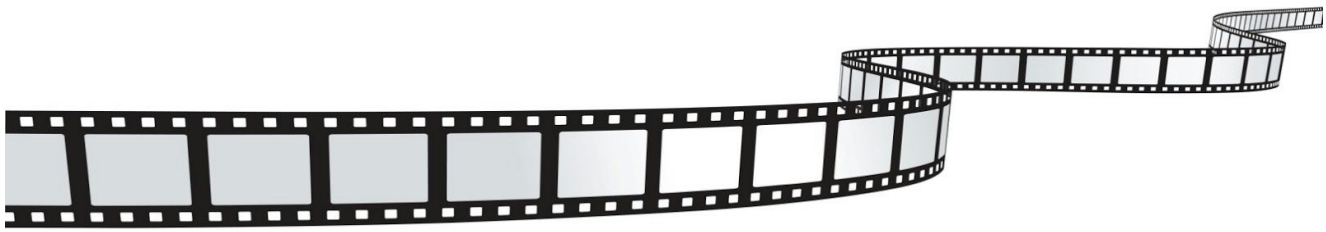
moviDebug Migration Guide

Using the next generation of the Movidius Debugger

00.50.74



Introduction

We presume you are already somewhat familiar with the previous version of the Movidius Debugger (`moviDebug`). This document intends to serve as a guide and reference to map your CLI knowledge and to rewrite your scripts for use with the new generation of the Movidius Debugger (`moviDebug2`).

General considerations

`moviDebug2` has been redesigned from the ground up to provide consistency across different debugging scenarios. These include but are not limited to:

- Interactive Command-line debugging.
- Scripted target operation, e.g debugging, testing, profiling.
- Eclipse-based debugging and software development.

We employed a set of existing technologies which have left their mark on the final product in terms of its interface, and sometimes they fundamentally changed the way debugging is performed.

The Interactive CLI and scripting front-end of the debugger use the industry standard **Tool Command Language** (Tcl) which imposed some syntactic restrictions and changes. The debugging backend uses an extended **Target Communication Framework** Agent prototype, this also had a considerable impact on the structure and operation some of the commands. We strived to remain faithful to the original command set, but in some cases this was not technically feasible.

We also redesigned the command set to group the existing commands around a **limited set of basic commands** the behaviour of which can be altered using **subcommands** and/or **switches**. This provide a better modularity and allows for a hierarchical organisation.

We also removed some obsolete features, or did not implement seldom used features. Most of these can be custom-written using the powerful scripting language available to the end-user.

CLI syntax changes

`moviDebug2` integrates a **complete Tcl/Tk 8.6.4 interpreter environment**. This means that its command language is in fact the Tcl language (https://www.tcl.tk/about/language.html) itself.

This has some important implications:

- Command names are ***case sensitive***. Official `moviDebug2` commands are lowercase, like built-in Tcl commands.
- The only acceptable parameter separator is (white)space (newline excluded). **Comma** (**,**) is a regular character, **needs to be removed** from old scripts.
- Freestanding white space will always separate words (newline separates commands) . Use double quotes **""** to group.
- Curly braces **{}** also avoid various forms of *substitution* in addition to grouping.
- The comment semicolon **;** has a different role -- it separates commands when they are on the same line. To get inline comments after a command, the semicolon must be followed by the Tcl comment marker **#**.
- Square brackets **[]** denote *command substitution* in scripts: result of enclosed script will be substituted as single word.

- Backslashes are escape characters used for *backslash substitution*. Please take care when using Windows paths: either use the forward slash **/** or surround your text with curly braces **{}**. Backslash-escaped space characters will also avoid splitting words, but damage readability.
- The dollar sign **$** is used for *variable substitution*. That is, variable names get replaced by their values. No further word splitting is done if they contain whitespaces. Use the Tcl argument list expansion operator **{*}** if you wish this sort of behaviour.
- Round parentheses **()** are usually plain characters but in conjunction with dollar sign **$** they are part of array variable syntax. Use backslashes or curly braces to disambiguate.

Tcl also has a set of powerful **built-in commands** which has made some of features of the Classic CLI obsolete.

- Support for round parenthesized () **expressions** has been removed, in favour of the Tcl built-in **expr** command plus command substitution. Change **(***expression***)** to **[expr {***expression***}]**
- **Macros have been removed** in favour of the built-in **proc** command (create procedure).

There are also **changes in notation** in the CLI help and command reference:

- Following the Tcl tradition, *optional parameters* are surrounded by question marks **??** instead of square brackets **[]**, to disambiguate from command substitution. We kept some of the square brackets in places where they were more expressive.

Reorganised command set

We chose that instead of the myriad of fixed function commands present in the original debugger, we will present a hierarchical organisation of commands and subcommands. A large part of the command set follows this pattern:

command ?*subcommand*?... ?**-***options* [*param*]?... ?*arguments*?...

Perhaps the most outstanding example of this is the **breakpoint** command which replaces the combinatorial explosion of the various [**d**][**a**][**r**][**w**][**un**]**break**[**8**|**16**|**32**|**64**] command group.

The user can always define a **custom set of aliases** if a shorter syntax is preferable.

New packaging

moviDebug2 uses the following physical layout:

*platformDir***/bin/moviDebug2**[**.exe**]

The **moviDebug2** Shell Executable

- started manually, from MDK or by Eclipse
- contains the CLI Console implementation and the Tcl/Tk interpreter threads

*platformDir***/lib/moviDebug2.**{**so**|**dll**}

The unified **moviDebug2** Shared Library

- loaded automatically by the Shell
- is also a loadable Tcl module
- it also has standalone C API
- contains TCF Agent and Model implementation
- **Static library linking is not supported**

*platformDir***/tcl/***

- The Tcl/Tk shells and libraries

`common/moviDebug2/include/*`

- Includes for the C API of the Library

`common/moviDebug2/tcl/*`

- contains all the Tcl scripts for the Shell and the Library
- most CLI commands are implemented here

`common/moviDebug/ddrinit/*`

- Myriad2 DDR initialisation support (shared with previous version of moviDebug)

New mechanics

Multithreading

`moviDebug2` uses three principal threads of execution.

