting for controlling how  
                          # special characters in strings are printed.
```

Related information

[show](#) on page 48

2.3.186 show escapes-in-filenames

Displays the 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
```

Parameters

None.

Example: Displaying the settings for special characters in paths

```
show escapes-in-filenames      # Display setting for controlling the use of  
                               # special characters in paths.
```

Related information

[show](#) on page 48

2.3.187 show idau-region

Displays the currently specified Implementation Defined Attribution Unit (IDAU) region parameters.

Syntax

```
show idau-region
```

Parameters

None.

Example: Displaying IDAU region parameters

```
show idau-region      # Display the currently specified IDAU region parameters.
```

2.3.188 show listsize

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
```

Parameters

None.

Example: Displaying listing size for the list command

```
show listsize      # Display listing size for list command.
```

Related information

[show](#) on page 48

2.3.189 show mmu use-cache-for-phys-reads

Displays the MMU setting that controls the coherency between translation table memory reads and cache data.

Syntax

```
show mmu use-cache-for-phys-reads
```

Parameters

None.

Example: Displaying the MMU coherency setting

```
show mmu use-cache-for-phys-reads      # Displays the MMU coherency setting.
```

Related information

[mmu](#) on page 41

[show](#) on page 48

[About debugging MMUs](#)

2.3.190 show os

Displays the Operating System (OS) control settings. You can use the `set os` command to modify these settings.

Syntax

```
show os {option}
```

Parameters

option

Specifies additional options:

enabled

Displays the setting for controlling OS support.

kernel-stack-size

Displays the stack size of the kernel.

log-capture

Displays the setting for controlling the capturing and printing of OS logging messages.

read-all-threads-on-stop

Displays the setting for the reading of threads when the target is stopped.

Operation

Before you can use this command, you must establish an OS aware connection.

Example: Displaying the OS control settings

```
show os log-capture      # Display setting for controlling os log capture.
show os enabled          # Display OS enabled setting.
```

Related information

[Operating System](#) on page 35

[show](#) on page 48

2.3.191 show print

Displays the debugger print settings. You can use the `set print` commands to modify these settings.

Syntax

```
show print {option}
```

Parameters

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: Displaying print settings

```
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.
```

Related information

[Display](#) on page 43

[show](#) on page 48

[Expressions in the Arm Debugger](#) on page 15

[printf\(\) style format string in Arm Debugger](#) on page 22

2.3.192 show semihosting

Displays the semihosting settings in the debugger. You can use the `set semihosting` commands to modify these settings.

Syntax

```
show semihosting {option}
```

Parameters

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.

policy

Displays the policy for the following semihosting operations on the host:

- Opening, renaming, and deleting files
- Running system commands

`block` specifies that the operations are not allowed and an error is displayed for each attempted operation.

`warn` specifies that the operations are allowed but a warning is displayed for each operation.

`allow` specifies that the operations are allowed.

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: Show semihosting settings

```
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.
```

Related information

[show](#) on page 48

[Support](#) on page 51

2.3.193 show solib-absolute-prefix

`show solib-absolute-prefix` is an alias for [show sysroot](#).

2.3.194 show solib-search-path

Displays the 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. You can also use the `set solib-search-path` command to specify additional directories.

Syntax

```
show solib-search-path
```

Parameters

None.

Operation

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

Example: Displaying the search path for shared libraries

```
show solib-search-path      # Display search path for shared libraries.
```

Related information

[Operating System](#) on page 35

[show](#) on page 48

2.3.195 show step-mode

Displays the step setting for functions without debug information. You can use the `set step-mode` command to modify this setting.

Syntax

```
show step-mode
```

Parameters

None.

Example: Displaying the step setting

```
show step-mode      # Display step setting (function without debug).
```

Related information

[Execution control](#) on page 31

[show](#) on page 48

2.3.196 show stop-on-solib-events

Displays the 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.

Syntax

```
show stop-on-solib-events
```

Parameters

None.

Operation

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

Example: Displaying the stop setting for shared library events

```
show stop-on-solib-events      # Display stop setting for shared library events.
```

Related information

[Operating System](#) on page 35

[show](#) on page 48

2.3.197 show substitute-path

Displays the 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
```

Parameters

None.

Example: Displaying all substitution rules

```
show substitute-path          # Display all substitution rules.
```

Related information

[Files](#) on page 36

[show](#) on page 48

2.3.198 show sysroot, show solib-absolute-prefix

Displays the system root directory in use by the debugger when searching for shared library symbols. You can use `set sysroot` or `set solib-absolute-prefix` 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.

Syntax

```
show sysroot
show solib-absolute-prefix
```

Parameters

None.

Operation

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

Example: Display the system root directory

```
show sysroot           # Display system root directory.
show solib-absolute-prefix # Display system root directory.
```

Related information

[Operating System](#) on page 35

[show](#) on page 48

[set sysroot](#), [set solib-absolute-prefix](#) on page 202

2.3.199 show trust-ro-sections-for-opcodes

Displays the 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
```

Parameters

None.

Example: Displaying the trust-ro-sections-for-opcodes setting

```
show trust-ro-sections-for-opcodes    # Display trust-ro-sections-for-opcodes
                                     # setting.
```

Related information

[show](#) on page 48

2.3.200 show version

Displays the version number of the debugger.

Syntax

```
show version
```

Parameters

None.

Example: Displaying the debugger version number

```
show version    # Display debugger version number.
```

Related information

[show](#) on page 48

[Support](#) on page 51

2.3.201 show wildcard-style

Displays the wildcard style for pattern matching. You can use the `set wildcard-style` command to modify this setting.

Syntax

```
show wildcard-style
```

Parameters

None.

Example: Displaying the wildcard style

```
show wildcard-style    # Display wildcard style.
```

Related information

[show](#) on page 48

2.3.202 silence

Disables the printing of stop messages for a specific breakpoint.

Syntax

```
silence [number]
```

Parameters

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: Disabling stop messages

```
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.
```

Related information

[Breakpoints and watchpoints](#) on page 29

2.3.203 snapshot

Generates files that contain a static snapshot of the target connection. A snapshot is useful for debugging an application when interactive debugging is not possible.

The snapshot file produced by this command is intended to be viewed and debugged using the Arm® Development Studio Snapshot Viewer. For more information, see [Working with the Snapshot Viewer](#). If you require snapshot data for use in other applications, use the [trace dump](#) command.

Syntax

```
snapshot {output-directory} [snapshot-script]
```

Parameters

output-directory

Specifies the full path to the directory where `snapshot` generates the snapshot files.

snapshot-script

Specifies the full path to a `snapshot.py` file to run a custom script. Use a custom script to specify and change memory parameters. For example, you can specify particular regions of memory to dump, and change the memory range.

If unchanged, the default memory range is 8 KB, centered on the core *Program Counter* (PC).

Outputs

snapshot outputs the following files to the specified directory:

- device<n>.ini files for each device on the target. Where:
 - n**
Refers to the number of the device, starting with 1. For example, if you have two devices, snapshot generates device1.ini and device2.ini.

Each file contains the following data:

- Device name
- Register values
- Reference to the relevant mem_<device_name>.bin file.
- mem_<device_name>.bin memory files for each device on the target.
- A snapshot.ini file consisting of one or more sections that emulate the state of the target at one point in time. This file refers to the device and trace .ini files. Use this file as the Snapshot Viewer initialization file to connect to the snapshot.

This file contains the following data:

- Register Values
- Memory Values
- Debug Symbols

This data enables debugging of an application using the Snapshot Viewer. For more information, see [Components of a Snapshot Viewer initialization file](#).

- A trace.ini file containing the trace metadata.
- If trace is enabled, a <trace_device_name>.bin file of the collected trace data.

Operation

If Arm Debugger fails to read the memory block, then the memory range halves until it is either successful, or the range reaches 8 bytes. At this point the script fails and returns an error. For example, if Arm Debugger fails to read the default memory block of 8 KB, then the memory range halves to 4 KB, and then to 2 KB, and so on, until the memory is successfully read or the range reaches 8 bytes.

Example: Generating snapshot files

