CEVA®
The DSP Powerhouse

CEVA-Toolbox™

Debugger Reference Guide Vol-II

Rev 15.1.0

August 2015

DSP Division

Documentation Control

History Table:

Version	Date	Description	Remarks
10.0	01/12/2012	First tracked version	Added on 24/6/2013
10.0.4	18/12/2013	Updated enable/disable exceptions CLI description	
10.0.5 draft	06/02/2014	Updated for CEVA-TL4	
10.1	09/03/2014	Updated 'start memory profiler' CLI	
10.1.1	21/05/2014	Updated 'emu read code' CLI. Removed decodeinrange option.	
10.0.2	20/08/2014	For CEVA-TLxxxx	
10.2	20/08/2014	For CEVA-XC: Updated "set source break" examples, moved "set break <funcName>" to "set break" CLI	
15.1.0	11/08/2015	Updated for SDT V15	

Disclaimer and Proprietary Information Notice

The information contained in this document is subject to change without notice and does not represent a commitment on any part of CEVA®, Inc. CEVA®, Inc. and its subsidiaries make no warranty of any kind with regard to this material, including, but not limited to implied warranties of merchantability and fitness for a particular purpose whether arising out of law, custom, conduct or otherwise.

While the information contained herein is assumed to be accurate, CEVA®, Inc. assumes no responsibility for any errors or omissions contained herein, and assumes no liability for special, direct, indirect or consequential damage, losses, costs, charges, claims, demands, fees or expenses, of any nature or kind, which are incurred in connection with the furnishing, performance or use of this material.

CEVA®, CEVA-XC™, CEVA-XC321™, CEVA-XC323™, CEVA-Xtend™, CEVA-XC4000™, CEVA-XC4100™, CEVA-XC4200™, CEVA-XC4210™, CEVA-XC4400™, CEVA-XC4410™, CEVA-XC4500™, CEVA-TeakLite™, CEVA-TeakLite-II™, CEVA-TeakLite-III™, CEVA-TL3210™, CEVA-TL3211™, CEVA-TeakLite-4™, CEVA-TL410™, CEVA-TL411™, CEVA-TL420™, CEVA-TL421™, CEVA-Quark™, CEVA-Teak™, CEVA-X™, CEVA-X1620™, CEVA-X1622™, CEVA-X1641™, CEVA-X1643™, Xpert-TeakLite-II™, Xpert-Teak™, CEVA-XS1100A™, CEVA-XS1200™, CEVA-XS1200A™, CEVA-TLS100™, Mobile-Media™, CEVA-MM1000™, CEVA-MM2000™, CEVA-SP™, CEVA-VP™, CEVA-MM3000™, CEVA-MM3100™, CEVA-MM3101™, CEVA-XM™, CEVA-XM4™, CEVA-X2™ CEVA-Audio™, CEVA-HD-Audio™, CEVA-VoP™, CEVA-Bluetooth™, CEVA-SATA™, CEVA-SAS™, CEVA-Toolbox™, SmartNcode™ are trademarks of CEVA, Inc.

All other product names are trademarks or registered trademarks of their respective owners

Support

CEVA® makes great efforts to provide a user-friendly software and hardware development environment. Along with this, CEVA® provides comprehensive documentation, enabling users to learn and develop applications on their own. Due to the complexities involved in the development of DSP applications that may be beyond the scope of the documentation, an on-line Technical Support Service (support@ceva-dsp.com) has been established. This service includes useful tips and provides fast and efficient help, assisting users to quickly resolve development problems.

How to Get Technical Support:

FAQs: Visit our web site http://www.ceva-dsp.com or your company's protected page on the CEVA® website for the latest answers to frequently asked questions.

Application Notes: Visit our website http://www.ceva-dsp.com or your company's protected page on the CEVA® website for the latest application notes.

Email: Use CEVA's central support email address support@ceva-dsp.com. Your email will be forwarded automatically to the relevant support engineers and tools developers who will provide you with the most professional support in order to help you resolve any problem.

License Keys: Please refer any license key requests or problems to sdtkeys@ceva-dsp.com. Refer to the *SDT Installation & Licensing Scheme Guide* for SDT license keys installation information.

Email: support@ceva-dsp.com
Visit us at: www.ceva-dsp.com

List of Sales and Support Centers

Israel	USA	Ireland	Sweden
2 Maskit Street P.O.Box 2068 Herzelia 46120 Israel **Tel:** +972 9 961 3700 **Fax:** +972 9 961 3800	1943 Landings Drive Mountain View, CA 94043 USA **Tel:** +1-650-417-7923 **Fax:** +1-650-417-7924	Segrave House 19/20 Earlsfort Terrace 3rd Floor Dublin 2 Ireland **Tel**: +353 1 237 3900 **Fax**: +353 1 237 3923	Klarabergsviadukten 70 Box 70396 107 24 Stockholm, Sweden **Tel**: +46(0)8 506 362 24 **Fax**: +46(0)8 506 362 20
China (Shanghai)	**China (Beijing)**	**China Shenzhen**	**Hong Kong**
Room 517, No. 1440 Yan An Road (C) Shanghai 200040 China **Tel:** +86-21-22236789 **Fax:** +86 21 22236800	Rm 503, Tower C, Raycom InfoTech Park No.2, Kexueyuan South Road, Haidian District Beijing 100190, China **Tel:** +86-10 5982 2285 **Fax:** +86-10 5982 2284	2702-09 Block C Tiley Central Plaza II Wenxin 4th Road, Nanshan District Shenzhen 518054 **Tel:** +86-755-86595012	Level 43, AIA Tower, 183 Electric Road, North Point Hong Kong **Tel:** +852-39751264 **:**
South Korea	**Taiwan**	**Japan**	**France**
#478, Hyundai Arion, 147, Gumgok-Dong, Bundang-Gu, Sungnam-Si, Kyunggi-Do, 463-853, Korea **Tel**: +82-31-704-4471 **Fax**:+82-31-704-4479	Room 621 No.1, Industry E, 2nd Rd Hsinchu, Science Park Hsinchu 300 Taiwan R.O.C **Tel**: +886 3 5798750 **Fax**: +886 3 5798750	3014 Shinoharacho Kasho Bldg. 4/F Kohoku-ku Yokohama, Kanagawa 222-0026 Japan **Tel:** +81 045-430-3901 **Fax:** +81 045-430-3904	RivieraWaves S.A.S 400, avenue Roumanille Les Bureaux Green Side 5, Bât 6 06410 Biot - Sophia Antipolis, France **Tel:** +33 4 83 76 06 00 **Fax:** +33 4 83 76 06 01

Table of Contents

List of Tables

17. Debugger Command Line Interpreter (DCLI)

17.1 Concept of the Debugger Command Line Interpreter

The debugger command line interface or DCLI, is a tool introduced with the V15 SDT releases for all CEVA architectures.

DCLI goal is to provide shell command line and scripting debugger capabilities.

In most of the cases it is backward compatible with the V10 CLI scripting commands and syntaxes.

The DCLI executes legacy V10 debugger scripts from Windows and Linux command line shells.

It serves as a bridge between the legacy V10 debugger scripting capabilities and the future eclipse scripting that will be introduced in the eclipse based tools.

To start DCLI, enter "> dcli" in windows or Linux command shells.

Example:

- dcli --core CEVA-XM4

Without parameters, dcli displays a list of the available command line options:

Example:

```
>dcli
Please specify CEVA core name
Allowed options:
  -h [ --help ]                    This help text
  -c [ --core ] arg                Mandatory parameter - CEVA core name. E.g.
                                   "CEVA-XC4500"
  -t [ --target ] arg              Target definition file
  -s [ --script ] arg              CLI script file name. Without this option,
                                   the programs works in interactive mode.
  -b [ --beg ] arg                 CLI commands to be sent prior script
                                   execution
  -e [ --end ] arg                 CLI commands to be sent after script
                                   execution
```

```
-o [ --output ] arg               Redirect console output to a file
-d [ --dbgserver ] [=arg(=11234)] Start a remote debugger server with
                                  optional port (default: 11234)
-v [ --version ]                  Print the debugger API version
```

The built in TCL scripting language enables the programmer to create powerful scripts, which may contain conditionals, loops, and file I/Os. For a summary of the TCL features, please refer to the CEVA-Toolbox Debugger Reference Guide Vol-III.

17.2 General Guidelines

(1) Only one command is allowed per line.
(2) Only the first 200 characters of a line are parsed and echoed in the Log window.
(3) Trailing comments signified by a semicolon (;) are allowed, except in the **dos** command.

(4) Lines that begin with an exclamation mark (!) in the first column are also comments, but they will not be echoed in the Log Window. When logging the Log window, system responses are prefixed with an exclamation mark. The exclamation mark (!)means the comment will be ignored during playback.

(5) Blank lines are allowed and will be echoed in the Log Window.
(6) The CLI is case-sensitive.
(7) When an abbreviated label name conflicts with a reserved word, the label must be fully specified, for example:

```
use MySeg ; data segment
open = 1 ; will cause a syntax error since "open" is reserved
MySeg.open = 1 ; this statement is ok
```

(8) CLI commands may be preceded by a caret (^). Such commands are referred to as high priority commands and will be placed at the head of the CLI command queue. The caret command will interrupt the execution of the Simulator, the Emulator, and script files.

(9) Environment variables can be used in CLI commands (in TCL mode as well) using the %% operators. This is in addition to the native support for environment variables in the TCL language.

(10) The syntax and semantics of numeric and address expressions are very similar, although not identical, to the C/C++ language. The operator hierarchy is also very similar to C/C++. Following is the list of major differences between C/C++ expressions and those supported by the Debugger:

(a) The C/C++ address-of operator (unary &) is not supported, because all symbols are addresses.