Console thread

This is the main thread of the application and it usually is waiting for user input the command line editing component. The issued command is sent to the Tcl interpreter, the result is then displayed here.

Tcl/Tk interpreter thread

This is the thread where all Tcl commands and scripts get executed. These are usually sent from the Console thread but can come from other sources as well.

It is also the home of the **Tcl/Tk event loop**, which means that a **long running scripts should also do event processing.** This automatically happens for most of the `moviDebug2` commands, but it does not happen for most built-in Tcl commands.

The **UART polling** and the **Debug Pipes** implementation run entirely in this thread as **periodic events**.

Standard Tcl input/output streams in this interpreter are synchronised with the Console thread.

There is also an additional set of Tcl streams linked to the Console which allow different categories of output messages (they usually appear in different colour).

TCF thread

This is where our Target Communication Framework Agent lives. It is also home for the Myriad platform abstraction Model which talks to the the Debug Server or the Simulator.

The Agent's primary responsibility is to handle TCF protocol commands coming from both Tcl/Tk and Eclipse, usually by executing various debugging tasks. It monitors and manages the Model accordingly.

It is highly asynchronous by design so it runs **its own event processing loop**.

The **TCF protocol messages** related to Agent and Model state changes are **sent back asynchronously** to Eclipse and the Tcl/Tk interpreter thread.

Autocomplete

When the user presses the TAB key during line editing, an autocomplete processing command is also sent to the Tcl interpreter. The result of the command is then used in the completion algorithm.

The fully scripted autocomplete logic is much more powerful than in the previous generation and helps the user discover new commands, options and parameters. It also does completion on Tcl variables, namespaces, TclOO object methods, and, of course, file names.

`MoviDebug2` commands provide their own autocomplete handlers, so that completion can be performed on almost any type of parameter. Aliases, imports and variable substitutions are traversed through during completion to discover the original command/object. **Note:** *this still does not mean that completion suggestions are always valid in all contexts.*

There is a bash-like *reverse-i-search* feature which helps locate previously issued commands.

Control-C

The meaning of the Control-C key combination has changed significantly.

- In line editing mode it will **cancel the editing of the current command**. This is especially useful for *multi-line* incomplete commands, when the user loses track of all the opened braces, brackets and quotes.
- When executing a command, the Control-C combination can be used to **cancel the currently executing script or command**. This will usually *propagate back* to the console as a *Tcl error result*.
- **No changes** will happen in the **target platform** as a result of the Control-C interrupt.
- **Exiting** the debugger using **double Control-C** is **no longer supported**. - - **use Control-D** which is the standard End Of File marker in most of POSIX CLI environments.

(Tcl) Errors

Error handling in `moviDebug2` has also changed dramatically.

- We use the Tcl **error result** mechanism. This can be described like **throwing string exceptions**.
- This means any error result **interrupts the normal flow** of the scripts and need to be caught using the **catch** command (https://www.tcl.tk/man/tcl8.6/TclCmd/catch.htm).
- *Uncaught errors in Tcl initialisation scripts* may cause the debugger to **not enter interactive mode**.

Different command line options

Although **we support the old format** of the most frequently used command line options, we recommend using a more POSIX-conformant format. The mapping from old-style to new style switches are presented in the table below:

moviDebug	moviDebug2
-h[elp]	**--help**
-version	**--version**
-verbose	**--verbose**
~~-b:<fileName>~~	**--tcl-init** <*fileName*> **--tcl-script** <*fileName*>
~~-err[:<fileName>]~~	**--log**[**-file**] <*fileName*>
~~-D:<symName>:[<symValue>]~~	**--**[**tcl-**]**var** variable**=**[*value*] **-D:***symbolName***=**[*value*]
~~-noColors~~	**--no-color** **--stdio**
-serverIP:<address>	**--server**[**-host**] <*host*>
-serverPort:<port>	**--**[**server-**]**port** <*port*>
-cv:<chipVersion>	**--chip-version** <*chipVersion*>
~~-noUnmappedChecks~~	Not supported.
~~-wd:<dirName>~~	**--**[**change-**]**dir**[**ectory**] <*directory*> **-C** <*dirName*>
~~-dlog[:<fileName>]~~	--log-file <fileName>
~~-uart[:<fileName>]~~	Not supported from CLI.
~~-noInit~~	Not supported.
~~-cacheAware~~	Not supported.

Command set migration

Expressions → expr

The classic parenthesized expression syntax has been superseded by the use of the built-in **expr** command and command substitution. Refer to https://www.tcl.tk/man/tcl/TclCmd/expr.htm for exact rules.

Addresses of register names can be retrieved using the `mdbg::getRegAddr` command and substituted using regular command substitution.

The supported register names and constants are taken from the platform-specific **registersMA2*xxx*.h** header files. There are some additional definitions in the **fragrakRegisters.tcl** file.

Macros → proc

Macros were superseded by the more advanced feature of Tcl procedures.

Classic Syntax:

```
macro <macroName>[ , <arguments>]
            …      ;macro body
endm
```

New Syntax:

```
proc name { [<arguments>] } {
      # body of procedure
      …
}
```

Details:

Procedure names are case-sensitive. The arguments of the macros are local variables on the procedures's stack, and can be referred to using variable substitution. As per Tcl rules, global variables are not automatically visible. See https://www.tcl.tk/man/tcl8.6/TclCmd/proc.htm

An example is given below:

Before:

```
macro SetDefaultShaveWindows svuNumber
   set (0x80140010 + 0x10000 * \svuNumber), (0x10008000 + 0x20000 * \svuNumber)
   set (0x80140014 + 0x10000 * \svuNumber), (0x10000000 + 0x20000 * \svuNumber)
   set (0x80140018 + 0x10000 * \svuNumber), (0x10010000 + 0x20000 * \svuNumber)
   set (0x8014001C + 0x10000 * \svuNumber), (0x10018000 + 0x20000 * \svuNumber)
endm
```

After:

```
proc SetDefaultShaveWindows {svuNumber} {
      foreach idx {0 1 2 3} {
            mset [expr {0x80140010 + 4 * $idx + 0x10000 * $svuNumber}] \
                  [expr {0x10000000 + (0x8000 * $idx) + 0x20000 * $svuNumber]
      }
}
```

Undef

Classic Syntax:

```
undef <macroName> [, <arguments>]
ud <macroName> [, <arguments>]
```

New Syntax:

```
rename <procedureName> {}
unset [-nocomplain] {variableName}
```

Details

Tcl procedures can be `renamed`. Renaming to empty string results in deletion of that procedure.