```
snapshot snapshot_output # Generates the snapshot files in a
                        # directory named snapshot_output
```

Related information

[Working with the Snapshot Viewer](#)

[Connect to the Snapshot Viewer from Arm Development Studio](#)

[Connect to the Snapshot Viewer from the Arm Debugger command-line](#)

2.3.204 source

Loads and runs a script file to control and debug your target.

The following types of script are available:

Arm Debugger

Arm® Debugger commands.

CMM

CMM is a scripting language supported by some third-party debuggers. Arm Debugger 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]
```

Parameters

v

Specifies verbose output. Script commands are interleaved with the debugger output.

filename

Specifies the script file. Use these file extensions to identify the script type:

.ds

Specifies Arm Debugger scripts.

.cmm, .t32

Specifies for CMM scripts.

.py

Specifies Jython scripts.

args

Specifies the number of arguments (zero or more) to pass to the script (only supported for Jython scripts).

Restrictions

When you execute commands from a script, the debugger views do not update.

Example: Running commands from a file

```
source myScripts\myFile.ds      # Run Arm 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.t32.
```

```
source /v myFile.ds          # Run Arm Debugger commands from myFile.ds and
                             # display commands interleaved with debugger
                             # output.
source myScripts\myFile.py   # Run a Jython script from file myFile.py.
```

Related information

[Scripts](#) on page 33

2.3.205 start

Sets a temporary breakpoint, calls the debugger `run` command, and then deletes the temporary breakpoint when it is hit. By default, the temporary breakpoint is set at the address of the global function `main()`.

Syntax

```
start[args]
```

Parameters

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.

Operation

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.

Example: Running the target to a temporary breakpoint

```
start                        # Start running the target to the
                             # temporary breakpoint.
```

Related information

[Execution control](#) on page 31

2.3.206 stdin

Specifies semihosting input requested by application code.

Syntax

```
stdin [input]
```

Parameters

input

Specifies semihosting input requested by application code. This must be terminated by `\n` to tell the debugger that the input is complete.

Operation

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.



This command is not required if you launch the debugger in Arm® Development Studio, or if you use a telnet session to interact directly with the application.

Example: Specifying semihosting input

```
stdin 10000\n      # Pass the number 10000 to the application.
```

Related information

[Support](#) on page 51

2.3.207 step

Steps through an application at the source level stopping on the first instruction of each source line including stepping into all function calls. You can modify the behavior of this command with the `set step-mode` command.

Syntax

```
step [count]
```

Alias:

```
s [count]
```

Parameters

count

Specifies the number of source lines to execute.

Operation

You must compile your code with debug information to use this command successfully.

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

Example: Stepping through code lines

```
step                # Execute one source line.  
step 5             # Execute five source lines.
```

Related information

[stepi](#) on page 230

[steps](#) on page 231

[next](#) on page 160

[nexti](#) on page 161

[nexts](#) on page 162

[Execution control](#) on page 31

2.3.208 stepi

Steps through an application at the instruction level including stepping into all function calls.

Syntax

```
stepi[count]
```

Alias:

```
si [count]
```

Parameters

count

Specifies the number of instructions to execute.

Operation

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

Example: Stepping through instructions

```
stepi              # Execute one instruction.
```

```
stepi 5                                # Execute five instructions.
```

Related information

[step](#) on page 229

[steps](#) on page 231

[next](#) on page 160

[nexti](#) on page 161

[nexts](#) on page 162

[Execution control](#) on page 31

2.3.209 steps

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 can modify the behavior of this command with the `set step-mode` command.

Syntax

```
steps [count]
```

Alias:

```
ss [count]
```

Parameters

count

Specifies the number of source statements to execute.

Operation

You must compile your code with debug information to use this command successfully.

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

Example: Stepping through statements

```
steps                                # Execute one source statement.  
steps 5                             # Execute five source statements.
```

Related information

[step](#) on page 229

[stepi](#) on page 230

[next](#) on page 160

[nexti](#) on page 161

[nexts](#) on page 162

[Execution control](#) on page 31

2.3.210 stop

`stop` is an alias for [interrupt](#).

2.3.211 symbol-file

`symbol-file` is an alias for [file](#).

2.3.212 tbreak

Sets an execution breakpoint at a specific location and deletes the breakpoint when it 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.

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

Syntax

```
tbreak [-d] [-p] [[filename:]<location><address>] [[threadcore] <number>...] [if  
<expression>]
```

Parameters

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

A line number.

function

A function name.

label

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 PC.

Restrictions

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

Example: Setting an execution breakpoint at a specific location

```
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.
```

Related information

[break](#) on page 70

[hbreak](#) on page 104

[thbreak](#) on page 233

[resolve](#) on page 172

[clear](#) on page 78

[Breakpoints and watchpoints](#) on page 29

2.3.213 thbreak

Sets a hardware execution breakpoint at a specific location and deletes the breakpoint when it 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.

You can use `info breakpoints capabilities` to display a list of parameters that you can use with breakpoint commands for the current connection.

Syntax

```
thbreak [-d] [-p] [[filename:]location | *<address>] [[thread | core] <number>...]
[vmid <vmid>] [context <contextid>] [if <expression>]
```

Parameters

-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.

<contextid>

Specifies the context ID to apply the breakpoint to. This can be either an integer or an expression that evaluates to an integer. You can only use the context parameter if your hardware supports it and your application makes use of the CONTEXTIDR register.

<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.

Restrictions

- The number of hardware breakpoints is usually limited. If you run out of hardware breakpoints, then delete or disable one that you no longer use.
- Breakpoints that are set in a shared object or kernel module become pending when the shared object or kernel module is unloaded.

Example: Setting a hardware execution breakpoint at a specific location

```
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
thbreak context 257 0x80000000 # Set conditional breakpoint at address 0x80000000
                               # that stops when CONTEXTIDR=257
```

Related information

[break](#) on page 70

[hbreak](#) on page 104

[tbreak](#) on page 232

[resolve](#) on page 172

[clear](#) on page 78

[Breakpoints and watchpoints](#) on page 29

2.3.214 thread

Displays information about the current thread.

Syntax

```
thread [id]
```

Parameters

id

Specifies the unique thread number.

If `id` is not specified, then the debugger switches control to the current thread before displaying information. You can use `info processes` or `info threads` to display the `id` numbers.

If `id` is specified, then the debugger switches control to that thread before displaying the information. Registers and call stacks are associated with a particular thread. This means that switching context also switches the registers and call stack to those belonging to the current thread.

Outputs

This command displays:

- The unique `id` number assigned by the debugger.
- The thread or processor state. For example, `stopped` or `running`.
- The current stack frame, including function names and source line numbers.



`thread` and `core` are aliases, but give different outputs.