(b) The syntax of addresses is different. Addresses are segmented into either the code page or the data page. Examples of code addresses are C:F000 and 0:1000. Examples of data addresses are d:ab2d and 1:2468.

(c) **Note** that the syntax of the conditional evaluation operator, i.e.

expr, "?", expr, ":", expr

conditional operator has lower precedence. Segmented addresses demand that there be no blank space before or after the colon (:), whereas in the case of the conditional evaluation operator blank spaces can precede or follow the colon. Therefore, to be safe, always have a blank space preceding and following the colon in the conditional evaluation operator, e.g.:

r0 = (5) ? 1 : 0 ; ok
r0 = (5) ? 1:0 ; not ok - the 1:0 will be mistaken for a data address expression

(d) A symbol expression can contain at most one dot (.) operator. When the right argument of the dot operator is a number or numeric expression, its meaning is that of a uni-dimensional array, e.g. MySeg.15

(e) The C array operator ([..]) is not supported. Syntactically, the same operator has been used as the address range operator, e.g.
copy [C:0000,C:10FF] output.dat

(f) A logical xor operator (^^) has been added.

(g) An address offset operator (offset) has been added, e.g.
offset MySeg.Myoffset

(h) A binary format has been added, e.g. 0b10010.

(i) The ordered list of operators (from strongest to weakest) is:
( ) group operator
[ ] address range operator
. dot operator
offset address offset operator
~ bit complement operator
! logical not operator
unary + positive operator
unary - sign change operator
unary * address de-reference operator
* multiplication operator
/ integer division operator
% modulo operator
+ addition operator
- subtraction operator
>> arithmetic shift right operator (sign extended)
<< shift left operator
== equality test operator
!= not equal test operator

< less than test operator
> greater than test operator
<= less than or equal test operator
>= greater than or equal test operator
& bit wise and operator
^ bit wise xor operator
| bit wise or operator
&& logical and operator
^^ logical xor operator
|| logical or operator
expr ? v1: v2 conditional operator
= assignment operator
|= bit wise or assignment operator
^= bit wise xor assignment operator
&= bit wise and assignment operator
<<= bit shift left assignment operator
>>= arithmetic shift right assignment operator
*= multiply assignment operator
/= divide assignment operator
%= modulo assignment operator
+= addition assignment operator
-= subtraction assignment operator
, comma operator

(j) Assignments cannot be embedded in numeric expressions, e.g. the following is incorrect syntax:

r0 = (r5 = 7) + 8

(11) Some CLI commands allow certain reserved words to be omitted. It is good practice to always use the long, explicit syntax. Examples are the commands that are applicable to memory-mapped I/O ports:

connect port, disable port, disconnect port, enable port, protect, rewind, unmap, unprotect

See also:

Message Management

CLI Commands

17.3 The Script File

Script files are ASCII text files containing CLI commands and comments. Each command must be placed on a separate line and follow the rules described in the General Guidelines Section. Script files can be nested up to 8 levels. Script files can be generated either off-line by any text editor prior to executing the Debugger, or by using a log file.

As mentioned before all CLI's are echoed to the Log window. For future use the Log window content can be saved as a text file: mouse right click on the window and choose save.

See also:

Message Management

CLI Commands

17.4 Message Management

All messages in the Debugger are divided into the following categories:
(1) assert - internal software error
(2) fatal - non-recoverable error
(3) error - error which does not allow the selected command to run to completion.
(4) warning - error which does not stop the selected command from running to completion.
(5) info - useful information message, e.g. breakpoint has been reached.
(6) debug - debug information for system developers.

Initially, all message categories are enabled, i.e. they are displayed in a pop-up message box and in the Log Window. Messages displayed in a pop-up message box need user-acknowledgement before the Debugger can continue. When a message category is disabled, messages of that type no longer appear in a pop-up message box, but they continue to appear in the Log Window. When automatically running a script file that cannot be interrupted, regardless of whether any errors occurred, all messages must be disabled, otherwise the script file will be blocked when an error occurs. When running the Debugger interactively, it is suggested not to disable any messages except perhaps those from the info category.

The CLI commands that control the messages to be displayed are enable message and disable message. Alternatively, it is also possible to control the appearance of messages by the Messages dialog box available from the Debug Menu.

Examples:

enable message all
disable message info

17.5 Exception Management

In Simulation Mode, certain exceptions can be disabled or enabled. When an exception is disabled, no message is displayed when the exception occurs. The following are exceptions that can be disabled/enabled:

(1) Stack overflow

(2) Modulo constraint violation

(3) Uninitialized memory read

(4) All warning exceptions

(5) Unmapped memory exceptions

In Emulation Mode, the above exceptions are disabled, that is, the hardware does not stop running following exception occurrence.

The CLI commands that control the exceptions to be displayed are enable exception and disable exception.

Examples:

disable exception modulo
disable exception warning
disable exception unmap

See also:

CLI Commands

17.6 The CLI Commands

17.6.1 assemble

assemble Address "Instruction"

The **assemble** command is used to invoke the In-Line Assembler at the specified code address, causing the current instruction at the address to be overwritten. The address must be mapped. No labels may be defined or referenced. Other than the DW directive, no directives may be used (see note below). The DUP, OFFSET, IMMEDOFFSET, and SHR operators may not be used. If the command causes half of the immediately preceding or following double-word instruction to be overwritten, then it will turn that instruction into a **nop**. The assemble command is useful for temporarily patching a program without editing the source files, invoking the macro Assembler and Linker, reloading the program and restarting the debugging process. It should be used with care, though, since symbols, directives and most operators are not allowed. The In-Line Assemble Option is also available through manually editing the code memory (Left click on the **mapped** address and editing it).

For more information, refer to the In-Line Assembler item in the Memory Menu.

Notes:

1. If a breakpoint was previously set on the old (replaced) instruction, this breakpoint will be canceled (removed). It must be set again in the new instruction if the breakpoint is needed.
2. The assemble command supports segments (paging) as well

Examples:

assemble 0x43F "mov r0,r1" ;

assembles "mov r0,r1" into current segment address 0x43F

assemble C3:4000 "inc a0" ;

assembles "inc a0" into segment 3 address 0x4000

Related Commands:

Assignment Commands

17.6.2 Assignment Commands

The **assignments** command are used to assign a new value(s) to the specified data memory address(es), register, flag, stack position or clock. If the data memory address is mapped as a "writable" I/O port, the 'output' command must be used to change its value.

Examples:

1. Assign the value 0 to data address 1:
1 = 0
2. Assign the value 1 to data address 2:
(23-21) = 1
3. Assign the value 2 to data address 3:
D:3 = 2
4. Assign the value 3 to data address 0xABCD:
D:ABCD = 2+1
5. Assign the value 4 to address Seg.Offset:
Seg.Offset = 4
6. Assign the value of register r0 to data address Seg.0x67:
Seg.0x67 = r0
7 Toggle carry flag:
c ^= 1
8. Add 17 to current value of register r0 & assign it to register r0:
r0 += 17
9. Multiple value of register a0 and value of register a1:
*a0 *= a1*
10. Assign half the current clock value to clock:
clock /= 2
11. Arithmetic shift right 3 the value of stack7, assign it to stack7:
stack7 >>= 3
12. The following command enables any code memory word to be changed (CodeAddress = value):
C:4FF1 = 23
13. Assign the specified values to the specified memory range:
[D:8000,4]={0x12 0x34 0x56 0x78}
14. Fill the specified memory range with zeros:
[C1:3000,C1:3100]={0}
15. Assign a value to symbol:

_foo = 0x100 ; when _foo is symbol
_foo + 2 = 0x100

16. Write a value to a write buffer register (CEVA-X and CEVA-XC only)
<writeBufferRegister> [value] = <list of the values space separated>

The <writeBufferRegister> must be one of the 8 write buffer registers: wb_a, wb_b, wb_c, wb_d, wb_e, wb_f, wb_g, wb_h.
The write buffer is only relevant for Cycle-accurate Simulator Pipeline mode.

Examples:
wb_a=0x12 0x34 0x56 0x78 ; ➔ double word written to register wb_a
wb_a=0x12 0x34 ; ➔ word written to register wb_a
wb_a value=0x12 0x34 0x56 0x78 ; ➔ equivalent to wb_a=0x12 0x34 0x56 0x78

17. Modify the write buffer register target address (CEVA-X and CEVA-XC only)
<writeBufferRegister> target = 'DataAddressExpression'

The <writeBufferRegister> must be one of the eight write buffer registers: wb_a, wb_b, wb_c, wb_d, wb_e, wb_f, wb_g, wb_h.
The write buffer is only relevant for Cycle-accurate Simulator Pipeline mode.

Examples:
wb_a target =0x1000
wb_a=0x12 0x34 0x56 0x78
; ➔ double word 0x78563412 is written to address 0x1000

wb_a target =0x1000
wb_a=0x12 0x34
; ➔ word 0x3412 is to address 0x1000

wb_a target =0x1000
wb_a=0x12 0x34 0x56 0x78
; ➔ double word 0x78563412 is written to address 0x1000
wb_a=0x9a 0xbc
; ➔ word 0xbc9a is written to address 0x1000 , the word starting at address 0x1000 remains unchanged = 0x7856

wb_a target =0x1002
wb_a=0x9a 0xbc

; ➔ word 0xbc9a will be written to address 0x1002

wb_a=0x12 0x34 0x56 0x78

; ➔ double word 0x78563412 will be written to address **0x1000** because the address 0x1002 is not aligned.

18. Write a word or double word to memory:

d:<startAddress>[,<numCells>]=value;

d:0,2 = 0x1234
d:0 = 0x12
d:0,1 = 0x12
d:4,4 = 0x12345678

19. Assign value to a full vector:
<vector name>:<vector unit> = value;

The value can be up to 256 bits. The CLI is relevant for CEVA-XC only

Examples:

voa:1 = 0xabcdabcdabcdabcdabcdabcd
vib:0 = 0xabcdabcdabcdabcdabcdabcd

20. Get the value of a full vector:
?<vector name>:<vector unit>;

The CLI is relevant for CEVA-XC only.