See https://www.tcl.tk/man/tcl8.6/TclCmd/rename.htm

Tcl variables can be `unset`. This means referring to them afterwards will cause an error.

See https://www.tcl.tk/man/tcl8.6/TclCmd/unset.htm

Comments

`moviDebug2` uses the Tcl comment syntax. See http://www.tcl.tk/man/tcl/TclCmd/Tcl.htm#M30

Batch files → Tcl scripts

`moviDebug2` is mostly a regular Tcl/Tk interpreter shell. It accepts script file names in the startup command line arguments list. These script files are mostly regular Tcl/Tk scripts.

The debugger has two modes of operation: scripted mode and interactive mode.

When running scripts specified from command line, the debugger runs in scripted mode.

After running the command line scripts, it enters interactive mode (depends on switches, see command line reference).

There are two major differences between scripted mode and interactive mode.

1. **Command name resolution**

- All the officially supported `moviDebug2` commands are defined inside the **::mdbg** namespace.
- This namespace is automatically added to the Tcl **namespace path** in interactive shell mode so the user can use the debugger commands directly.
- When executing Tcl (startup) scripts specified at command line, this namespace is **not automatically available**. This is to keep consistent with the case when **tclsh** or **wish** loads the script, and to be able to use regular Tcl/Tk scripts unmodified. It is also the way `moviDebugTcl.so` worked.
- This means that in Tcl scripts the `moviDebug2` commands they either should be *explicitly qualified*, *imported* using **namespace import** or the *namespace path* could be altered with the **namespace path** Tcl command.
- Our recommendation *for simple scripts* is to add the following line to the top of the file:

```
namespace path ::mdbg
```

2. **Echoing the result**

- The `moviDebug2` interactive Shell is essentially a Read-Eval-Print Loop (REPL). This means that the command is first entered, sent for evaluation and the result left in the interpreter is printed to the standard output. moviDebug2 will highlight the result of the command to easily differentiate if from regular standard output.

- In scripted mode there is no REPL. The commands are executed and the result of the commands is silently set into the interpreter. Invoking new commands will overwrite that result.
- To work around this the user needs to manually print whatever the output of the script needs to be, usually using **puts** [*command ?args?*]
- The command **mdbg::eval%** was specifically created for the scenario when user input needs to be emulated. This command will display both the command in question and its result, highlighted in separate colours. Commands from the **::mdbg namespace** are **automatically available** *during* evaluation. There is also some extra Tcl/Tk event processing taking place. These make the scripts behave almost exactly as if issued interactively. Consult the command reference for more details. (e.g. type **help eval%** in interactive mode.)

3. The Tcl/Tk event loop

- Scripts run in the Tcl/Tk interpreter thread. In interactive mode this thread is usually processing all the Tcl/Tk messages, but in scripted mode it's mostly busy executing the scripts. The user needs to make sure that Tcl event processing is not suppressed during the execution of Tcl/Tk scripts.
- This usually is not a problem, because a large portion of the debugger command set does Tcl event processing while waiting.
- *Avoid* using the Tcl **after** command *for unconditional delay* in scripts. Use the **wait -ms** command of the debugger instead.

Core/Target → target, startupcore

Classic Syntax:

```
core <coreName>
target <targetName>
t <targetName>
```

New Syntax:

```
target <targetName>
startupcore <targetName>
```

Details:

Target names remain unchanged. There are also additional targets.

Semantic change:

The function of the old **target** command has been split into two distinct commands.

Before:

```
core s0            ;select SHAVE 0 as current core
```

After:

```
target s0          ; # select SHAVE 0 as current core
startupcore LRT    ; # select LRT as entry point for application
```

SetDebugGroup
GetDebugGroup
UnsetDebugGroup
SetDebugMaster
UnsetDebugMaster

These commands were not properly supported for MA2100 even in moviDebug Classic.

There is no easy alternative to them, the user can employ a Tcl list variable and **foreach** to iterate over each core.

Example:

```
set group {S0 S1 S2}
foreach core $group {
    step over -asm -target $core -async
}
foreach core $group {
    wait -target $core -suspended
}
```

Start

Classic Syntax:

```
start <targetName>
```

New Syntax:

- Not available.

Alternative:

```
cpr enable <arguments...>
```

Details:

This command has been removed because its functionality has been integrated into the TCF Myriad Platform Model's ELF file loader. The startup core and the Shave cores having window registers defined in the ELF are automatically started up by the Model.

The Clock/Power/Reset bits are individually controllable by the **cpr** command.

Stop

Classic Syntax:

```
stop <targetName>
```

New Syntax:

- Not available.

Alternative:

```
cpr disable <arguments….>
```

Details:

This command has seldom been used and was removed.

Reset

Classic Syntax:

```
reset [<targetName>]
res [<targetName>]
```

New Syntax:

- Not available.

Alternative:

```
breset
```

Details:

This command has seldom been used and was removed.

Breset

Classic Syntax:

```
boardReset
breset
```

New Syntax:

```
breset
```

Semantic change:

UART is reinitialised only if enabled.

Step

Classic Syntax:

```
step [<n>]
st[<n>]

step [<coreName>]
st [<coreName>]
```

New Syntax:

```
step [into] [-target <coreName>] [-count <n>] [-asm]
```

Semantic change:

Stepping is implemented using the TCF Agent. Multiple steps are always executed as a sequence of single-step operations, regardless of the target.