Example: Displaying thread information

```
thread 699          # Set current thread to number 699
                    # and display information about thread 699.
```

Related information

[Execution control](#) on page 31

[Operating System](#) on page 35

[Expressions in the Arm Debugger](#) on page 15

2.3.215 thread apply

Switches control to a specific thread to execute a 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}
```

Parameters

all

Specifies all threads.

id

Specifies the unique thread number. You can use `info processes` or `info threads` to display the `id` numbers.

command

Specifies the debugger command that you want to execute.

If `all` is specified then the command executes on each thread successively before switching back.

Example: Switching control to a thread

```
thread apply all print /x $pc          # Cycle through all threads and print address  
                                       # in PC register (hexadecimal).
```

Related information

[Execution control](#) on page 31

[Operating System](#) on page 35

2.3.216 trace clear

Clears the trace on the specified trace capture device. If no device is specified, clears the trace on all connected trace capture devices.

Syntax

```
trace clear [trace_capture_device]
```

Parameters

trace_capture_device

Specifies the trace capture device.

If no `trace_capture_device` is specified, then all trace capture devices are cleared.

Restrictions

Trace capture devices do not support clearing while capture is active.

Example: Clearing trace

```
trace clear          # Clears all connected trace capture devices.
trace clear ETB      # Clears trace capture device named ETB.
```

Related information

[Tracing](#) on page 33

2.3.217 trace dump

Dumps raw trace data to a directory, along with target trace configuration metadata, from a trace capture device or a trace source.

Syntax

```
trace dump {output_path} [-<option>] [trace_capture_device | trace_source]...
```

Parameters

output_path

Specifies the destination of the trace dump. It creates a directory named `output_path`. It creates the metadata and trace data in this directory. It generates an error if this directory already exists.



Note

If you specify a folder name only or a relative path, then it creates the output directory in, or relative to, the current working directory.

-<option>

Where <option> is one of:

raw

Dumps raw data. Raw data is the captured trace data with trace device specific formatting. The raw option only applies to trace capture devices.

no_metadata

Suppresses the metadata.

no_tracedata

Suppresses the trace data.

split_file_size=<value>

Specifies the maximum file size (in bytes) of the trace data files generated by the `trace dump` command. If the size of the file exceeds this amount, a new trace data file is

generated. Specify -1 to keep trace data in a single file. Default value is 1073741824. Minimum value is 65536.

trace_capture_device

Specifies the trace capture device.

trace_source

Specifies a trace source.

Operation

If no `trace_capture_device` or `trace_source` is specified, then all trace capture device buffers are dumped.

If a trace capture device is specified and a trace source from that device is also specified then the trace data for that source is dumped twice. First into the complete buffer for the device and again as a dump of just the specified trace source.

Example: Dumping trace data and configuration metadata

```
trace dump TraceDump
# Creates a directory named TraceDump. Dumps the buffers of all active
# trace capture devices into TraceDump, along with the metadata
# describing them.

trace dump TraceDump ETB
# ETB is the name of a trace capture device. Dumps the contents of the
# ETB buffer to TraceDump.

trace dump TraceDump DSTREAM -raw
# DSTREAM is the name of a trace capture device. Dumps the contents of
# the DSTREAM buffer to TraceDump in raw format.

trace dump TraceDump PTM 1
# PTM_1 is the name of a trace source. Extracts the trace data for PTM_1
# from the trace device buffer and dumps it to TraceDump.

trace dump TraceDump ETB -no_metadata
# Dumps the contents of the ETB buffer to TraceDump, but does not write
# the metadata.

trace dump TraceDump ETB -no_tracedata
# Writes the metadata for ETB in TraceDump, but does not write the trace
# data.

trace dump TraceDump ETB -no_tracedata -no_metadata
# Creates an empty directory named TraceDump.
```

Related information

[Tracing](#) on page 33

2.3.218 trace info

Displays details about trace capture devices and trace sources.

Syntax

```
trace info [option] [trace_capture_device | trace_source]
```

Parameters

option

Specifies how information is displayed:

-showdisabled

Displays disabled devices and sources.

trace_capture_device

Specifies the trace capture device.

trace_source

Specifies the trace capture source.

Operation

If no `trace_capture_device` or `trace_source` is specified, then all trace capture devices and sources are displayed.

Example: Displaying trace capture and sources details

```
trace info
# Display all the enabled trace capture devices and trace sources.

trace info -showdisabled
# Display all trace capture devices and trace sources including disabled ones.

trace info ETB
# Display the trace capture device or trace source named ETB.
```

Related information

[Tracing](#) on page 33

[show dtsl-temporary-directory](#) on page 213

[set dtsl-temporary-directory](#) on page 186

2.3.219 trace list

Lists the trace capture devices and trace sources.

Syntax

```
trace list
```


Parameters

None.

Example: Listing trace devices and sources

```
trace list      # List all of the trace capture devices and trace sources
```

Related information

[Tracing](#) on page 33

2.3.220 trace report

Produces a trace report, containing the decoded trace data, for the currently selected core.

Syntax

```
trace report [option = <value>]...
```

Parameters

option

Specifies the name of a trace report option to set.

<value>

Specifies the new value of the option.

The option names are not case sensitive. The options are:

OUTPUT_PATH

Specifies the directory to save the trace report files in. The default value is the current working directory.

FILE

Specifies the base file name of the trace report. If trace report generates multiple files, then each file will have a zero-padded number inserted before the file name extension. The default value is `Trace_Report.txt`.

SPLIT_FILE_SIZE

Specifies the maximum file size, in bytes, that trace report generates. If the file size is larger than `SPLIT_FILE_SIZE`, trace report generates a new report file. Specifying `-1` indicates that there is no maximum file size, so the trace report is not split into separate files. The default value is `1073741824`.

START

Specifies the position in the trace buffer to start decoding trace from. The default value is `0`, which starts the decoding from the beginning of the buffer.

END

Specifies the position in the trace buffer to stop decoding trace. Specifying `-1` indicates that the trace report should decode to the end of the buffer. The default value is `-1`.

FORMAT

Specifies the format of the report. Valid values are `csv` (Comma-Separated Values) and `tsv` (Tab Separated Values). The default value is `tsv`. Format values are not case sensitive.

SOURCE

Specifies the trace source to report. Execute the `trace list` command to view the list of available trace sources. The default is to dump the trace source associated with the current core.

CORE

Specifies the core to report. Execute the `info cores` command to view the list of cores available. This option is analogous to the `SOURCE` option, except that the source for the given core will be discovered automatically. You can specify either a `SOURCE` or `CORE` but not both.

CONFIG

Specifies a configuration file. This is used to specify decoding details for STM and ITM trace sources. The default configuration is to decode all Ports, Transmitters, and Channels as binary data. This file is created by exporting it from the `Event Viewer Settings` dialog box.

COLUMNS

Specifies a comma separated list of columns to include in the report. The column names are not case sensitive.

Valid values for instruction trace sources are:

RECORD_TYPE

The type of the record.

INDEX

The index of the instruction. Canceled instructions do not have an index.

ADDRESS

The address of the instruction.

OPCODE

The opcode of the instruction, in hexadecimal, with no prefix.

OPCODE_WITH_PREFIX

The opcode of the instruction, in hexadecimal, with a `0x` prefix.

CYCLES

The cycle count of the instruction.

DETAIL

For instruction records, this gives the disassembly of the instruction. For other record types, this gives various information.

FUNCTION

The function of the instruction.

BRANCH

This is true if the instruction is a branch. Otherwise, this is false.

For instruction trace sources, the default is `ADDRESS`, `OPCODE`, `DETAIL`.

Valid values for STM trace sources are:

TRANSMITTER

The transmitter number can be 0 to 128.

CHANNEL

The channel number can be 0 to 65535.

TIMESTAMP

An approximate timestamp for each record, if available.

SIZE

Size of the row in bytes.

DATA

The row data.

For STM trace sources, the default is `TRANSMITTER`, `CHANNEL`, `DATA`.

Valid values for ITM trace sources are:

PORT

The port number can be 0 to 255.

TIMESTAMP

The global timestamp for the record, if available (M-profile only). This column name is synonymous with the global time stamp (GTS).

DATA

The row data.

LTS

The local timestamp for the record, if available.

GTS

The global timestamp for the record, if available (M-profile only).

COMP

For DWT data trace packets, the number of the matching DWT comparator (M-profile only). This column is only useful if the DWT option is specified as true.

For ITM trace sources, the default is `PORT`, `DATA`.

DWT

For M-profile ITM trace sources, specifies whether to include DWT packets in the report. The default value is false. To include DWT packets, specify true.

PORTS

For ITM trace sources, specifies a comma-separated list of stimulus ports to include. Output from stimulus ports not listed is suppressed from the report. If the option is not present, output from all stimulus ports is included.

DECODERS

For ITM trace sources, specifies a comma-separated list of decoder assignments. Each decoder assignment has the form `P<n>:<decoder_name>` where `<n>` is a stimulus port number, and `<decoder_name>` is one of the names available in the `Encoding` drop-down list in the [Event Viewer Settings](#) dialog box. The decoders available by default are `TAE`, `Text`, and `Binary`. If no decoder is assigned to a stimulus port, the default is `Binary`.

HEADERS

Specifies whether to include the column headers in the report. The default value is `false`. To include headers, specify `true`.

Example: Producing a trace report