Examples:

?voa:1
?vib:0

21. Write to MSS registers using register name or CPM address

<mssFlagName> = <value>|<text_value>
<mssRegisterName> = <value>
CPM:<address> = <value>

Examples :

pcc = cache_locked *;* ➔ Sets pcc flag to cache locked.

```
pcc = cache_locked          ; ➔ Sets pcc flag to 2 (cache locked)
mss_dmba = 0x10000          ; ➔ Sets data memory base address to 0x10000
CPM:14  = 0x10000           ; ➔ Sets data memory base address to 0x10000 using cpm.
```

Related Commands:

etp read, etp write, assemble, evaluate, define symbolName

17.6.3 call

call **FileName**

The **call** command causes the Debugger to start accepting commands from the specified script file. When the end of the file is reached, execution returns to the point from where the script file was invoked, i.e. nestable script files are allowed. Execution of script files can be temporarily suspended via the '^pause' command.

Notes:

1. **The default file name extension are .dbg and .log**
2. **@ (followed by the filename) can be used instead of "call"**

Examples:

1. Invokes 'myscript.dbg' from current directory:
call myscript
2. Invokes 'myscript.x' from current directory:
@myscript.x
3. Invokes 'f:\debug\myscript.dbg':
call f:\debug\myscript.dbg
4. Invokes %MyDbgDir%\myscript.dbg (using environment variable support):
call %MyDbgDir%\myscript.dbg

Related Command:

pause

17.6.4 cancel break

cancel break Address
cancel break data value
cancel break data combined
cancel break RegName
cancel break extreg
cancel break clock
cancel break all

The **cancel break** command is used to cancel a breakpoint at a specified address, data value, combined data address and data, registers, external registers, or all breakpoints.

Examples:

cancel break C:FF00
cancel break D:0100
cancel break Seg.Offset
cancel break data value
cancel break data combined
cancel break r0
cancel break extreg
cancel break clock
cancel break all

Related Commands:

disable break, enable break, set break

17.6.5 cancel interrupt

cancel interrupt nmi/int0/int1/int2/vint/all

The **cancel interrupt** command cancels the simulation of the specified interrupt, or of all interrupts.

Note:

This command is valid in Simulation Mode **only**

Examples:

cancel interrupt int0
cancel interrupt all

Related Commands:

disable interrupt, enable interrupt, generate interrupt

17.6.6 cancel source break

cancel source break fileName lineNumber

Cancels a breakpoint at a specific line number in a specific source file.

Examples:

cancel source break hello.c 12
set source break main.c main

Related Commands:
cancel break, set source break

17.6.7 cancel symbols location

cancel symbols location

The **cancel symbols location** command is used to cancel the user defined symbols location which set by the set symbols location CLI command and return to the default Debugger symbols ambivalent resolution behavior. That is, pop-up a dialog box requesting the user to choose the requested symbols (whenever a symbol is located in both code and data memories).

Examples:

set symbols location incode
…
cancel symbols location

Related Commands:

load coff, load coff symbols, load section symbols, disable load coff reset, enable message, set symbols location

17.6.8 cd

cd PathName

The cd command is used to change the default device and directory path, when accessing files that don't have a complete device and directory path specification.

Examples:

cd e:
cd f:\branch1\branch2

Related Commands:

call , load coff , load coff symbols, copy , start log, start trace

17.6.9 config cevax type

config cevax [type] CEVA-XTypeName

This CLI is relevant for the CEVA-X only.

The **config cevax type** command is used to configure the core type of the CEVA-X Debugger.

CEVA-XTypeName can be one of the following:

For CEVA-X 16-Bit: 1641, 1643

Note:

The Debugger automatically configures the core type according to the COFF file loaded.

Example:

config cevax 1643 ;configure the Debugger to be of CEVA-X1643
?cevax [type] ;prints "CEVA-X1643"

Related Commands:

config simul, config test mode, etp read, etp write

17.6.10 config cfileio

config cfileio ansi/realtime [cacheSize] [maxNumberOfFiles]

The **config cfileio** command main role is to configure the Debugger to C-language File I/O mode (default is the old-style, low-level data I/O, using DSP_read and DSP_write). The Debugger will configure itself automatically to C File I/O when a C/C++ application that is compiled with the '–mfileio' Compiler invocation switch is loaded.

This CLI command can configure the nature of the C File I/O functions and some File I/O administration characteristics. Three parameters are configured through the **config cfileio** command:

- **ansi** means that all I/O library (***fileio.lib***) functions (e.g. fopen, fclose, fgets,…) will wait for return values as the ANSI-C definition of these functions. **realtime** means that the library function will not wait and will always return a "SUCCESS" return value. When real-time is important, it is useful to use the **realtime** option in order not to waste the time of the DSP application on waiting (polling) for the Debugger's response (default is **realtime**).
- **cacheSize** determines the cache size held in the C-language FILE type. Note that a larger cache size means better real-time performance. On the other hand, a larger cache size consumes more data memory (default is 250 Words).
- **maxNumberOfFiles** determines the size of the file table that the DSP application holds. This file table restricts the maximum number of files that the application can handle at a given moment (default is 20).

The command can be generated through the CFileIO dialog box I/O Menu as well.

Examples:

config cfileio ansi
config cfileio realtime 100
config cfileio ansi 100 10

Related Commands:

enable cfileio debug

17.6.11 config demangle

config demangle 0/1

The **config demangle** command (“config demangle 1”) is used to activates the C++ related symbol demangling mechanism of the Debugger. e.g. in the code window, in breakpoint symbol interpretation, etc.

The “config demangle 0” disables the demangling, showing the symbols in their original C++ name - this is the default.

Examples:

config demangle 1

This way the user can type “set break foo(int, float)”
Instead of “set break _foo@if3**”**

17.6.12 config int size

config int size 16/32

The **config int size** command is used to change the view format of a variable.

In CEVA-TeakLite-III & CEVA-TeakLite-4 Integer size is:

- 16 bits when using compatible mode
- 32 bits when using native mode.

The instruction is used to switch between the above formats in order to display the variable correctly in the locals and watch windows.

Examples:

1. Instructs the Debugger to use 16 bit format:

config int size 16

2. Instructs the Debugger to use 32 bit Format:

config int size 32

Related documents:

For a full overview of a variable size, refer to the *C/C++ Compiler User's Guide.*

17.6.13 config floating point format

config ieeefloat/fastfloat

The **config floating point format** command is used to enable the usage of the Fast Floating Point Format or IEEE floating point format by and instructing the Debugger which format to use in order to display the "float" type variables in the locals and watch windows.

In CEVA-TeakLite-III & CEVA-TeakLite-4 32-bit and CEVA-X, the default is IEEE float.

Examples:

1. Instructs the Debugger to use IEEE format:

config ieeefloat

2. Instructs the Debugger to use Fast Floating Point Format:

config fastfloat

Related documents:

For a full overview of the Fast Floating Point, refer to the C/C++ Compiler Manual.

17.6.14 config mss

config mss enable | disable | disable code | disable data
config mss code blocks 1 | 2
config mss data blocks 2 | 3| 4| 8
config mss data block width 1 | 2
config mss data dma width 8 | 16
config mss data bank width 16 | 32
config mss external code width 2 | 4
config mss code block size 32 | 64
config mss data block size 16 | 32
config mss data cache size 0 | 8 | 16 | 32 | 64
config mss axi width 64 | 128
config mss data cache size 0 | 8 | 16 | 32 | 64 axi width 64 | 128

This CLI is relevant for the CEVA-X/CEVA-XC/CEVA-TeakLite-III & CEVA-TeakLite-4 only.

- **Config mss:**

The **config mss** command is used to configure the MSS of the Cycle-accurate Simulator. When disabling the MSS, the relevant modules behave similarly to the Instruction Set Simulator. This means no arbitration, no usage of write buffer and immediate (zero wait-states) fetch from external memory. By default the MSS is enabled.

Following are MSS configurable attributes:

- **Config mss code blocks(CEVA-X, CEVA-XC only):**

  Number of internal code memory blocks (1 / 2 -> CEVA-X1620 default is 2).

- **Config mss data blocks:**

  Number of internal data memory blocks. In CEVA-TeakLite-III & CEVA-TeakLite-4the command should be called before “config pipeline”, otherwise the default value will be used.

  CEVA-X , CEVA-XC323 – 2 | 4
  CEVA-XC4XXX – 4 | 8
  CEVA-TeakLite-III, CEVA-TeakLite411, CEVA-TeakLite420, CEVA-TeakLite421 – 2|3
  CEVA-TeakLite411 – 2 | 3 | 4

- **Config mss data block width(CEVA-X, CEVA-XC only):**

  Data memory block width 1 | 2 * width of LS unit bandwidth.

- **Config mss data dma width(CEVA-X, CEVA-XC only):**

  Data memory DMA bandwidth in Bytes .

- **Config mss data bank width(CEVA-X, CEVA-XC only):**

  Data memory internal bank width.

- **Config mss external code width(CEVA-X, CEVA-XC only):**

  External Code memory port width. The denominator of the fetch line width . (2-> 0.5 * IFW / 4 -> 0.25 * IFW).

- **Config mss code block size(CEVA-X, CEVA-XC only):**

  Internal code memory block size in K-bytes (32 / 64K).

- **Config mss data block size(CEVA-X, CEVA-XC only):**

  Internal data memory block size in K-bytes (16 / 32K).

- **Config mss data cache size (CEVA-X 1643 only):**

  Data cache block size in K-bytes (0, 8, 16, 32, or 64 -> CEVA-X1643 default is 32K).