Stepo

Classic Syntax:

```
stepo
```

New Syntax:

```
step over -asm
```

Semantic changes:

Stepping is implemented using the TCF Agent. Not specifying the -asm switch will make the operation source-line based. Shave core supported. Might employ hardware breakpoint to correctly return from callee.

Run

Classic Syntax:

```
run
run <entryPoint>
r <entryPoint>
```

New Syntax:

```
run
```

Semantic change:

`run` always just **`cont`**inues execution of the main core specified by the **`startupcore`** command. (LOS is default) **Issuing `run` the second time after the application terminates will not relaunch it.**

Entry points are set during initial load.

Setting custom entry points is not supported directly. The values for the registers need to be manually set.

Before:

```
run
run main
```

After:

```
run

# run main
set addr [sym addr main]
mset PC $addr
mset NPC [expr {$addr + 4}]
cont
```

RunW

Classic Syntax:

```
RunAndWait [<entryPoint>] [<maxWaitTime>]
runw [<entryPoint>] [<maxWaitTime>]
rw [<entryPoint>] [<maxWaitTime>]
```

New Syntax:

```
run -wait
runw
runandwait
```

Semantic change:

This is essentially a variant of run. In addition to the normal retrurn condition, any breakpoint on any core will cause the command to return.

Timing out is considered a failure by the command and will raise an error.

Before:

```
runw
runw main
```

After:

```
runw
-
```

Continue

Classic Syntax:

```
continue [<targetName>]
cont [<targetName>]
c [<targetName>]
```

New Syntax:

```
cont [<targetName>]
```

ContinueAndWait

Classic Syntax:

```
ContinueAndWait [<targetName>] [<maxWaitTime>]
contw [<targetName>] [<maxWaitTime>]
cw [<targetName>] [<maxWaitTime>]
```

New Syntax:

```
cont -wait [-timeout <timeout>] [<target>]
contw [-timeout <timeout>] [<target>]
```

Semantic change:

This is essentially a variant of **cont**. In addition to the normal return condition, hitting any breakpoint on any other core will cause the command to return.

Timing out is considered a failure by the command and will raise an error.

Get → mget

Classic Syntax:

For registers:

get	<registerName>[, <count>]	g	<registerName>[,<count>]
getword	<registerName>[, <count>]	gw	<registerName>[,<count>]
get32	<registerName>[, <count>]	g32	<registerName>[,<count>]
x	<registerName>[, <count>]		
geth32	<registerName>[, <count>]	gh32	<registerName>[,<count>]
geti	<registerName>[, <count>]	gi	<registerName>[,<count>]
geti32	<registerName>[, <count>]	gi32	<registerName>[,<count>]
getu	<registerName>[, <count>]	gu	<registerName>[,<count>]
getu32	<registerName>[, <count>]	gu32	<registerName>[,<count>]
getfloat	<registerName>[, <count>]	gf	<registerName>[,<count>]
getfloat32	<registerName>[, <count>]	gf32	<registerName>[,<count>]
getshort	<registerName>[, <count>]	gs	<registerName>[,<count>]
geth16	<registerName>[, <count>]	gh16	<registerName>[,<count>]
gethalf	<registerName>[, <count>]	gh	<registerName>[,<count>]
geti16	<registerName>[, <count>]	gi16	<registerName>[,<count>]
getu16	<registerName>[, <count>]	gu16	<registerName>[,<count>]
getfloat16	<registerName>[, <count>]	gf16	<registerName>[,<count>]
getbyte	<registerName>[, <count>]	gb	<registerName>[,<count>]
geth8	<registerName>[, <count>]	gh8	<registerName>[,<count>]
geti8	<registerName>[, <count>]	gi8	<registerName>[,<count>]
getu8	<registerName>[, <count>]	gu8	<registerName>[,<count>]

New Syntax:

```
mget [-reg] <registerName> [-type int|unsigned|float] [-size 1|2|4|8] [<count>]
state -pc
```

For memory content:

get	<address>[, <count>]	g	<address>[, <count>]
getword	<address>[, <count>]	gw	<address>[, <count>]
get32	<address>[, <count>]	g32	<address>[, <count>]
x	<address>[, <count>]		
geth32	<address>[, <count>]	gh32	<address>[, <count>]
geti	<address>[, <count>]	gi	<address>[, <count>]
geti32	<address>[, <count>]	gi32	<address>[, <count>]
getu	<address>[, <count>]	gu	<address>[, <count>]
getu32	<address>[, <count>]	gu32	<address>[, <count>]
getfloat	<address>[, <count>]	gf	<address>[, <count>]
getfloat32	<address>[, <count>]	gf32	<address>[, <count>]
getshort	<address>[, <count>]	gs	<address>[, <count>]
geth16	<address>[, <count>]	gh16	<address>[, <count>]

```
gethalf         <address>[, <count>]          gh     <address>[, <count>]
geti16          <address>[, <count>]          gi16   <address>[, <count>]
getu16          <address>[, <count>]          gu16   <address>[, <count>]
getfloat16      <address>[, <count>]          gf16   <address>[, <count>]
getbyte         <address>[, <count>]          gb     <address>[, <count>]
geth8           <address>[, <count>]          gh8    <address>[, <count>]
geti8           <address>[, <count>]          gi8    <address>[, <count>]
getu8           <address>[, <count>]          gu8    <address>[, <count>]
```

New Syntax:

```
mget ?-addr? address ?-type int|unsigned|float? ?-size 1|2|4|8? ?-count count?
jtag get64 address   ; # for aligned 64-bit transaction
mget -size 8         ; # for 64-bit data size
mget -type float -size 8 ; # show as double-precision IEEE float
```

Alternative:

```
mdump <location> <count>
```

Notes:

- Ranged, stepped get is not available, needs to be user-coded.
- Full C/C++ expression evaluator is available with DWARF support.
- `mget` supports data structures and arrays if `-depth` is greater than zero.
- For more details please consult the `mget` command reference.