```
trace report
# Produces a default trace report named "Trace_Report.txt" in the
# current working directory.
# Instruction trace for the current core is reported.

trace report FILE=MyReport.csv OUTPUT_PATH=C:/files/trace_reports FORMAT=CSV
# Produces a comma-separated value trace report named "MyReport.csv"
# in C:/files/trace_reports.

trace report COLUMNS=RECORD_TYPE,INDEX,ADDRESS,OPCODE_WITH_PREFIX,DETAIL
HEADERS=true
# Produces a trace report with alternate columns.
# The first line of the report contains the column names.

trace report SOURCE=ITM COLUMNS=PORT,DATA HEADERS=true
# Produces an ITM trace report with alternate columns.
# The first line of the report contains the column names.

trace report SOURCE=ITM PORTS=1,2 DECODERS=P1:Text,P2:TAE HEADERS=true
# Specifies custom decoders for stimulus ports 1 and 2, and suppresses
# output from all other stimulus ports.
# The first line of the report contains the column names.

trace report SOURCE=CSITM DWT=true COLUMNS=PORT,COMP,DATA HEADERS=true
# Produces an ITM trace report with DWT packets included, and DWT
# comparator numbers for data trace packets.
# The first line of the report contains the column names.
```

Related information

[Tracing](#) on page 33

2.3.221 trace start

Starts trace capture on the specified trace capture device. If no device is specified, starts trace capture on all connected trace capture devices.

Syntax

```
trace start [trace_capture_device]
```

Parameters

trace_capture_device

Specifies the trace capture device.

If no `trace_capture_device` is specified, then all trace capture devices are started.

Example: Starting trace capture

```
trace start          # starts all connected trace capture devices
trace start ETB      # starts trace capture device named ETB
```

Related information

[Tracing](#) on page 33

2.3.222 trace stop

Stops trace capture on the specified trace capture device. If no device is specified, stops trace capture on all connected trace capture devices.

Syntax

```
trace stop [trace_capture_device]
```

Parameters

trace_capture_device

Specifies the trace capture device.

If no `trace_capture_device` is specified, then all trace capture devices are stopped.

Example: Stopping trace capture

```
trace stop          # stops all connected trace capture devices
trace stop ETB      # stops trace capture device named ETB
```

Related information

[Tracing](#) on page 33

2.3.223 unset

Modifies the current debugger settings.

Syntax

```
unset {option}
```

Parameters

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: Modifying debugger settings

```
unset substitute-path      # Delete all substitution paths
```

Related information

[Support](#) on page 51

2.3.224 **unsilence**

Enables the printing of stop messages for a specific breakpoint.

Syntax

```
unsilence [number]
```

Parameters

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: Enabling stop messages for a breakpoint

```
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
```

Related information

[Breakpoints and watchpoints](#) on page 29

2.3.225 up

Moves and displays 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.

Syntax

```
up [offset]
```

Parameters

offset

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

Operation

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.

Example: Moving the frame pointer up the call stack

```
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
```

Related information

[Call stack](#) on page 34

2.3.226 up-silently

Moves the current frame pointer up the call stack towards the top frame.

Syntax

```
up-silently [offset]
```

Parameters

offset

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

Operation

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.

Example: Moving the frame pointer up the call stack

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

Related information

[Call stack](#) on page 34

2.3.227 usecase help

Displays help for a use case script.

The command prints information about the use case script and gives a list of options that can be provided when invoking the script.

Syntax

```
usecase help [flag] {script_name} [entry_point]
```

Parameters

flag

Specifies the location of the use case script. This can be one of:

-p

The directory associated with the current platform in the Arm® Debugger Configuration databases.

-s

The `Scripts\usecase` directory in the Arm Debugger Configuration databases.

script_name

Name of the use case script to print help for.

entry_point

Specifies a named entry point in the use case script. If there is only one entry point defined in the use case script, it is not necessary to specify the entry point on the command line. If the use case script contains more than one entry point, then you must specify which one to use, as a parameter to this command.

Example: Displaying help for a use case script

```
usecase help script.py
# Print help for script.py from the current working directory

usecase help -p db_script.py
# Print help for db_script.py from the current platform directory
```



```
usecase help multi_usecase.py mainOne
# Print help for the mainOne entry point in multi_usecase.py

usecase help multi_usecase.py mainTwo
# Print help for the mainTwo entry point in multi_usecase.py
```

Related information

[Scripts](#) on page 33

2.3.228 usecase list

Lists use case scripts.

By default, the command lists all the use case scripts in the current working directory.

Syntax

```
usecase list [-p | -s | -a | directory]
```

Parameters

-p

Lists all the use case scripts associated with the current platform. The use case scripts must be in the same directory where the DTSL scripts and .xvc file for the current platform are stored in the Arm® Debugger Configuration databases.

-s

Lists all the use case scripts in the scripts\\usecase directory in the Arm Debugger Configuration databases.

-a

Lists all the use case scripts that are in any of these categories:

- In the current working directory.
- Associated with the current platform.
- In the scripts directory in the Arm Debugger Configuration databases.

directory

Lists all the use case scripts in the specified directory.

Example: Listing use case scripts

```
usecase list
# Lists all the use case scripts in the current working directory

usecase list -p
# Lists all the use case scripts for the current platform

usecase list -s
# Lists all the use case scripts in the Scripts\usecase folder in the
# Arm DS Configuration databases

usecase list c:\usecase\scripts
```

```
# Lists all the use case scripts in c:\usecase\scripts
usecase list scripts
# Lists all the use case scripts in the scripts folder in the current
# working directory
```

Related information

[Scripts](#) on page 33

2.3.229 usecase run

Runs a use case script.

Syntax