- **Config mss axi width (CEVA-X 1643, CEVA-XC only):**

  AXI bus width in bits: 64 or 128 bit. (CEVA-X1643, CEVA-XC default is 128 bit).

Notes:

-

1. Setting the configuration parameters in Emulation mode must fit the actual EDP parameters, otherwise undefined behavior is expected.

Examples:

config mss disable code
config mss code block size 64

Related Commands:

config simul, etp read, etp write

17.6.15 config simul

config [simul] instruction [mode]
config [simul] [cycle] pipeline [mode]

This command is relevant for the CEVA-X, CEVA-XC,TeakLite-III and CEVA-TeakLite-4 only

The **config simul** command is used to select between the two Simulator optional modes. Each mode provides different simulation accuracy.

The following simulation modes are available:

- **Instruction Set Simulator mode:**

  In this mode the Simulator executes one instruction packet per step including all its associated pipeline stages. The Simulator executes the instruction in a single stage unlike normal operation when the instruction is broken into pipeline stages. As a result this mode is faster but less accurate.

- **Cycle-accurate Pipeline Simulator mode:**

  In this mode the Simulator performs one core cycle per step. The Simulator holds all the pipeline information of the core, simulates the core interface and can display the pipeline information too.

Examples:

? debugger mode *; prints " Instruction Set Simulation "*

config cycle pipeline mode

? debugger mode **:** *prints "Cycle Accurate Pipeline Simulation*

config simul instruction mode

? debugger mode **:** *prints "Instruction Set Simulation"*

17.6.16 config test mode

config test mode [clock maxClock] [interrupt intType [vectorAddress]]

This CLI is relevant for the CEVA-X and CEVA-XC only.

The **config test mode** command is used to configure Simulator to work in testing mode.

Under this mode all the testing related changes takes effect. This command is used for testing the Simulator vs. the hardware model.

- maxClock - If used this option causes the Simulator to break after maxClock cycles.
- intType - interrupt signal to use for single stepping interrupt mechanism. The interrupt signal will be asserted once for each instruction.
- vectorAddress - Target address for the vector interrupt to jump to. Should be used when intType is vint.

Example:

config test mode ; configures the Debugger to be in test mode.

Related Commands:

config simul, etp read, etp write

17.6.17 connect external debugger

connect external debugger DebuggerUniqueName

The **connect external debugger** command is used to establish a connection to another CEVA-Toolbox Debugger. The command changes the current connection to the newly connected Debugger. After a connection has been established, any CLI can be issued to the **active** connected Debugger using the run external cli command.

This method of connection between multiple CEVA-Toolbox Debugger is supported in addition to other communication methods, such as: DDE (see appendix), UserDBG.DLL, etc.

Notes:

1. This command changes the active connection.
2. It is possible to connect up to ten connections at a time, therefore it is recommended to close unused connections.
3. Before using the cli open another debugger instance

Examples:

start cevaxcdbg -ddeConnectName-secDbg_cevaxc4210 disable.dbg

*connect external debugger secDbg_cevaxc4210***Related Commands:**

disconnect external debugger, run external cli, transfer

17.6.18 connect port

connect [**file**] **FileName** [**port**] **PortAddress** **input** [**nowrap**/**wrap**]
[**OffsetInFile**] [**binary**/**text**/**word**/**dword**]
connect [**file**] **FileName** [**port**] **PortAddress** **output** [**append**]
[**binary**/**text**/**word**/**dword**]
connect outwin **OutputWindowName** [**port**] **PortAddress**
[**hexadecimal**/**ascii**/**mixed**]
connect user [**port**] **PortAddress** [**input**/**output**]
connect user [**port**] **PortAddressRange** [**input**/**output**]
connect user [**port**] **all**

The **connect port** commands allow memory-mapped I/O ports to be simulated while debugging an application program with the simulator. A data address that already been mapped as an I/O device can be connected to a file, a user-defined DLL function or an Output Window. The port can be output-connected to a file, a user-defined DLL function or an Output Window at the same time.

For more information on connecting memory ports to user-defined DLL functions, see the UserIO Simulator Extension DLLCEVA-Toolbox Debugger User's Guide Vol-I.
The following restrictions and comments should be considered:

- Ports with input connections must be mapped with read access, that is **ior**, and ports with output connections must be mapped with write access, that is. **iow**.
- The same port can have both an input and output connection; provided that it is mapped accordingly that is **iorw**.
- Whenever a file is connected to an input port (in an hexadecimal file format), and the port is read, a new value is read from the current line of the file, and the file pointer advances one line.
- If a memory-mapped input port is connected to a user-defined function, a new value is obtained by calling the user-supplied DLL function UseIO_InPort().
- Reading from mapped but unconnected I/O ports causes a dialogue box to pop up, to allow the user to manually specify the value.
- Defining an output file connection deletes any previous version of the output file.

- Writing to an output port that is connected to a file appends the value being written to the end of the file.
- Input files that are connected with **wrap mode** are rewound to the beginning when the end-of-file has been reached. Input files not connected with **wrap mode** will be disabled when the end-of-file has been reached and a dialogue box will pop up in order to prompt the user to manually specify the value.
- Multiple input ports may be independently connected to the same file, i.e. each port has its own independent current line pointer.
- Output files that are connected in **append** mode will not be erased upon connection.
- Output files may be shared with other output connections, but may not be shared with other input connections. This means that multiple ports may be output-connected to the same file, but one file cannot be input and output-connected to the ports at the same time.
- The format of an I/O file can be one of the following:
  - **BYTE** format:

    A text file containing BYTE size values delimited by standard white space characters, specified in hexadecimal notation (0xXX) or decimal notation. Blank lines are skipped; comments following a semicolon (;) are permitted. This option isn't available in the CLI since it's relevant only for CEVA-X and CEVA-XC and is the default.
  - **WORD** format:

    A text file containing the specific DSP WORD size values delimited by standard white space characters, specified in hexadecimal notation (0xXXXX) or decimal notation. Blank lines are skipped; comments following a semicolon (;) are permitted. Output files will always be in hexadecimal notation without blank lines or comments.
  - **DWORD** format:

    A text file containing DWORD (double word) size values delimited by standard white space characters, specified in hexadecimal notation (0xXXXXXXXX) or decimal notation. Blank lines are skipped; comments following a semicolon (;) are permitted.
  - **Binary** format:

    An ordinary binary file with no limitations. The size of an item read / written is a single character.

- **Text** format:

  An ordinary text file with no limitations. The size of an item read / written is a single character.

- By default, I/O files are expected to be in the native core format (i.e. **WORD** for CEVA-TeakLite-III, **BYTE** for CEVA-X and CEVA-XC). Output files will be generated without blank lines or comments.
- Only one input and one output file connection can be opened at a time. The most recently used input and output port file handles are open. Therefore, in order to improve efficiency of applications that interleave their port usage, it is best to have the files saved on a RAM disk.
- The open output port cannot be shared by other applications.
- Connections are closed whenever they are rewound, disabled, a new program is loaded, or the command **dos** is given.
- Connecting an already connected file will close the old file and open the new file. Consequently, a pop-up dialogue box will precede this action and request confirmation by the user. If a connected port is shown on the screen, it will be highlighted in yellow and display the last value read from or written to the port. For more information on mapping I/O devices, see the map program .
- Ports mapped for output may be connected to a named Output Window. The values written to the port may be displayed in **hexadecimal** (one value on a line), **ASCII** (as a text file) and **mixed** (one hexadecimal value on a line with its ASCII equivalent) formats. The default display format is **hexadecimal**.

For more information on connecting ports to the UserIO DLL , see the connect user command.

For more information on connecting ports, see the I/O Ports and Internal Registers in the I/O menu CEVA-Toolbox Debugger User's Guide Vol-I..

Note: When connecting to an output file, ASCII codes (0xXXXX - hex numbers) corresponding to the string's characters are output to the file. By using the ***dspprnt.exe*** utility, it is possible to convert the stored hex numbers in the file (representing the ASCII codes of the printed string) into readable character strings. Similarly, the ***dspprnt.exe*** utility is capable of converting a text string back to its representative hex/ASCII numbers (by using the BACK

command line option and activating the ***dspprnt.exe*** for getting its usage). This can be useful for preparing text input files readable by the Debugger (as input-connected files).

Refer to the C/C++ Compiler User's Guide under "Simulation Support I/O" for more details.

Notes:

1. These port connection commands are valid only in Simulation Mode.
2. By default, input files do not wrap around at end-of-file.
3. By default, existing output files will be erased upon connection.

Examples:

connect %MyDir%\my.dat D:1234 input wrap

connect file my1.dat Xram1.CodecIn input

connect c:\oak\demo\out1.txt port Xram1.CodecOut output

connect user port [D:4000,D:400F] input

connect user port all

Related commands:

connect register, disable port, disable register, disconnect port, disconnect register , enable register, enable port, map program, map , rewind port , set user, protect port , protect register , rewind register

17.6.19 connect register

connect [file] FileName [register] extExtRegId input
[nowrap/wrap] [OffsetInFile] [binary/text/byte/word/dword]
connect [file] FileName [register] extExtRegId output [append]
[binary/text/byte/word/dword]
connect outwin OutputWindowName [register] extExtRegId
[hexadecimal/ASCII/mixed]
connect user [register] extExtRegId [input/output]

Note: This CLI is **irrelevant** for the CEVA-X and CEVA-XC cores.

The **connect register** commands allow I/O-mapped external registers to be simulated while debugging an application program with the Simulator. An external register already mapped as an I/O device can be connected to either a file, a user-defined DLL function oran Output Window.

External registers with output connections can be connected to a file, a user-defined DLL function and an Output Window at one time.