Set → mset

Classic Syntax:

```
set <registerName> <value> [<count>]
s <registerName> <value> [<count>]

set <address> <value> [<count>]
s <address> <value> [<count>]

wm <address> <value> [<count>]
```

New Syntax:

```
mset -type {int|unsigned|float} -size {1|2|4|8} <location> <value>
jtag set64 <address> <value>
```

Fill

Classic Syntax:

```
fill   <startAddr>, <numberOfElements>, <stride>, <listOfValues>
fill32 <startAddr>, <numberOfElements>, <stride>, <listOfValues>
fillw  <startAddr>, <numberOfElements>, <stride>, <listOfValues>
fill16 <startAddr>, <numberOfElements>, <stride>, <listOfValues>
fillh  <startAddr>, <numberOfElements>, <stride>, <listOfValues>
fill8  <startAddr>, <numberOfElements>, <stride>, <listOfValues>
fillb  <startAddr>, <numberOfElements>, <stride>, <listOfValues>
```

New Syntax:

- Not implemented.

Details:

The **`mfill`** command was left out of the current implementation.

SetRegisterFields

Classic Syntax:

```
SetRegisterFields <registerAddress>, <description>, <fields>
```

New Syntax:

- Not available.

SetRegisterFormat

Classic Syntax:

```
SetRegisterFormat <formatSpecifier>
srf <formatSpecifier>
```

New Syntax:

- Not available.

Break
Hbreak

Classic Syntax:

```
break [<address>]
b [<address>]
hbreak [<address>]
hb [<address>]
```

New Syntax:

```
breakpoint add -type {software|hardware} [-location] <address>
bp add -type hardware [-read] [-write] -location <address|variable>
```

Semantic change:

The breakpoint insertion and removal logic has fundamentally changed because of the way the Target Communication Framework (Agent) handles breakpoints.

- See top-level documentation for the **breakpoint** command (type **help breakpoint** in the CLI).
- A breakpoint may not be physically planted right away if its address cannot be resolved.
- The framework re-evaluate breakpoints when the executable's memory layout changes (e.g. load).

Unbreak,
Hunbreak,
Dunbreak ⟶ breakpoint remove

Classic Syntax:

```
hunbreak <address>
hub <address>
```

```
unbreak <address>
ub <address>
dunbreak [<breakpointNumber>|<address>]
```

New Syntax:

```
breakpoint {remove|rm|delete} {<ids>|-all}
```

Before:

```
unbreak 0x1D0000FF      ;remove the SIB present at address 0x1D0000FF
unbreak all                    ;remove all SIB present of current target
```

After:

```
breakpoint remove #1            ; # removes the breakpoint inserted first
breakpoint remove 0x1D0000FF    ; # only works if set by numeric address
breakpoint remove -all          ; # or "bp rm -a" if you like
```

Breakstart, Breakend

Classic Syntax:

```
breakstart <address>
breakend
```

New Syntax:

- Not available.

We do not currently support executing Tcl scripts at breakpoint hit. This might change in the future.

Breaknow → halt

Classic Syntax:

```
breaknow
bn
```

New Syntax:

```
halt
bn
```

Breakall → halt -all

Classic Syntax:

```
breakall
```

New Syntax:

```
halt -a[ll]
bn -a
```

Darbreak, Dawbreak, Daabreak, Drarbreak, Drawbreak

Classic Syntax:

```
darbreak[{8|16|32|64}] [!][<address>][, <length>][, LEW][, <prefetch>]
darbreak <address>
dawbreak[{8|16|32|64}] [!][<address>][, <length>][, LEW][, <prefetch>]
dawbreak <address>
daabreak[{8|16|32|64}] [!][<address>][, <length>][, LEW][, <prefetch>]
daabreak <address>
drarbreak[{8|16|32|64}] [!][<startAddress>:<endAddress>][, LEW][,<prefetch>]
drawbreak[{8|16|32|64}] [!][<startAddress>:<endAddress>][, LEW][,<prefetch>]
draabreak[{8|16|32|64}] [!][<startAddress>:<endAddress>][, LEW][,<prefetch>]
```

New Syntax:

```
breakpoint add -type hardware [-read] [-write] -location <...> -size <...>
```

Changes:

Currently only the Leon+Shave common subset of the classic data breakpoint command set features is supported. Please consult the **breakpoint add** command reference.

Batch, Qbatch → source

Classic Syntax:

```
batch <fileName>
ba <fileName>
qbatch <fileName>
qba <fileName>
```

New Syntax:

```
source <fileName>
```

Semantic change:

- batch files have been replaced by Tcl scripts
- commands are not echoed back, this needs to be manually done if necessary
- results are not printed, needs to be done manually
- use the `mdbg::`command **`eval%`** if verbosity is important.

Before:

```
batch normalTest.bat     ;execute commands from batch file and display
                         ;them to stdout
qbatch normalTest.bat          ;execute commands from batch file
```

After:

```
source normalTest.tcl
```

See https://www.tcl.tk/man/tcl8.6/TclCmd/source.htm

Repeat

Classic Syntax:

```
repeat <numberOfRepetitions>, <command>
```

New Syntax:

```
repeat numberOfRepetitions command/script ?args…?
```

Load

Classic Syntax:

```
load [<address>] [<format>] <fileName>
l [<address>] [<format>] <fileName>

load <rf> [<format>] <fileName>
l <rf> [<format>] <fileName>
```

New Syntax:

```
loadfile [-address <baseAddress>] <fileName>
```

Semantic change:

only little-endian binary files and ELF files are supported

Before:

```
loadfile 0x90100000 test.bin     ;load the program from file test.bin
target s0
loadfile -addr i0 irfTest.bin     ;load the irf registers with the content of
file
```

After:

```
loadfile -bin -address 0x90100000 test.bin     ;#load the program from file
test.bin
target s0
loadfile -addr i0 irfTest.bin     ;#load the irf registers with the content of
file
```

Loadhex

Classic Syntax:

```
loadhex [<address>] <fileName>
lh [<address>] <fileName>
```

New Syntax:

```
Not supported currently.
```

Verify

Classic Syntax:

```
verify <address> <fileName>
v <address> <fileName>
verify <elfFileName>
v <elfFileName>
```

New Syntax:

```
loadfile -noload -verify <fileName>
verify <fileName>
```

Semantic change:

- Mismatch is considered failure and will raise Tcl error result.