```
usecase run [flag] {script_name} [entry_point] [--<option> | positional_argument]...
```

Parameters

flag

Specifies the location of the use case script. This can be one of:

-p

The directory associated with the current platform in the Arm® Debugger Configuration databases.

-s

The `scripts\usecase` directory in the Arm Debugger Configuration databases.

script_name

Name of the use case script to run.

entry_point

Specifies a named entry point in the use case script. If there is only one entry point defined in the use case script, it is not necessary to specify the entry point on the command line. If the use case script contains more than one entry point, then you must specify which one to use, as a parameter to this command.

--<option>

Specifies a named option defined in the use case script and its value. You can specify more than one `<option>`.

positional_argument

Specifies a positional argument to the entry point. You can specify more than one `positional_argument`.

Example: Running a use case script

```
usecase run myscript.py
# Runs a script named myscript.py in the current directory

usecase run -p platform_script.py entry
# Runs platform_script.py in the current platform directory in the
```

```
# Arm DS Configuration database, with entry point set to entry
usecase run -s db_script.py --opts.x=1
# Runs db_script.py in the Scripts\usecase directory in the Arm DS
# Configuration database, with the option opt.x defined as 1

usecase run second_script.py main x y z
# Runs second_script.py passing in x, y, and z as positional arguments
# to the entry point main

usecase run -s myscript.py --cores=4 --target="run" t.txt
# Runs myscript.py in the Scripts\usecase directory with options cores
# and target and a positional argument t.txt
```

Related information

[Scripts](#) on page 33

2.3.230 wait

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]]
```

Parameters

time-out

Specifies the period of time.

ms

Specifies the time in milliseconds. This is the default.

s

Specifies the time in seconds.

Example: Waiting until the target stops

```
wait 1000          # Wait or time-out after 1 second
wait 0.5s          # Wait or time-out after half a second
```

Related information

[Execution control](#) on page 31

[advance](#) on page 62

2.3.231 watch

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.

Watchpoints are supported on single memory addresses for all processors, on hardware and models. For M-class Arm processor hardware, watchpoints are also supported on structs and arrays, and arbitrary memory ranges.

The availability of watchpoints depends on your target. In the case of Linux application debug using gdbserver, the availability of watchpoints also depends on the Linux kernel version and configuration.

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] [-w <width> | -m <mask> | -s <size>] [-f] {[filename:]symbol |
* <address>}] [vmid <number>] [if <condition>]
```

Parameters

-d

Creates the watchpoint disabled.

-p

Specifies whether or not the resolution of an unrecognized watchpoint location results in a pending watchpoint being created.

-w <width>

Specifies the width to watch at the given address, in bits. Accepted values are: 8, 16, 32, and 64 if supported by the target. This parameter is optional.

The width defaults to:

- 32 bits for an address.
- The width corresponding to the type of the symbol or expression, if entered.

-m <mask>

This option is only available for Arm®v6-M, Armv7-M, Armv8-M, Armv8-R, Armv8-A, and Armv9-A hardware.

Specifies a mask that represents a memory range. The mask value is the number of least significant bits that will be masked. For example, a value of 5 would watch a range of 32 bytes.

**Note**

The range could be widened if the watched address is not specified as a power of 2.

-s <size>

This option is only available for Armv6-M, Armv7-M, Armv8-M, Armv8-R, Armv8-A, and Armv9-A hardware.

Specifies the size of a memory area to watch. If the <size> and *<address> are not specified so that the range is exactly coverable by an address mask, you must set the -f option.

-f

This option is only available for Armv6-M, Armv7-M, Armv8-M, Armv8-R, Armv8-A, and Armv9-A hardware.

You must set this parameter if the hardware does not support arbitrary ranges and the range described by *<address> and <size> is not exactly coverable by an address mask. If set, a mask is used that is inclusive of the range specified by <size>, but may extend beyond that range. If the mask extends beyond the range specified by <size>, the Arm Debugger could halt if there are accesses outside of the intended range.

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 option can be either an address or an expression that evaluates to an address.

vmid <number>

Specifies the Virtual Machine ID (VMID) to apply the watchpoint to. This option can be either an integer or an expression that evaluates to an integer. Applicable only on targets which support hypervisor / virtual machine debugging.

if <condition>

Specifies the condition which must evaluate to true at the time the watchpoint is triggered for the target to stop. You can create several conditional watchpoints, but when a conditional watchpoint is enabled, no other watchpoints (regardless of whether they are conditional) can be enabled.

Example: Setting a watchpoint for a symbol

```
watch myVar1          # Set write watchpoint on myVar1
watch *0x80D4         # Set write watchpoint on
                      # address 0x80D4
watch myVar1 if myVar1 == 2 # Set write watchpoint on myVar1
                      # which is hit only if myVar1
                      # evaluates to 2
```

```

watch myVar1 if ($LR & 0xFF) == 0x12      # Set write watchpoint on myVar1
                                           # which is hit only if ($LR & 0xFF)
                                           # evaluates to 0x12 when myVar1
                                           # is accessed
watch 0x500884D0 if *((int*) 0x500884D0) == 0  # On Armv6-M, Armv7-M, Armv8-A, and
                                           # Armv8-R hardware, set a write
                                           # watchpoint to stop execution
                                           # when a zero is written with
                                           # a 4-byte data width to a specific
                                           # address
watch -s 200 *0x80D4                      # On Armv6-M, Armv7-M, Armv8-A, and
                                           # Armv8-R hardware, set a write
                                           # watchpoint on address 0x80D4 with
                                           # a range of 200 bytes
watch -m 5 *0x8000                        # On Armv6-M, Armv7-M, Armv8-A, and
                                           # Armv8-R hardware, set a write
                                           # watchpoint on address 0x8000
                                           # with a range of 32 bytes

```

Related information

[rwatch](#) on page 175

[clearwatch](#) on page 79

[awatch](#) on page 67

[Breakpoints and watchpoints](#) on page 29

2.3.232 watch-set-property

Updates the properties of an existing watchpoint.

Syntax

```
watch-set-property {number} {property}
```

Parameters

number

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

property

Specifies the property to set. The valid properties are:

if [expression]

Specifies an expression that is evaluated when the watchpoint is hit. If the value of the expression evaluates to true, then the debugger stops the target, otherwise execution resumes. If no expression is specified then the watchpoint condition is deleted.

data-width [bits]

Specifies the width to watch at the given address, in bits. Accepted values are: 8, 16, 32, and 64 if supported by the target. This parameter is optional.

The width defaults to:

- 32 bits for an address.

- The width corresponding to the type of the symbol or expression, if entered.

Other target-dependent properties

This command supports other types of `property` depending on your target. Use the [info watchpoints capabilities](#) command to display a list of `property` types that you can use for the current connection.

Example: Updating a watchpoint property

```
watch-set-property 4 if myVar1 == 2    # Update the 'if' property of watchpoint 4,  
                                       # meaning the watchpoint will only be hit if  
                                       # myVar1 evaluates to 2
```

2.3.233 `whatis`

Displays the data type of an expression.

Syntax

```
whatis [expression]
```

Parameters

expression

Specifies an expression. If no `expression` is specified then the last expression is repeated.

Operation

This command does not execute the expression.

Example: Displaying an expression data type

```
whatis 4+4                # Display data type of expression result  
whatis myVar              # Display data type of variable (myVar)
```

2.3.234 `where`

`where` is an alias for [info stack and backtrace](#).

2.3.235 while

Allows you to write scripts with conditional loops that execute debugger commands.

Syntax

```
while {condition}
```

```
...
```

```
[optional_commands]
```

```
...
```

```
end
```

Parameters

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: 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
```

Related information

[Scripts](#) on page 33

2.3.236 x

Displays the content of memory at a specific address.