The following restrictions and comments should be considered:

- External registers with input connections must be mapped with read access, that is **ior**, and registers with output connections must be mapped with write access, that is **iow**.
- The same register can have both an input and output connection, provided that it is mapped accordingly, that is **iorw**.
- Whenever a file is connected to an external register, and the register is read, a new value is read from the current line of the file, and the file pointer advances one line.
- Reading from mapped but unconnected external registers causes a dialogue box to pop up to allow the user to manually specify the value.
- Defining an output file connection deletes any previous version of the output file.
- Writing to an external register that is connected to a file appends the value being written to the end of the file.

- Writing to an unconnected register, or to a disabled connection, causes a message box to pop up displaying the data value.
- Input files that are connected with **wrap mode** are rewound to the beginning when the end-of-file has been reached.
- Input files not connected with wrap mode will be disabled when the end-of-file has been reached, and a dialogue box will pop up in order to prompt the user to manually specify the value.
- Multiple registers with read access may be independently connected to the same file, i.e. each register has its own independent current line pointer.
- Output files that are connected in **append** mode will not be erased upon connection.
- Output files may be shared with other output connections, but may not be shared with other input connections. This means that multiple registers may be output-connected to the same file, but one file cannot be input and output-connected to the registers at the same time.
- Output files may not be shared with other output or input connections.
- The format of an I/O file can be one of the following:
  - **BYTE** format:

    A text file containing BYTE size values delimited by standard white space characters, specified in hexadecimal notation (0xXX) or decimal notation. Blank lines are skipped; comments following a semicolon (;) are permitted.
  - **WORD** format:

    A text file containing the specific DSP WORD size values delimited by standard white space characters, specified in hexadecimal notation (0xXXXX) or decimal notation. Blank lines are skipped; comments following a semicolon (;) are permitted. Output files will always be in hexadecimal notation without blank lines or comments.
  - **DWORD** format:

    A text file containing DWORD (double word) size values delimited by standard white space characters, specified in hexadecimal notation (0xXXXXXXXX) or decimal notation. Blank lines are skipped; comments following a semicolon (;) are permitted.
  - **Binary** format:

An ordinary binary file with no limitations. The size of an item read / written is a single character.

- **Text** format:
  An ordinary text file with no limitations. The size of an item read / written is a single character.

- By default, I/O files are expected to be in **WORD** format. Output files will be generated without blank lines or comments.
- Only one input and one output file connection is open at a time. The most recently used input and output file handles are open. Therefore, in order to improve efficiency of applications that interleave their port usage, it is best to have the files on a RAM disk.
- The open output port cannot be shared by other applications.

- Connections are closed whenever they are rewound, disabled, a new program is loaded, or the **dos** command is given.
- Connecting an already connected file will close the old file and open the new. Consequently a
- pop-up dialogue box will precede this action and request confirmation by the user. A connected external register is shown in the Register Window**,** highlighted (in yellow) and displaying the last value read from or written to the register.
- External registers mapped for output may be connected to a named Output window. The values written to the register may be displayed in **hexadecimal** (one value on a line), **ASCII** (as a text file) and **mixed** (one hexadecimal value on a line with its ASCII equivalent) formats.

The default display format is **hexadecimal**.

For more information on mapping I/O devices, see the map command.
For more information on connecting ports, see the **I/O Ports and External Registers** in the I/O menu**.**
For more information on connecting external registers to the UserIO DLL (ASSYST), see the connect user command.
Note: When connecting to an output file, ASCII codes (0xXXXX - hex numbers) corresponding to the strings characters are output to the file. By using the ***dspprnt.exe*** utility,

it is possible to convert the stored hex numbers in the file (representing the ASCII codes of the printed string) into readable character strings. Similarly, the ***dspprnt.exe*** utility is capable of converting a text string back to its representative hex/ASCII numbers (by using the BACK command line option and activating the ***dspprnt.exe*** for getting its usage). This can be useful for preparing text input files readable by the Debugger (as input-connected files).

Refer to the C/C++ Compiler User's Guide under "Simulation Support I/O" for more details.

Notes:

1. This command is valid only in Simulation Mode.
2. By default, input files do not wrap around at end-of-file.
3. By default, existing output files will be erased upon connection.
4. By default, external registers are connected to user functions for both input and output.

Examples:

connect %MyDataDir%\mydata.out ext0 output
connect user register ext2 output
connect user ext5
connect outwin Output_ExternalRegister3 register ext3

Related commands:

connect user ,connect port, disable port , disable register , disconnect port, disconnect register, enable port , enable register , map memory , map register, rewind port , set user , disable load coff reset , protect port , protect register , rewind register

17.6.20 connect user

connect user [**port**] **Address** [**input**/**output**]
connect user [**port**] **AddressRange** [**input**/**output**]
connect user [**port**] **all**
connect user [**register**] **ext**ExtRegId [**input**/**output**]
connect [**user**] **iu0**/**iu1**

The **connect user** commands allow memory-mapped I/O ports to be simulated while debugging an application program with the Simulator. A data address already mapped as an I/O device can be connected to either a file, a user-defined DLL function oran Output Window. Ports with input connections must be mapped with read access, i.e. **ior**, and ports with output connections must be mapped with write access, i.e. **iow**. The same port can have both an input and output connection, provided that it is mapped accordingly, i.e. **iorw**.

If a memory-mapped input port is connected to a user-defined function, a new value is obtained by calling the user-supplied DLL function UserIO_InPort(). Reading from mapped, but unconnected, I/O ports causes a dialogue box to pop up and prompt the user to manually specify the value.

If a memory-mapped output port is connected to a user-defined function, a new value is sent to the user-supplied DLL function UserIO_OutPort(). Writing to an unconnected I/O port, or to a disabled connection, causes a message box which displays the data value written to pop up.

The **connect user port all** command can be used to instruct the Debugger to connect to the user functions all the ports which were not connected to file, Output Window user. This enables the user to connect a large data memory area to the UserIO DLL, thus allowing it to simulate a vast array of memory behaviors. The ports that are connected by this command may not be selected to be disconnected/disabled/enabled, except through use of the disconnect user command.

The **connect user register** command can be used to connect an external register mapped as I/O to the UserIO functions. If an external register is mapped for input and is connected to a user-defined function, the value is obtained by calling the DLL function UserIO_InExtReg(). Reading from mapped, but unconnected, external registers causes a dialogue box to pop up to allow the user to manually specify the value. If an external register is mapped for output and is connected to a user-defined function, the value is sent to the DLL function UserIO_OutExtReg(). Writing to an unconnected register or to a disabled connection causes a message box which displays the data value written to pop up.

The **connect user iu0/iu1** command allows user input pins (**iu0, iu1**) to be simulated with the DLL library while debugging an application program in Simulation Mode. An input user pin being already connected to the DLL might be changed by the DLL library in order to affect the simulated DSP Core application program. Note that the output user pins are always connected to the user-defined DLL function, so when modified can signal the DLL library to change its functional behavior.

For more information on the UserIO DLL functions, see The UserIO DLL Functions.

Notes:

These commands are valid only in Simulation Mode.

Examples:

```
connect user port CodecOut output
connect user port [D:4000,D:400F] input
connect user port all
connect user register ext2 output
connect user iu0
connect iu1
```

Related Commands:

connect port , connect register , set user

17.6.21 continue

The **continue** command when invoked with a caret, i.e. **^continue**, restarts execution of a go/run command or a command file that had been temporarily suspended via the **^pause** command; otherwise the command has no effect.

Examples:

^continue

Related Commands:

call, go, next, pause, run

17.6.22 copy

copy AddressRange FileName
[binary/text/word/dword] [**append**]
copy FileName AddressRange [**OffsetInFile**]
[binary/text/word/dword]

The **copy** to file command allows the contents of code memory or data memory to be copied to a file. When the **append** option is used, instead of overwriting its previous contents, the contents of the memory range are appended to the end of the specified file. The default is to overwrite the previous contents of an existing file.

The copy from file command allows the values in a file to be copied to either data or code memory. COFF format files can be loaded via the load coff command. When copying from a file to data memory, the number of values copied is the minimum of the number of values in the file and the length of the range. **OffsetInFile** may be used to denote the first file value from which to start copying.

notes:

1. Addresses copied (from/to) need to be mapped first
2. The copy will stop prematurely when an unmapped address or I/O port address is encountered.
3. The file format is the same as the format used for I/O connections (see the **connect** commands for details).

Examples:

copy [C:0000,C:FFFF] %MyProjDir%\prog.dat
copy [D:0000,D:FFFF] data.dat append

Copy 0x20 values from xram.dat to address 0x0000 (and on) in the data memory:

copy xram.dat [D:0000,0x20]

Copy the contents of input.dat to addresses 0x0000 - 0x1FF0 in the code memory starting from the 100th value in the file:

copy %MyInputDir%\input.dat [C:0000,C:1FF0] 100
copy c:\data.ini [C:1100,0xFF]

Related Commands:

load coff, load coff symbols, fill, connect port, connect register

17.6.23 ctag

<coreAddress> = miss/ hit [nocopy]

This CLI is relevant for the CEVA-X, CEVA-XC and TeakLite-III only.

Ctag:

Assign mis or hit values into a cache address.

Examples:

ctag: 0x1000 = miss

?ctag: 0x1000 -> evaluate

Related Commands:

etp read, etp write

17.6.24 debugger new session

The **debugger new session** command is used to start a new debugging session. Issuing this command result in the following actions:

- The return of the Debugger to the startup state once invoked (Simulation mode)
- All memory and external registers are unmapped and de-allocated from the Debugger's memory usage
- All ports, external registers and Output Window are disconnected
- All breakpoints are removed
- Initializes messages and exceptions status with default values

  Note - By default, all exceptions and levels of messages severity are enabled.
- Initializes internal registers to their default values
- If in Emulation Mode, the Debugger reverts to simulation
- If in TCL mode, the Debugger exits TCL Mode
- If in Log Mode, the Debugger exits Log Mode
- If in Profiling Mode, the Debugger exits Profiling Mode.