Before:

```
load test.elf
verify test.elf
```

After:

```
loadfile test.elf
verify test.elf
```

LoadAndVerify

Classic Syntax:

```
LoadAndVerify <address> <fileName>
lv <address> <fileName>

LoadAndVerify <elfFileName>
lv <elfFileName>
```

New Syntax:

```
loadfile -verify <fileName>
loadandverify <fileName>
```

Semantic change:

- mismatch is considered failure and will raise Tcl error result

Before:

```
LoadAndVerify test.elf
LoadAndVerify 0x40000000, testFrame.bin
```

After:

```
loadfile -verify test.elf
loadandverify -addr 0x40000000 testFrame.bin
```

Save

Classic Syntax:

```
save <startAddress> <lengthInBytes> [<format>] <fileName>
sv <startAddress> <lengthInBytes> [<format>] <fileName>

save <startRegister> <numberOfBytes> [<format>] <fileName>
sv <startRegister> <numberOfBytes> [<format>] <fileName>

save <startAddress> <lengthAddress> [<format>] <fileName>
sv <startAddress> <lengthAddress> [<format>] <fileName>
```

New Syntax:

```
savefile <fileName> [<startAddress>] [<lengthInBytes>]
```

Details:

Only raw little-endian binary format is supported.

Ssave

Classic Syntax:

```
ssave <mofFile> [<filePrefix>]
```

New Syntax:

- Not supported

Append

Classic Syntax:

```
append <startAddress> <lengthInBytes> [<format>] <fileName>
a <startAddress> <lengthInBytes> [<format>] <fileName>

append <startRegister> <numberOfBytes> [<format>] <fileName>
a <startRegister> <numberOfBytes> [<format>] <fileName>

append <startAddress> [<lengthAddress>] [<format>] <fileName>
a <startAddress> [<lengthAddress>] [<format>] <fileName>
```

New Syntax:

```
savefile -append <fileName> [<startAddress>] [<lengthInBytes>]
```

Before:

```
append 0, 100, prgMem.bin      ; append 100 bytes from address 0x00 from data
                               ; memory to the file prgMem.bin
```

```
Append i4, 8, i2i3.bin          ; append i4 and i5 to the file 'i2i3.bin'
```

After:

Help

Classic Syntax:

```
help [<commandName>]
h [<commandName>]
```

New Syntax:

```
help [<commandName>] [*|<subcommandName>]...
```

Before:

```
help
help get
help stepo
```

After:

```
help
help mget
help step over
```

Echo

Classic Syntax:

```
echo [<text>] [<address>]
e [<text>] [<address>]
```

New Syntax:

```
puts [<channelId>] <string>
```

Before:

```
echo Second testpoint
echo DCR = [0x80141800]
```

After:

```
puts "Second testpoint"
puts "DCR = \[0x80141800\]"
```

See https://www.tcl.tk/man/tcl8.6/TclCmd/puts.htm

History, Sethistorydepth

```
sethistorydepth <nr>
shd <nr>
history[<coreName>]
hist [<coreName>]
```

New Syntax:

```
hist [-count <nr>] [[-target] <coreName>]
```

Before:

```
sdg s1, s2          ;create the debug core group
hist                ;display the history for SHAVE 1 and SHAVE 2
```

After:

```
hist s1; hist s2
```

Dasm

Classic Syntax:

```
dasm [<startAddress> | <functionName>] [<length>]
dis [<startAddress> | <functionName>] [<length>]
```

New Syntax:

```
dasm [-count <count>] [-address <address>]
```

Setbits

Classic Syntax:

```
setbits <address>, <bitmask>
```

New Syntax:

- Not available.

Before:

```
setbits 0x80141800, 0x00000001
```

After:

```
set addr 0x80141800
set bits [mget -addr $addr -value]
set bits [expr {$bits | 0x00000001}]
mset  -addr $addr $bits
```

Clearbits

Classic Syntax:

```
clearbits <address>, <bitmask>
```

New Syntax:

- Not available.

Before:

```
setbits 0x80141800, 0x00000001
```

After:

```
set addr 0x80141800
set bits [mget -addr $addr -value]
set bits [expr {$bits &~ 0x00000001}]
mset  -addr $addr $bits
```

Setall

Classic Syntax:

```
setall <address>
```

New Syntax:

- Not available.

Before:

```
setall 0x80141800
```

After:

```
mset 0x80141800 -type int -1
```

Clearall

Classic Syntax:

```
clearall <address>
```

New Syntax:

- Not available.

Before:

```
clearall 0x80141800
```

After:

```
mset 0x80141800 0
```

Test

Classic Syntax:

```
test <address>, <value>
```

New Syntax:

- Not available.

Before:

```
test [0x8014105C] = 0x1d000040
```

After:

```
if {[mdbg::mget -value -addr 0x8014105C] != 0x1d000040]} {
```

```
    puts stderr "Test FAIL"
}
```

Assert

Classic Syntax:

```
assert <address>, <value>
```

New Syntax:

- Not available.

Before:

```
assert [0x8014105C] = 0x1d000040
```

After:

```
if {[mdbg::mget -value -addr 0x8014105C] != 0x1d000040]} {
    error "Test FAIL"
}
```

PC0, PC1

- Not available.

Version

Classic Syntax:

```
version
ver
```

New Syntax:

```
help -version
dll::version
::mvproto::versionString
```

Semantic change:

The help -version will display the version string for the Shared Library and copyright notice.

The mdbg::dll::version function only returns the Shared Library version string,

mvproto::versionString returns the version of the debug server/simulator the debugger is attached to.

Before:

```
ver
version
```

After:

```
help -version
puts [mdbg::dll::version]
```

Clearscreen

Classic Syntax:

```
clearscreen
clear
cls
```

New Syntax:

- Not available.