Syntax

```
x{/flag}...{/flag} [<width=<n>>:]{address}
```

Parameters

flag

Specifies additional flags:

count

Specifies the number of values to display. If nothing is specified, then the default is 1.

Size of data to be read:

b

One byte

h

Two bytes

w

Four bytes (default)

g

Eight 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

<width=<n>>

Specifies the access width in bits to use when accessing memory. If the width is narrower than the value being accessed, then more than one access is performed to access the value. If this parameter is not specified, access width defaults to 32-bit.

`number-of-bits` can be one of the following: 8, 16, 32, or 64.

address

Specifies the address. This can be either an address, a symbol name, or an expression that evaluates to an address. You can optionally specify an address space prefix for 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.

Restrictions

- If you specify either `x/b`, `x/h`, or `x/g`, and then in a later `x` command you remove the specified size, the debugger uses the previous size that you specified; it does not revert to the default size of `x/w`.
- The size of the memory accessed is not the same as the width the debugger uses to access the memory. Use the `<width>` parameter to specify the width, in bits, the debugger uses to access the memory.

Operation

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.

Example: Displaying content of memory

```
x 0x8000           # Display memory at address 0x8000
x/3wx 0x8000       # Display three words of memory from address 0x8000 in
                   # hexadecimal
x/4b $SP           # Display four bytes of memory from the address held
                   # in the SP register
x/4i $PC           # Display four instructions from the address held in
                   # the PC register
x /h 0x8000        # Read a half-word from address 0x8000
x/w 0x2D000000     # Display the four bytes at the address, using the
                   # default access width for the target
x/w <width=32>:0x2D000000 # Display the four bytes at the address, forcing the
                   # use of 32-bit accesses
x /w EL3<width=16>:0x80000000 # Display one word using two half-word accesses from
```

```
# EL3:0x80000000
```

Related information

[Memory group](#) on page 39

[Display](#) on page 43

[Expressions in the Arm Debugger](#) on page 15

[Address space prefixes](#) on page 24

3. CMM-style commands supported by the debugger

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.

To execute CMM-style commands you must create a debugger script file containing the CMM-style commands and then use the Arm® Debugger `source` command to run the script.



Note

For full debug support, Arm recommends that you use the Arm Debugger commands. See [Arm 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.

Using 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.

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 providing an alias.

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.

Using expressions with CMM-style commands

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.

3.1 CMM-style commands groups: All

Displays all the CMM-style commands by group.

Related information

[Controlling breakpoints](#) on page 261

[Controlling data and display settings](#) on page 262

[Controlling images, symbols, and libraries](#) on page 262

[Controlling target execution and connections](#) on page 262

[Displaying the call stack and associated variables](#) on page 263

[Controlling the debugger and program information](#) on page 263

[Supporting commands](#) on page 263

3.1.1 Controlling breakpoints

List of CMM-style commands that enable you to control the starting and stopping of the debugger using breakpoints.

break.delete

Deletes a breakpoint at the specified address.

break.disable

Disables a breakpoint at the specified address.

break.enable

Enables a breakpoint at the specified address.

break.set

Sets a software breakpoint at the specified address.

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

3.1.2 Controlling data and display settings

List of all the CMM-style commands that enable you to display specific output on the command-line.

data.dump

Displays data at a specific address or address range.

data.set

Writes data to memory.

print

Concatenates the results of one or more expressions.

register.set

Sets the value of a register.

var.global

Displays all global variables.

var.local

Displays all local variables in a function.

var.print

Concatenates the results of one or more expressions.

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

3.1.3 Controlling images, symbols, and libraries

List of all the CMM-style commands that enable you to load files:

data.load.binary

Loads a binary image file.

data.load.elf

Arm Executable and Linking Format (ELF) file.

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

3.1.4 Controlling target execution and connections

List of all the CMM-style commands that enable you to connect to a target:

break

Stops running the target.

go

Starts running the device.

system.down

Disconnects the debugger from the target.

system.up

Connects to the specified target.

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

3.1.5 Displaying the call stack and associated variables

List of all the CMM-style commands that enable you to display stacks and variables:

var.frame

Displays the stack frame.

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

3.1.6 Controlling the debugger and program information

List of all the CMM-style commands that enable you to control scripts:

var.new

Creates a new script variable and zero-initializes it. Script variables are for use at runtime only.

var.set

Sets and displays the value of an existing script variable.

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

3.1.7 Supporting commands

List of all the miscellaneous CMM-style commands

help

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

wait

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.

3.2 CMM-style commands listed in alphabetical order

Describes the CMM-style commands supported by the Arm® Debugger.

Table 3-1: CMM-style commands supported by the Arm Debugger

Debugger command	Description
CMM-style commands: break	Stops running the target.
CMM-style commands: break.delete	Deletes a breakpoint at the specified address.
CMM-style commands: break.disable	Disables a breakpoint at the specified address.
CMM-style commands: break.enable	Enables a breakpoint at the specified address.
CMM-style commands: break.set	Sets a software breakpoint at the specified address.
CMM-style commands: data.dump	Displays data at a specific address or address range.
CMM-style commands: data.load.binary	Loads a binary image file.
CMM-style commands: data.load.elf	Loads an Arm <i>Executable and Linking Format</i> (ELF) file.
CMM-style commands: data.set	Writes data to memory.
CMM-style commands: go	Starts running the device.
CMM-style commands: help	Displays help information for a specific command or a group of commands listed according to specific debugging tasks.
CMM-style commands: print	Concatenates the results of one or more expressions.
CMM-style commands: register.set	Sets the value of a register.
CMM-style commands: system.down	Disconnects the debugger from the target.
CMM-style commands: system.up	Connects to the specified target.
CMM-style commands: var.frame	Displays the stack frame.
CMM-style commands: var.global	Displays all global variables.
CMM-style commands: var.local	Displays all local variables in a function.
CMM-style commands: var.new	Creates a new script variable and zero-initializes it.
CMM-style commands: var.print	Concatenates the results of one or more expressions.
CMM-style commands: var.set	Sets and displays the value of an existing script variable, as well as the result of an expression.
CMM-style commands: wait	Pauses the execution of a script for a specified period of time.

3.2.1 CMM-style commands: break

Stops running the target.

Syntax

```
break
```

Alias:

```
b
```

Parameters

None.

Example: Stopping the target

```
break ; Stop running the target
```

3.2.2 CMM-style commands: break.delete

Deletes a breakpoint at the specified address.

Syntax

```
break.delete {expression}
```

Alias

```
b.d {expression}
```

Parameters

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: Deleting a breakpoint using various expressions

```
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()
```

3.2.3 CMM-style commands: break.disable

Disables a breakpoint at the specified address.

Syntax

```
break.disable {expression}
```

Alias

```
b.dis {expression}
```

Parameters

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: Disabling a breakpoint using various expressions

```
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()
```

3.2.4 CMM-style commands: break.enable

Enables a breakpoint at the specified address.

Syntax

```
break.enable {expression}
```

Alias:

```
b.en {expression}
```

Parameters

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: Enabling a breakpoint using various expressions

```
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()
```

3.2.5 CMM-style commands: break.set

Sets a software breakpoint at the specified address.