Notes:

1. With every mapping command the Debugger allocates more memory, which increases the memory usage of the Debugger and which may slow the PC down - **make sure to map only the necessary memory area**.
2. The **unmap** command unmaps memory that was mapped by the map command - it doesn't free it from the Debugger's memory usage. In order to free mapped memory, use the **debugger new session** command.
3. **It is good practice to use this command when starting a new script file that contains mapping commands of previously mapped areas**. Note that when using the **debugger new session** command, it will bring the Debugger to the state it was in when it was first invoked; thus, any memory that has been mapped already and its contents will be erased.

Examples:

debugger new session

Related Commands:

map , map program, unmap, call, connect port, connect register ,set break , start tcl, start log, start cevatl, start memory profiler

17.6.25 deb stop tcl

deb stop tcl

The **deb stop tcl** command is used to exit TCL mode and returns the Debugger to normal CLI Mode. The command is valid only when in TCL Mode. Note that the prefix **deb** is used, as this is a sign for the Debugger that this command is a CLI command.

For more information, refer to CEVA-Toolbox Debugger User's Guide Vol-III.

Examples:

deb stop tcl

Related Commands:

start tcl

17.6.26 define memory

define memory **MemoryName** [**MemoryIndex**] [**wait** **MemoryWaitStateCycles**]

The **define memory** command is used to define the different classes of memory used in the system and their aliases. These aliases must be used in the map command. The **define memory** command is similar to defining memory classes in the Linker script file. Note that three classes are predefined; thus, up to 13 new classes of memory may be defined.. The wait-states are defined in the DSP Core processor instruction cycle units. Wait-states affect the simulated clock time, and the simulated interrupts. The highest allowable wait state is 15.

The optional **MemoryIndex** indicates the memory class index, if no MemoryIndex is defined, the Debugger automatically assigns an index for the defined memory class.

The optional burst support switch can be used for memories with burst transaction support. The number of cycles of the burst should be specified in **numBurstCycles**. It is also possible to disable the burst support by using the switch **burstOff**.

The optional **freq** switch allows the memory clock frequency denominator to be specified. The memory clock frequency is specified with reference to the core clock frequency e.g. ½ , 1/3, ¼. The **frequencyDivisionDenominator** parameter should indicate the denominator of the memory clock frequency (e.g. 2 = 1/2, 3 = 1/3, 4 = 1/4).

Note:

1. When no wait-state is specified, a default of 0 is assumed.
2. Repeating the ‘define memory’ command with part of the attributes will not delete previous attributes given. This in order to support short commands.

Examples:

Syntax:

define memory MyType 3 wait 2
define memory MyMem
define memory myExternal burst 2
define memory myExternal freq 4

Using "define memory" command to configure external memory access wait states.

Assume external memory section *Block1* defined in Linker Script File (.lnk) as follows:

```
…
classes:
     Block1  [D:10000, D:1ffff]   external
…
Block1:
     segment 0
          my_data_segmet
…
```

Use *define memory* and *map* commands in a Debugger Script File as follows:

```
     start cevatl3210 code 64K data 64K
     ; adding memory definition for Block1
     define memory Block1 wait 10
     config pipeline mode
     load coff debug\WaitStates.a
     ; Map data external memory space to defined memory Block1
     ; The map needs to be done after loading the COFF file
     map [d:10000, d:1ffff] Block1
     start memory profiler
     go
     ; Any access in the range d:10000, d:1ffff will have 10 wait states
external memory access time.
```

Related Commands:

undefine memory, map, show memory classes, show memory map

17.6.27 define symbolName

AddressExperession

The **define symbolName** command is used to define a symbol as a representor to a specific address in the memory. After a symbol is defined, the value in this memory address can be viewed or modified by using the evaluate and assign CLI's with the symbolName.

The symbol can be defined for all possible address spaces:

1. Code memory: C:XXX
2. Data memory: D:XXX

CEVA-X, CEVA-XC, CEVA-TeakLite-III and CEVA-TeakLite-4 only:

3. I/O memory: IO:XXX

CEVA-X and CEVA-XC only:

4. Internal I/O memory: CPM:XXX

Examples:

```
define myReg D:1000     ; defines a register names myReg in address 0x1000 of data.
myReg = 0x100           ; sets the value of 0x100 in myReg
*myReg                  ; gest the actual value in myReg
```

Related Commands:

evaluate, Assignment Commands

17.6.28 disable asm source debug

This command sets the behavior mode of the source debug commands (**step source**, **next source**, **step out source**). When using this command, the debug information is taken only from C/C++ source files (will skip Assembly file information).

Example:

disable asm source debug

Related Commands:
etp read, etp write, enable asm source debug

17.6.29 disable branch to self

This CLI is relevant for the CEVA-X, CEVA-XC only.

This command sets the behavior of branch to self to disable. In disable mode, when reaching a branch to self, a message pops stating that we are in branch to self and stops the debugger run.

This is the default behavior of branch to self.

This behavior control allows the user to write an application that reaches at certain points with branch to self and enters an endless loop.

Example:

disable branch to self

Related Commands:

enable branch to self

17.6.30 disable break

disable break **BreakpointAddress**
disable break all

The **disable break** command is used to disable the breakpoint at the specified address or disables all breakpoints. They may be enabled using the **enable break** command. For more details on setting/creating address breakpoints, see the explanation of the **set break** command.

Examples:

disable break d:abcd
disable break main
disable break all

Related Commands:

cancel break, enable break, set break

17.6.31 disable cfileio debug

disable cfileio debug

The **disable cfileio debug** command is used to disable the C file I/O debug messages mode.

Examples:

disable cfileio debug

Related Commands:

enable cfileio debug, etp read, etp write

17.6.32 disable exception

disable exception internal read unmap/
external read unmap/
internal read uninit/
external read uninit/
internal write unmap/
external write unmap/
memory/
modulo/
stack/
restriction/
bubble/
unmap/
all

The **disable exception** command is used to disable the specified exception(s).

The following should be noted:

- Exceptions affect the Simulator only.
- At initialization, all exceptions are enabled.
- When the **external/internal read/write unmap/uninit** exception is disabled, the Simulator will not stop (or report) when uninitialized/unmapped data External/Internal memory is read/write.
- When the **memory** exception is disabled, the Simulator will not stop (or report) when uninitialized data memory is read.
- When the **modulo** exception is disabled, the Simulator will not stop (or report) when a modulo constraint violation has been detected (see also the specific DSP Core architecture specification).
- When the **stack** exception is disabled, the Simulator will not stop (or report) when stack over/under-flow conditions have been detected.
- When the **restriction** exception is disabled, new code (after ***movd***, inline Assembler...) will not be checked for restrictions (illegal instruction sequences).

- In Teaklite-III, when the restriction is disabled, the bubble remains enabled. Note that if the bubbles are disabled, the restriction is automatically disabled as well.
- When the **unmap** exception is disabled, all exceptions related to unmapped memory will not be issued.
- **Note** that the related **disable message** command stops message boxes and dialog boxes from popping up (that must be acknowledged by the user before the next CLI command is processed), but the Emulator/Simulator execution will be halted and the message will be written to the Log Window.

Notes:

1. This command is valid only in Simulation Mode.
2. **Debugger new session** CLI initializes messages and exceptions status with default values.
3. The **disable exception all** CLI doesn't affect the **unmap** exception due to compatibility reasons.

Examples:

disable exception stack
disable exception all

Related Commands:

disable load coff reset , enable exception, enable message

17.6.33 disable interrupt

disable interrupt nmi/int0/int1/int2/vint/all

The **disable interrupt** command is used to disable the simulation of the specified interrupt(s) (After generate the interrupt). The interrupt(s) can be enabled via the command enable interrupt. Simulated interrupts are generated/created via the command generate interrupt**.**

Note:
This command is valid only in Simulation Mode.

Examples:

disable interrupt int0

Related Commands:
cancel interrupt**,** enable interrupt**,** generate interrupt

17.6.34 disable load coff reset

disable load coff reset

The **disable load coff reset** command is used to disable the core reset during loading of a COFF file. By default the core reset in the load coff instruction is enabled (reset is performed).

Examples:

disable load coff reset

Related Commands:

enable load coff reset, load coff, etp read, etp write

17.6.36 disable message

disable message assert/
debug/
error/
fatal/
info/
warning/
dialog/
all

The **disable message** command is used to prevent message boxes from popping up when the specified message severity is reported. Message boxes demand acknowledgement from the user before any other commands is processed. The messages will still be output to the Log Window, and the Simulator/Emulator will stop execution and proceed to the next command in the queue. Messages always include the message severity. Note that messages with an assert severity signify a fatal internal software error. The **dialog** option is for disabling the dialog boxes pop-up when a read from an unconnected I/O port or register is encountered. Except when running large batch procedures, it is not a good practice to disable any severity level other than info.

Notes:

1. When dialog boxes are disabled, a zero value will be read.
2. **debugger new session** CLI initializes messages and exceptions status with default values

Examples:

disable message info
disable message all

Related Commands:

disable exception, enable exception, enable message

17.6.37 disable mss_reg counter

The **disable mss_reg counter** command is used for pausing the counter of the mss_reg.

This command set the relevant bit in the mss pause register.

Notes:

mss_reg can only be of the profiler counter regs in the mss

Examples:

disable prof_nop_inst counter

Related Commands:

enable mss_reg counter, reset mss_reg counter

17.6.38 disable overwritten instruction nop padding

disable overwritten instruction [nop padding]

The **disable overwritten instruction** is used to download a section to the code memory in run-time, overwriting an existing, pre-loaded code section, with the download starting from the second word of an instruction in the overwritten section, the decoding of the binary contents of the code memory by the Debugger may be incorrect. This Debugger command is designed to prevent this. When **overwritten instruction nop padding** is enabled, the Debugger automatically adds a *nop* instruction (to the remains of the overwritten instruction) before the new downloaded section to allow proper decoding.