Pwd

Classic Syntax:

```
pwd
dir
```

New Syntax:

```
pwd
```

Symbols

Classic Syntax:

```
symbols
sym
```

New Syntax:

```
symbol list
sym
```

Copy

Classic Syntax:

```
copy <source> <destination> [<lengthInBytes>]
cp <source> <destination> [<lengthInBytes>]
```

New Syntax:

jtag writeBlock *destination* **[jtag readBlock** *source size***]**

Exit

Classic Syntax:

```
exit
q
```

New Syntax:

```
exit
```

Semantic change:

Safe shutdown performed.

Wait

Classic Syntax:

```
wait <numberOfMilliseconds>
```

New Syntax:

```
wait [-ms <numberOfMilliseconds>]
wait [-suspended [-anybp]] [-timeout <numberOfMilliseconds>]
```

Semantic change:

The **-ms** switch will make the command wait unconditionally.

the **-timeout** switch will make timing out an error result.

Before/After:

```
wait 500                ;wait 500 ms
```

After:

```
wait -ms 500
wait -suspended -timeout 500
```

Asr

Classic Syntax:

```
asr
```

New Syntax:

```
state -reg -asr
```

Semantic change:

The previous command displayed `LEON ASR[16:31]` registers.

The new one only displays `LEON` ASR17 and `ASR[24:31]` registers.

Float

Classic Syntax:

```
float
f
```

New Syntax:

```
state -reg -F
```

Reg

Classic Syntax:

```
reg
```

New Syntax:

```
state -reg

registers
```

Icache, Ricache, Wicache, Rictag, Wictag, Dcache, Rdcache, Wdcache, Rdctag, Wdctag, L1cache

These commands are no longer supported. Cache coherency is managed by the TCF Myriad Platform Model. Doing **mget** and **mset** from a target's perspective will correctly show and update the contents of the Level 1/2 caches.

Ps

Classic Syntax:

```
ps [processor]
```

New Syntax:

```
state [[-target] <core>]

state -all

ps -a
```

Semantic change:

History is not printed for cores.

Before:

`ps` – displays the last 16 instructions and the current state of all processors

`ps s0` – the current state of processor `s0`

After:

```
set cores {los lrt s0 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11}
foreach core $cores {
      state -target $core
      hist $core
}
state s0; hist s0;
```

Debug

Classic Syntax:

```
debug <ON|OFF|SILENT>
```

New Syntax:

```
displaystate <on|off|verbose>
```

Semantic change:

Core state monitoring is done by the TCF Agent so it **cannot be stopped**.

Before:

```
debug on
```

After:

```
displaystate on
```

Chdir

Classic Syntax:

```
chdir <path>
cd    <path>
```

New Syntax:

```
cd    <path>
```

Before:

```
cd C:\
```

After:

cd c:/

New Syntax:

- Not avaliable

Alternative:

```
hist -count <N>
```

Loadsym

Classic Syntax:

```
loadsym <fileName>
```

New Syntax:

```
loadfile -elf -symbols-only <fileName>
loadsym <fileName>
```

Dir

Classic Syntax:

```
dir
```

ls

New Syntax:

```
dir
ls
```

LoadL1, SaveL1, Lic, Sic, Ldc, Sdc,

- **No longer avaliable**

Ahbtrace

Classic Syntax:

```
ahbtrace [<nr>]
ahb [<nr>]
```

New Syntax:

- Not implemented.

Gpio

Classic Syntax:

```
gpio st[atus] [<all> | <pin> | <pin> <number>]
gpio {si|sig|signal}[<all> | <pin> | <name>]
gpio {cfg|config[ure]} <pin> <options>
```

New Syntax:

- for MA2100 target the moviDebug2 command supports an almost identical syntax
- for MA2x5x the command has a completely different syntax, please consult the documentation.

Ddrinit

Classic Syntax:

```
ddrinit
```

New Syntax:

```
ddrinit
```

Semantic change:

The MoviDebug2 command will not check if the DDR is already initialized.

Scp

Classic Syntax:

```
scp indexProcessor
```

New Syntax:

- Currently not supported

Gnp

Classic Syntax:

```
gnp
```

New Syntax:

```
mvproto::getNumberOfChips
```

Asiread, Asiwrite

Classic Syntax:

```
asiread <asiReg> offset [length]

asiwrite <asiReg> offset value [count]
```

New Syntax:

- Not implemented.

Callstack

Classic Syntax:

```
Callstack
```

New Syntax:

```
callstack

cs

state -stack
```

Semantic changes:

- moviDebug2 supports call stack for SHAVE when the compiler emits Call Frame Information
- also works for LEON RTEMS thread contexts

JTAG

Classic Syntax:

```
jtag read <address> [[,] length] [address [[,] length]]...

jtag write <address>, <value> [[,] value]...

jtag ir <address>, <ircode> [[,] ircode]...
jtag pins <pins> <state> ...
```

New Syntax:

```
jtag get32 <address>

jtag set32 <address> <value32>

jtag get64 <address>

jtag set64 <address> <value64>

jtag getBurst32 <address> <numWords32>

jtag setBurst32 <address> <listOfWords>

jtag ir number

jtag pins <pins> <state> ...
```