Syntax

```
break.set {expression} [/<flag>]
```

Alias:

```
b.s {expression} [/<flag>]
```

Parameters

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. `dis` is the alias for this flag.

Example: Setting a breakpoint using various expressions

```
break.set 0x8000      ; Set breakpoint at address 0x8000
break.set main        ; Set breakpoint at address of main()
break.set main+4      ; Set breakpoint 4 bytes after address of main()
break.set main\2      ; Set breakpoint 2 source lines after address of main()
```

3.2.6 CMM-style commands: data.dump

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>] ...
```

Alias:

```
d.dump {expression} [/<flag>] ...
```

Parameters

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. **b** is the alias for this flag.

word

Formats the data as 2 bytes. **wo** is the alias for this flag.

long

Formats the data as 4 bytes. **l** is the alias for this flag.

quad

Formats the data as 8 bytes. **q** is the alias for this flag.

width

Specifies the number of columns. **wi** is the alias for this flag.

nohex

Suppresses the hexadecimal output. **noh** is the alias for this flag.

noascii

Suppresses the ASCII output. **noa** is the alias for this flag.

le

Formats the data as little endian

be

Formats the data big endian.

Operation

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: Displaying data at various addresses or address ranges

```

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

```

3.2.7 CMM-style commands: data.load.binary

Loads a binary image file.

Syntax

```
data.load.binary {filename} [expression]
```

Alias:

```
d.load.b {filename} [expression]
```

Parameters

filename

Specifies the image file.

expression

Specifies the load address. This can be an address, a symbol name, or an expression that evaluates to an address. If no expression is specified, then the default is 0x0.

Operation

Loading a binary image does not change the program counter or any symbols that are currently loaded.

Example: Loading an image using various filenames and expressions

```
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
```

3.2.8 CMM-style commands: data.load.elf

Loads an Arm *Executable and Linking Format* (ELF) file. This format is described in the Arm ELF specification and uses the `.axf` file extension.

Syntax

```
data.load.elf {filename} [ /<flag> ] ...
```

Alias:

```
d.load.e {filename} [ /<flag> ] ...
```

Parameters

filename

Specifies the image file.

/<flag>

Specifies additional flags:

nocode

Do not load code and data to the target.

nosymbol

Do not load symbols. `nos` is the alias for this flag.

noclear

Symbol table is not cleared before loading the image. `noc` is the alias for this flag.

noreg

Do not set register values, for example, PC and status registers. `nor` is the alias for this flag.

Restrictions

Loading an ELF image sets the program counter to the entry point of the image, if present.

Operation

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: Loading images and symbols using various filenames

```
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
```

3.2.9 CMM-style commands: data.set

Writes data to memory.

Syntax

```
data.set {address} [%<format>] {expression} [/<flag>] ...
```

Alias:

```
d.s {address} [%<format>] {expression} [/<flag>] ...
```

Parameters**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. *b* is the alias for this flag.

word

Formats the data as 2 bytes. *w* is the alias for this flag.

long

Formats the data as 4 bytes. *l* is the alias for this flag.

quad

Formats the data as 8 bytes. *q* is the alias for this flag.

float.ieee

Formats the data as a 4 byte floating-point. *f*.*i* is the alias for this flag.

float.ieee dbl

Formats the data as an 8 byte floating-point. *f*.*a* is the alias for this flag.

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. *v* is the alias for this flag.

compare

Compares the data in memory but does not write to memory. *cp* is the alias for this flag.

Example: Writing to memory at various address or address ranges

```
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
```

3.2.10 CMM-style commands: go

Starts running the device.

Syntax

```
go
```

Alias:

```
g
```

Parameters

None.

Example: Starting a device

```
go ; Start running the device
```

3.2.11 CMM-style commands: help

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

Syntax

```
help [command | group]
```

Alias:

```
h [command | group]
```

Parameters

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: Displaying 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
```

3.2.12 CMM-style commands: print

Concatenates the results of one or more expressions.

Syntax

```
print [%<printing_format>] {expression}...
```

Parameters**<printing_format>**

Specifies one of [ascii | binary | decimal | hex] with aliases of [a | bin | d | h]. If none specified, then the default is decimal format.

expression

Specifies an expression that is evaluated and the result is returned.

Example: Displaying strings and variables

```
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
```

3.2.13 CMM-style commands: register.set

Sets the value of a register.

Syntax

```
register.set {name} {expression}
```

Alias:

```
r.s {name} {expression}
```

Parameters

name

Specifies the name of a register.

expression

Specifies an expression that is evaluated and the result assigned to a register.

Example: Setting register values

```
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()
```

3.2.14 CMM-style commands: system.down

Disconnects the debugger from the target.

Syntax

```
system.down
```

Alias:

```
sys.d
```

Parameters

None.

Example: Disconnecting from a target

```
system.down ; Disconnect from target
```

3.2.15 CMM-style commands: system.up

Connects to the specified target.

Syntax

```
system.up
```

Alias:

```
sys.u
```

Parameters

None.

Example: Connecting to a target

```
system.up ; Connect to target
```

3.2.16 CMM-style commands: var.frame

Displays the stack frame.

Syntax

```
var.frame [%<printing_format>] [/<flag>] ...
```

Alias:

```
v.f [%<printing_format>] [/<flag>] ...
```

Parameters

%<printing_format>

Specifies one of [ascii | binary | decimal | hex] with aliases of [a | bin | d | h]. If none specified, then the default is decimal format.

/<flag>

Specifies additional flags:

novar

Disables the display of variables. **nov** is the alias for this flag.

nocaller

Disables the display of function callers. **noc** is the alias for this flag. This flag is set by default.

args

Displays arguments. **a** is the alias for this flag. This flag is set by default.

locals

Displays local variables. `l` is the alias for this flag.

caller

Displays function callers. `c` is the alias for this flag.

json

Specifies an output option to display messages in JSON format.

Example: Displaying the stack 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
```

3.2.17 CMM-style commands: var.global

Displays all global variables.

Syntax

```
var.global [%<printing_format>] [/<flag>]
```

Alias:

```
v.g [%<printing_format>] [/<flag>]
```

Parameters**%<printing_format>**

Specifies one of [ascii | binary | decimal | hex] with aliases of [a | bin | d | h]. 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: Displaying global variables

```
var.global          ; Display all global variables
var.global %h       ; Display all global variables in hexadecimal
```

3.2.18 CMM-style commands: var.local

Displays all local variables in a function.

Syntax

```
var.local [%<printing_format>] [/<flag>]
```

Alias:

```
v.l [%<printing_format>] [/<flag>]
```

Parameters

%<printing_format>

Specifies one of [ascii | binary | decimal | hex] with aliases of [a | bin | d | h]. 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: Displaying local variables

```
var.local                ; Display all local variables
var.local %h             ; Display all local variables in hexadecimal
```

3.2.19 CMM-style commands: var.new

Creates a new script variable and zero-initializes it. Script variables are for use at runtime only.

Syntax

```
var.new \<name>
```

Alias:

```
v.n \<name>
```

Parameters

<name>

Specifies the name of a script variable.

Example: Creating a new script variable

```
var.new \myVar ; Create new script variable
```

3.2.20 CMM-style commands: var.print

Concatenates the results of one or more expressions.

Syntax

```
var.print [%<printing_format>] {expression} ... [/<flag>]
```

Alias:

```
v.print [%<printing_format>] {expression} ... [/<flag>]
```

Parameters

%<printing_format>

Specifies one of [ascii | binary | decimal | hex] with aliases of [a | bin | d | h]. 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: Displaying strings and variables