Disabling **overwritten instruction nop padding** is useful when the first word of the new downloaded section should compose one instruction of two words along with the first word of the partially overwritten instruction of the overwritten section.

Examples:

disable overwritten instruction nop padding

Related Commands:

enable overwritten instruction nop padding, load section symbols

17.6.39 disable port

disable port PortAddress [input/output]
disable port PortAddress [input/output] file
disable port PortAddress [input/output] outwin
disable port PortAddress [input/output] user

The **disable port** command is used to disable the connection of a memory-mapped I/O port to a file, Output Window or user-defined DLL function. If file, outwinor user are used in the command, only this connection will be disabled. Otherwise all previous connections to this address will be disabled. The connection can be re-enabled with the enable port command. I/O port connections are created via the command **connect port**. While disabled, I/O port activity is redirected to dialog and message boxes, to which the user must manually respond before execution continues. For more details on I/O port connections, see the command connect port.

Notes:

1. This command is valid only in Simulation Mode.
2. If no I/O mode is given, both inputs and outputs are disabled.

Examples:

disable port D:0536
disable port 0xF047 output
disable port d:f7ff input file
disable port d:f7fe output outwin
disable port d:f7fe output user

Related Commands:

disconnect port , enable port, connect port, map , rewind

17.6.40 disable printing tcl calls

The **disable printing tcl calls** command is used to disable the printing of the TCL calls to the Debugger's CLI commands. In TCL Mode, each Debugger command (as opposed to a pure TCL command) must be preceded by the key word **deb**. This is the sign for the TCL interpreter to call the Debugger's Interpreter (CLI) with the command after the **deb** key word and not interpret it itself. By default, this call will cause the printing of the command to the Log Window.

Note:
This command is relevant only in TCL mode.

Examples:
1. For the following command the TCL Interpreter will call the Debugger with the set break main command, so if printing TCL calls is enabled, the following printings will appear in the Log Window:

The CLIs:
enable printing tcl calls
deb set break main

The output
set break main

2. If the disable printings TCL calls command has been previously issued, the only printing will be of the original command:

The CLIs:
disable printing tcl calls
deb set break main
Output:
No output

Related Commands:
start tcl , deb stop tcl, enable printing tcl calls

17.6.41 disable register

disable register ext ExtRegId **[input/output]**
disable register ext ExtRegId **[input/output] file**
disable register ext ExtRegId **[input/output] outwin**
disable register ext ExtRegId **[input/output] user**

This CLI is relevant for the TeakLite-III only.

The **disable register** command is used to disable the connection of an I/O-mapped external register to a file, Output Windowor user-defined DLL function. If file, outwinor user are used in the command only, this connection is disabled, otherwise all previous connections to this register will be disabled. The connection can be re-enabled with the enable register command. Register connections are created via the connect register command. While disabled, external register activity is redirected to dialog and message boxes, to which the user must manually respond before execution continues.

Notes:
1. This command is valid only in Simulation Mode.
2. If no I/O Mode is given, both inputs and outputs are disabled.

Examples:

disable register ext0
disable register ext3 output
disable register ext3 output file
disable register ext3 output outwin
disable register ext3 input user

Related Commands:
enable register , connect register, disconnect port , enable port , map , rewind , connect port

17.6.42 disable relative path to script

disable relative path to script

The **disable relative path to script** command configures the Debugger to treat all paths in a script file according to the current directory.

Example:

disable relative path to script ; Disable relative path to script file.

Related Commands:

enable relative path to script, etp read, etp write

17.6.43 disable rtl version

disable rtl [RtlVersion]

The **disable rtl version** command is used to disable the special RTL version restriction checking by the Debugger. For more details see the enable rtl versionCLI command.

Examples:

disable rtl 1.2.0

disable rtl

Related Commands:

enable rtl version

17.6.45 disassemble

disassemble [AddressRange]

The **disassemble** command is used to disassemble all the program memory mapped by the user as well as the updated program memory. Usually, the Debugger disassembles the program automatically when needed (after ***movd***, in-line assemble, assignment, load coff symbols and copy to code memory).

It is possible to apply an optional address range specifying the range in which the Debugger is requested to perform the disassembly operation.

Examples:

disassemble
disassemble [C:1100,25]

Related Commands:

assemble , Assignment Commands, copy

17.6.46 disconnect external debugger

disconnect external debugger DebuggerUniqueName

disconnect all external debugger

The **disconnect external debugger** command is used to terminate a connection to another CEVA-Toolbox Debugger.

When disconnecting the active connection, the last connected Debugger becomes the active connection.

When using the 'all' option, all connections of CEVA-Toolbox Debuggers are terminated.

Example:

disconnect all external debugger

Related Commands:

connect external debugger, run external cli, transfer

17.6.47 disconnect port

disconnect [port] PortAddress [input/output]
disconnect [port] PortAddress [input/output] file
disconnect [port] PortAddress [input/output] outwin
disconnect [port] PortAddress [input/output] user
disconnect user port all
disconnect port all

The **disconnect** port command is used to break the Debugger's file, Output Window or DLL function connections to the specified I/O-mapped memory. If file, outwin or user are used in the command, only this connection will be broken, otherwise all previously-made connections to this address will be broken.

For more details on I/O file, Output Window or DLL Function Connections, see the connect port command.

Note:

1. This command is valid only in Simulation Mode.

2. When no input nor output is used, both will be disconnected.

Examples:

1. Disconnects input and output file, Output Window or DLL function connections to memory-mapped I/O port at D:3000:

disconnect port 0x3000

2. Disconnects output file connection to memory-mapped I/O port at D:3000:

disconnect port 0x3000 output file

3. Disconnects user connection to memory-mapped I/O port at D:3000:

disconnect port 0x3000 input user

4. Disconnects input file, Output Window or DLL function connection to port at 'Port.Control':

disconnect port Port.Control input

5. Disconnects all input and output file, Output Window or DLL function connections to all memory-mapped I/O ports:

disconnect port all

6. Disconnects all memory-mapped I/O that is connected to the UserIO DLL through the connect user port all:

disconnect user port all

Related Commands:

disable port , enable port, map , rewind , connect port

17.6.48 disconnect register

disconnect [register] extExtRegId [input/output]
disconnect [register] extExtRegId [input/output] file
disconnect [register] extExtRegId [input/output] outwin
disconnect [register] extExtRegId [input/output] user
disconnect register all

Note: This CLI is **irrelevant** for the CEVA-X and CEVA-XC cores.

The **disconnect register** command is used to break the Debugger's file, Output Window or DLL function connections to the specified I/O-mapped external register. If file, outwin or user are used in the command, only this connection will be broken. Otherwise, all previously-made connections to this register will be broken.

For more details on I/O file, Output Window or DLL function connections, see the connect register command.

Note:

1. This command is valid only in Simulation Mode.

2. When neither input nor output is used, both will be disconnected.

Examples:

1. Disconnects input and output file, Output Window or DLL function connections to ext2:

disconnect register ext2

2. Disconnects output file connection to ext2:

disconnect register ext2 output file

3. Disconnects output window connection to ext2:

disconnect register ext2 output outwin

4. Disconnects input user connection to ext2:

disconnect register ext2 input user

5. Disconnects input and output file, Output Window or DLL function connections to all I/O-mapped external registers:

disconnect register all

Related Commands:

connect register, disable register, enable register, rewind register

17.6.49 disconnect user

disconnect [user] iu0/iu1

The disconnect user command is used to disconnect the user input pins (**iu0, iu1**) from the UserIO DLL.

For more information on the UserIO DLL Functions, refer to The UserIO DLL Functions Chapter.

For more details on memory-mapped I/O connections to DLL functions, see the connect port and disconnect port commands.

For more details on external register-mapped I/O connections to DLL functions, see the connect register and disconnect register commands.

Note:

This command is valid only in Simulation Mode.

Examples:

disconnect iu1

Related Commands:

connect user, connect port, connect register

17.6.50 dos

dos [DosCommand]

The **dos** command allows synchronous DOS commands to be entered from the Debugger, either in batch mode or interactive mode.

Notes:

1. This command cannot contain comments.

2. This command is only valid in the PC/Windows version.

Example:

Enters DOS in interactive mode and returns to the Debugger when the user enters 'exit' in the DOS box:

dos

17.6.51 emu config devchip

emu config combo/devchip

The **emu config combo/devchip** command is used to enable/disable the automatic BIU settings and checks that are performed by the Debugger according to the specifications in the **map memory** and **define memory** commands.

When **emu config combo** is issued (default), the Debugger lets the user application set the BIU Registers (disabling setting by the Debugger).

When the **emu config devchip** is issued, the Debugger automatically sets and checks BIU Register values according to the specifications in the **map memory** and **define memory** commands. In this case, mapping attributes that will be set in the BIU registers are:

- Memory and external-registers onchip/offchip configuration
- Memory and external-registers number of off-chip wait-states.

Notes:

1. This command is valid only in Emulation Mode.

2. emu set combo is the default setup.

3. emu set devchip should be applied (following **map memory** and **define memory**) whenever the MMIO address in the BIU is modified (using the **emu set internal register**, internal register 17 CLI). This should be done in order to avoid the Debugger from restoring the MMIO start address to its original value when following a reset sequence.

Examples:

```
emu config devchip
emu config combo
```

Related Commands:

define memory, map, emu set internal register, config simul

17.6.52 emu config reset wait

emu config reset wait NumOfMilliseconds

The **emu config reset wait** command is used to configure the delay between resets initiated by the Debugger. When the Debugger establishes the connection with the hardware, it performs a first reset and a second reset (with DBG and BOOT bits on). If the hardware requires a delay between the two resets, it can be configured through this command. By default, there is no special delay between the two resets.

Note:

This command is valid in Emulation Mode only.