Rtemsthreads

Classic Syntax:

```
rtemsthreads
```

New Syntax:

```
state -children [target]:RTEMS -stack
```

Rtemsregisters

Classic Syntax:

```
rtemsregisters
```

New Syntax:

```
registers -target [target]:T<threadID>
```

Summary of command mappings

moviDebug	**moviDebug2 (mdbg:: implicit in shell)**
start	Not intended to be implemented
stop	Not intended to be implemented
run	mdbg::run
r	
rw	mdbg::run -wait / mdbg::runw
runandwait	
prfrun	Not intended to be implemented
profilerun	
step	mdbg::step ?into? ?-asm? /
st	mdbg::stepasm / mdbg::stepinstr
	mdbg::stepline
continue	mdbg::cont
cont	
c	
continuew	mdbg::cont -wait /
contw	mdbg::contw
cw	
continueandwait	
break	mdbg::breakpoint
b	mdbg::unbreak
sbreak	
sb	
hbreak	
hb	
unbreak	
ub	
hunbreak	
hub	
get64	mdbg::jtag get64 <address>;
g64	mdbg::mget -size 8
gfloat64	mdbg::mget -type float -size 8

getf64	
gf64	
get	
g	
getword	
gw	
get32	
g32	
x	
geth32	
geti	
gi	
geti32	
gi32	
getu	
gu	
getu32	
gu32	
getfloat	mdbg::mget;
gf	mdbg::mget -type X -size Y
getfloat32	mdbg::state -pc
gf32	
getshort	
gs	
geth16	
gh16	
gethalf	
gh	
geti16	
gi16	
getu16	
gu16	
getfloat16	
gf16	
getbyte	
gb	

geth8	
gh8	
geti8	
gi8	
getu8	
gu8	
set	
s	
set64	mdbg::jtag set64 mdbg::mset -size 8
s64	
set32	mset -size 4 -type unsigned
s32	
seth32	
sh32	
wm	
set16	mdbg::mset -size { 1 | 2 } -type { int | float | double }
s16	
set8	
seth8	
sh8	
setfloat16	
sf16	
fill	mdbg::mfill NOT IMPLEMENTED
fill32	
fillw	
fill8	
fillb	
fill16	
fillh	
getnumberofprocessors	Not supported yet
gnp	
exit	exit
quit	mdbg::quit
q	
load	mdbg::loadfile / mdbg::loadelf
l	

loadhex	Not intended to be implemented
lh	
qload	Not intended to be implemented
ql	
verify	mdbg::loadfile -noload -verify / mdbg::verify
v	
loadandverify	mdbg::loadfile -verify / mdbg::loadandverify
lv	
save	mdbg::savefile
sv	
saveoutput	not supported
so	
append	mdbg::savefile -append
a	
info	~ mdbg::cpr
i	
batch	source
ba	
qbatch	
qba	
help	mdbg::help
h	
reset	--> mdbg::breset
res	
echo	puts
e	
wait	mdbg::wait
pwd	pwd
dir	dir
ls	ls
reg	state -reg
float	state -reg -F
f	
symbols	
sym	sym

copy	Not intended to be implemented
cp	
l1cache	Not intended to be implemented
l1	
rl1	
wl1	
rl1t	
rl1tag	
wl1t	
wl1tag	
sl1	
savel1	
ll1	
loadl1	
target	mdbg::target
t	
core	
scp	Not supported
setcurrentprocessor	
darbreak	mdbg::breakpoint
darbreak8	
darbreak16	
darbreak32	
darbreak64	
dawbreak	
dawbreak8	
dawbreak16	
dawbreak32	
dawbreak64	
daabreak	
daabreak8	
daabreak16	
daabreak32	
daabreak64	
drarbreak	
drarbreak8	

drarbreak16	
drarbreak32	
drarbreak64	
drawbreak	
drawbreak8	
drawbreak16	
drawbreak32	
drawbreak64	
draabreak	
draabreak8	
draabreak16	
draabreak32	
draabreak64	
dunbreak	
asr	mdbg::state -reg -asr
movicompile	Not intended to be implemented
movilink	
moviasm	
asm	
log	Not intended to be implemented
portcheck	
dasm	mdbg::dasm
dis	
breakstart	Not intended to be implemented
breakend	Not intended to be implemented
profile	Not intended to be implemented
prof	Not intended to be implemented
pc0	Not intended to be implemented
pc1	Not intended to be implemented
enable	Not intended to be implemented
disable	Not intended to be implemented
setbits	Not intended to be implemented
setall	
clearbits	

clearall	
assert	Not intended to be implemented
test	
version	mdbg::help -version;
ver	mdbg::dll::version
clear	Not intended to be implemented
cls	
clearscreen	
icache	Not intended to be implemented
ic	
dcache	
dc	
ricache	
ric	
wicache	
wic	
rdcache	
rdc	
wdcache	
wdc	
rdct	
rdctag	
wdct	
wdctag	
rict	
rictag	
wict	
wictag	
sic	
lic	
sdc	
ldc	
store	Not intended to be implemented
restore	
flush	
macro	proc

endm	
regress	Not intended to be implemented
connect	Not intended to be implemented
reconnect	
disconnect	
breset	mdbg::breset
boardreset	
loadsym	mdbg::load --symbols-only / mdbg::loadsym
breaknow	mdbg::breaknow
bn	
breakall	mdbg::breakall
bl	
gms	Not implemented yet
lm	Not implemented yet
um	Not implemented yet
ps	state -all
setcurrentboard	Not intended to be implemented
setboard	
scb	
setdebugmaster	Not intended to be implemented
sdm	
unsetdebugmaster	
udm	
setdebuggroup	
sdg	
getdebuggroup	
gdg	
unsetdebuggroup	
udg	
hist	mdbg::hist
history	
setregisterfields	Not intended to be implemented
setregisterformat	Not intended to be implemented
srf	
debugstart	Not intended to be implemented
ds	

sethistorydepth shd	--> mdbg::hist -count N
cd chdir	mdbg::cd
debug	~ mdbg::displaystate
checkbus	Not intended to be implemented
uartsend us	Not intended to be implemented
ahbtrace ahb	Not implemented yet
testall	Not intended to be implemented
ssave	Not intended to be implemented
gpio	mdbg::gpio
repeat	for while mdbg::repeat
undef ud	unset variable rename procedure {}
callstack	mdbg::callstack; state -callstack
writeflash	Not intended to be implemented
wf	Not intended to be implemented
ddrinit	mdbg::ddrinit
asiwrite	Not implemented yet
asiread	Not implemented yet
stepo	mdbg::step over -asm
jtag	mdbg::jtag
uart	mdbg::uart
pipe	mdbg::pipe
rtemsthreads	state -children [target]:RTEMS -stack
rtemsregisters	registers -target [Target]:T<threadID>
shell	mdbg::shell::exec
lasted	mdbg::lasted time