```
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
```

3.2.21 CMM-style commands: var.set

Sets and displays the value of an existing script variable. It can also display the result of an expression.

Syntax

```
var.set [\<name>] [=] [expression]
```

Alias:

```
v.s [\<name>] [=] [expression]
```

Parameters

\<name>

Specifies the name of an existing script variable.



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.

Operation

Script variables are for use at runtime only.

Example: Setting and displaying a script variable

```

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

```

3.2.22 CMM-style commands: wait

Pauses the execution of a script for a specified period of time.

Syntax

```
wait {number}{m | s}
```

Parameters

<number>

Specifies the period of time.

m

Specifies the time in milliseconds.

s

Specifies the time in seconds.

Example: Pausing a running script

```

wait 1s           ; Wait one second
wait 0.5s         ; Wait half a second
wait 1000m        ; Wait one thousand milliseconds

```

4. GNU Free Documentation License Details

Provides details about the GNU Free Documentation License and how to use it.

4.1 GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc. 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. 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. Secondly, 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 "copy left", 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 copy left 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.

1. 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.

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 "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by

reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

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:

- 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.
- 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.
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- 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.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- 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.
- 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.
- 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.
- 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.
- Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate

some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights 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.

8. 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.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt otherwise to copy, modify, sublicense, or distribute the Document 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.

10. 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.

4.2 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
```

```
FrontCover Texts being <list>, and with the Back-Cover Texts being <list>.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

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.

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>.

Copyright © 2009 ARM. 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”.

Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in Arm documents.

Product status

All products and services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

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

Revision history

These sections can help you understand how the document has changed over time.

Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

Document history

Issue	Date	Confidentiality	Change
6.5.0-00	13 March 2025	Non-Confidential	Updated document for Arm Debugger 6.5.0
6.4.0-00	29 November 2024	Non-Confidential	Updated document for Arm Debugger 6.4.0
6.3.0-00	23 October 2024	Non-Confidential	Updated document for Arm Debugger 6.3.0
6.2.0-00	11 June 2024	Non-Confidential	Updated document for Arm Debugger 6.2.0
6.1.2-00	10 April 2024	Non-Confidential	Updated document for Arm Debugger 6.1.2

Issue	Date	Confidentiality	Change
6.1.0-00	29 February 2024	Non-Confidential	Document moved from Arm Development Studio to Arm Debugger documentation set, to create Arm Debugger version 6.1.0 document.
2023.1-00	25 October 2023	Non-Confidential	Updated document for Arm Development Studio 2023.1
2023.0-00	13 April 2023	Non-Confidential	Updated document for Arm Development Studio 2023.0
2022.2-00	17 November 2022	Non-Confidential	Updated document for Arm Development Studio 2022.2
2022.1-00	21 July 2022	Non-Confidential	Updated document for Arm Development Studio 2022.1
2022.0-01	27 April 2022	Non-Confidential	Updated document for Arm Development Studio 2022.0
2022.0-00	29 March 2022	Non-Confidential	Updated document for Arm Development Studio 2022.0 Beta
2021.2-00	10 November 2021	Non-Confidential	Updated document for Arm Development Studio 2021.2
2021.1-01	26 August 2021	Non-Confidential	Documentation update 1 for Arm Development Studio 2021.1
2021.1-00	9 June 2021	Non-Confidential	Updated document for Arm Development Studio 2021.1
2021.0-00	19 March 2021	Non-Confidential	Updated document for Arm Development Studio 2021.0
2010-00	28 October 2020	Non-Confidential	Updated document for Arm Development Studio 2020.1
2000-01	3 July 2020	Non-Confidential	Documentation update 1 for Arm Development Studio 2020.0
2000-00	20 March 2020	Non-Confidential	Updated document for Arm Development Studio 2020.0

Issue	Date	Confidentiality	Change
1910-00	1 November 2019	Non-Confidential	Updated document for Arm Development Studio 2019.1
1901-00	15 July 2019	Non-Confidential	Updated document for Arm Development Studio 2019.0-1
1900-00	11 April 2019	Non-Confidential	Updated document for Arm Development Studio 2019.0
1800-02	31 January 2019	Non-Confidential	Documentation update 2 for Arm Development Studio 2018.0
1800-01	18 December 2018	Non-Confidential	Documentation update 1 for Arm Development Studio 2018.0
1800-00	27 November 2018	Non-Confidential	First release for Arm Development Studio

Change history

For information about the technical changes to the Arm Debugger Command Reference, see the [Arm Debugger Release Notes](#).

Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
<i>italic</i>	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.

Convention	Use
monospace	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
monospace <u>underline</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: <div>MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2></div>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the <i>Arm® Glossary</i> . For example, IMPLEMENTATION DEFINED , IMPLEMENTATION SPECIFIC , UNKNOWN , and UNPREDICTABLE .



We recommend the following. If you do not follow these recommendations your system might not work.



Your system requires the following. If you do not follow these requirements your system will not work.



You are at risk of causing permanent damage to your system or your equipment, or harming yourself.



This information is important and needs your attention.



A useful tip that might make it easier, better or faster to perform a task.



A reminder of something important that relates to the information you are reading.

Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
Arm Compiler for Embedded Arm C and C+ + Libraries and Floating-Point Support User Guide	100073	Non-Confidential
Arm Compiler for Embedded User Guide	100748	Non-Confidential
Arm Development Studio Getting Started Guide	101469	Non-Confidential
Arm Development Studio Heterogeneous system debug with Arm Development Studio	102021	Non-Confidential
Arm Debugger Release Note	109667	Non-Confidential
Arm Development Studio User Guide	101470	Non-Confidential
Arm DSTREAM-HT Getting Started Guide	101760	Non-Confidential
Arm DSTREAM-HT System and Interface Design Reference Guide	101761	Non-Confidential
Arm DSTREAM-PT Getting Started Guide	101713	Non-Confidential
Arm DSTREAM-PT System and Interface Design Reference Guide	101714	Non-Confidential
Arm DSTREAM-ST Getting Started Guide	100892	Non-Confidential
Arm DSTREAM-ST System and Interface Design Reference Guide	100893	Non-Confidential
Arm DSTREAM-XT Getting Started Guide	102443	Non-Confidential
Arm DSTREAM-XT System and Interface Design Reference Guide	102444	Non-Confidential
Component Architecture Debug Interface User Guide	100963	Non-Confidential
CoreSight Trace Memory Controller Technical Reference Manual	DDI0461	Non-Confidential
Fast Models Fixed Virtual Platforms Reference Guide	100966	Non-Confidential
Fast Models Reference Guide	100964	Non-Confidential
Iris User Guide	101196	Non-Confidential

Arm® architecture and specifications	Document ID	Confidentiality
Arm Debug Interface Architecture Specification ADIV5.0 to ADIV5.2	IHI0031	Non-Confidential
ARMv7-M Architecture Reference Manual	DDI0403	Non-Confidential
CoreSight CoreSight Components Technical Reference Manual	DDI0314	Non-Confidential
CoreSight Program Flow Trace Architecture Specification	IHI0035	Non-Confidential
CoreSight System Trace Macrocell Technical Reference Manual	DDI0444	Non-Confidential

Non-Arm resources	Documentation	Document ID
Eclipse Documentation	-	Eclipse Foundation
GNU licensing	-	GNU Operating System
Mastering Regular Expressions	ISBN 0-596-52812-4	Jeffrey E. F.Friedl