Examples:

emu config reset wait 50

Related commands:
start emu, start cevatl

17.6.53 emu debug input

emu debug input PcPortAddress

Following Emulation mode entry, the **emu debug input** command can be used for simple FPGA register reads .

Combined with the emu debug output command, it is possible to read and write mailbox addresses through these FPGA registers.

Note:

This command is valid in Emulation Mode only.

Usage:

1. **JTAG/CDI connection:**
   - To Write a value to a mailbox address use the following CLI sequence:

     emu debug output <basePortAddress> + 0xA, <mailboxAddress>
     ; write the mailbox address to the FPGA address register (base + A)

     emu debug output <basePortAddress> + 0xC, <mailboxValue>
     ; write the requested value to be written into the FPGA data register (base + C)

     Example: Write value 0x1234, into mailbox address 0x233, when base-port is 0x300:

     emu debug output 0x30A, 0x233

     emu debug output 0x30C, 0x1234

   - To Read from a mailbox address use the following CLI sequence:

     emu debug output <basePortAddress> + 0xA, <mailboxAddress>
     ; write the mailbox address to the FPGA address register (base + A)
     emu debug input <basePortAddress> + 0xC
     ; read the requested value from the FPGA data register (base + C)

Example: Read the value from mailbox address 0x233, when base-port is 0x300:

emu debug output 0x30A, 0x233

emu debug input 0x30C

2. Parallel Port Connection (assuming port address is 0x378):

- To Write into a mailbox address use the following CLI sequence:

emu debug output 0x80, <mailbox-address>

; write the address of the of the mailbox to be used into the FPGA mailbox address register (= 0x80)

emu debug output 0x81, <value>

; write the value to be written into the mailbox, to the FPGA mailbox-data register (=0x80)

Example: Write value 0x1234, into mailbox address 0x233, when base-port is 0x378:

emu debug output 0x80, 0x23

emu debug output 0x81, 0x1234

- To Read from a mailbox address use the following CLI sequence:

emu debug output 0x80, <mailbox-address>

; write the address of the of the mailbox to be used into the FPGA mailbox address register (= 0x80)

emu debug input 0x81

; read the requested value from the FPGA mailbox-data register (=0x80)

Example: Read the value from mailbox address 0x233, when base-port is 0x378:

emu debug output 0x80, 0x233

emu debug input 0x81

Related commands:

start emu, emu debug output

17.6.54 emu debug output

emu debug output PcPortAddress Value

Following Emulation mode entry, the **emu debug output** command can be used for simple FPGA register writes.

Combined with the emu debug input command, it is possible to read and write mailbox addresses through these FPGA registers.

Note:

This command is valid in Emulation Mode only.

Usage:

1. JTAG/CDI connection:

- To Write a value to a mailbox address use the following CLI sequence:

  emu debug output <basePortAddress> + 0xA, <mailboxAddress>

  ; write the mailbox address to the FPGA address register (base + A)

  emu debug output <basePortAddress> + 0xC, <mailboxValue>

  ; write the requested value to be written into the FPGA data register (base + C)

  Example: Write value 0x1234, into mailbox address 0x233, when base-port is 0x300:

  emu debug output 0x30A, 0x233

  emu debug output 0x30C, 0x1234

- To Read from a mailbox address use the following CLI sequence:

  emu debug output <basePortAddress> + 0xA, <mailboxAddress>

  ; write the mailbox address to the FPGA address register (base + A)

  emu debug input <basePortAddress> + 0xC

  ; read the requested value from the FPGA data register (base + C)

  Example: Read the value from mailbox address 0x233, when base-port is 0x300:

  emu debug output 0x30A, 0x233

  emu debug input 0x30C

2. Parallel Port Connection (assuming port address is 0x378):

- To Write into a mailbox address use the following CLI sequence:

emu debug output 0x80, <mailbox-address>

; write the address of the of the mailbox to be used into the FPGA mailbox address register (= 0x80)

emu *debug* output 0x81, <value>

; write the value to be written into the mailbox, to the FPGA mailbox-data register (=0x80)

Example: Write value 0x1234, into mailbox address 0x233, when base-port is 0x378:

emu debug output 0x80, 0x233

emu debug output 0x81, 0x1234

- To Read from a mailbox address use the following CLI sequence:

emu debug output 0x80, <mailbox-address>

; write the address of the of the mailbox to be used into the FPGA mailbox address register (= 0x80)

emu debug input 0x81

; read the requested value from the FPGA mailbox-data register (=0x80)

Example: Read the value from mailbox address 0x233, when base-port is 0x378:

emu debug output 0x80, 0x233

emu debug input 0x81

Related commands:

start emu, emu debug input

17.6.55 emu disable jbox command

emu disable jbox command commanNumber

Relevant for CEVA-X16xx, CEVA-TeakLite-III, CEVA-TeakLite-4 and CEVA-XC debuggers.

In order to improve performance on JBOX2, the Debugger sends “JBOX commands” during the Emulation session. Unlike JBOX1 where the debugger sends only naïve scan-chains to the JBOX, The JBOX2 commands triggers JBOX2 to send several scan chains to the core and return the relevant data to the debugger.

The user can disable/enable sending JBOX2 command by using the proper CLI.

JBOX2 Commands

Command name	Description	CEVA-X16xx ID	CEVA-XC ID	CEVA-TeakLite-III / CEVA-TeakLite-4 ID
Read all registers	Reads all registers	0x15	0x15	0x12
Read data memory	Reads a block of data memory	0x14	0x14	0x14
Read all VU registers	Reads Vector-Unit registers	--	0x16	--

Examples:

emu disable jbox command 0x15

Related Commands:

start emu, emu enable jbox command

17.6.56 emu debug set message

emu debug set message level [levelNumber] [, [file] fileName]

The emu debug set message command enables writing various debugging messages (in Emulation mode) into the Log Window or into a file.

This command can help debug problems in Emulation and allows visibility of Debugger low level communication to the driver and the development board

levelNumber - The debug message level number:

0 - Default - No debug messages.

1 - Print the main Debugger function calls messages **only.** The main functions are:

resetOcemRegisters, resetDevKit, abort, startEmulation, stopEmulation, performSingleStep, writeCode, writeTargetData, setCurrentCodeSegmentNumber, setCurrentDataSegmentNumber and run (can be customized by the user).

2 - Print all Debugger functions direct calls messages.

These are High level functions, that is, interface functions of the UserHWIF DLL (defined in the UserHWIF.CPP file).

3 - Print communication debug messages **only**.

These are Low level function calls, of functions that communicate with the PC device driver (.vxd/.sys).

4 - Print all Debugger & Communication debug messages (all message levels).

5 – Print Scan-Chains information only (relevant for CEVA-X and CEVA-XC only).

fileName - An optional file to be used as an output for the debug messages. If no file is specified, the debug messages are displayed in the Log **Window**.

Examples:

emu debug set message level 1 ; main level messages into the Log window

emu debug set message level 4, myMsgLog.txt ; all messages into a file.

17.6.57 emu enable jbox command

emu enable jbox command commanNumber

Relevant for CEVA-X16xx, CEVA-TeakLite-III, CEVA-TeakLite-4 & CEVA-XC debuggers.

In order to improve performance on JBOX2, the Debugger sends “JBOX commands” during the Emulation session. Unlike JBOX1 where the debugger sends only naïve scan-chains to the JBOX, The JBOX2 commands triggers t JBOX2 to send several scan chains to the core and return the relevant data to the debugger.

The user can disable/enable sending JBOX2 command by using the proper CLI.

JBOX2 Commands:

Command name	Description	CEVA-X16xx ID	CEVA-XC ID	CEVA-TeakLite-III/ CEVA-TeakLite-4 ID
Read all registers	Reads all registers	0x15	0x15	0x12
Read data memory	Reads a block of data memory	0x14	0x14	0x14
Read all VU registers	Reads Vector-Unit registers	--	0x16	--

Examples:

emu disable jbox command 0x15

Related Commands:

start emu, emu disable jbox command

17.6.58 emu get internal register

emu get internal register InternalRegId

The **emu get internal register** command is used to get Debugger's configuration concerning various emulation-related issues. For more details, see the emu set internal register CLI command.

Note:

This command is valid only in Emulation Mode.

Examples:

emu get internal register 25

Related Commands:
start emu, stop emu

17.6.59 emu get jbox register

emu get jbox register [offset]

Log This CLI is relevant only when using J-Box2.

This CLI must be entered after emulation session started.

Related Commands:

start emu, stop emu, emu set jbox register, emu set communication type

17.6.60 emu read code

emu read code CodeAdressRange [errorinrange]

The **emu read code** command allows code to be read from the board in Emulation Mode. This command instructs the Debugger to update its Code Memory Window, with regard to a certain code memory range. This command is useful when some of the DSP Core code memory is loaded from an external source.

errorinrange - Reports decoding errors only if they occur in the specified address range.

Whenever this command is issued, the Debugger reads the (non-I/O) mapped DSP Core's data memory (if the Debugger is in Update Mode, otherwise it reads only the (non-I/O) mapped data memory locations that are displayed in the data windows) and updates its data window as well.

As a result disassembly of read code will be shown in code memory window.

Note:

The **read code** command is valid only in Emulation Mode.

Examples:

1. Read the code between addresses 0x0000-0xFFFF:

emu read code [C:0000,C:FFFF]

2. Read 0xFF code words starting from address 0x0000:

emu read code [C:0000,0xFF]

Related Commands:

emu set mode

17.6.61 emu set communication type

emu set communication type

ethernet / Parallel

The **emu set communication type** CLI determines the connection method between the PC and the J-Box. The debugger can communicate with the J-box using several communication types.
When using J-Box1, the main connection type is while using Parallel Port. In J-Box2, the connection type is Ethernet.

This CLI actually chooses the J-Box type: when using parallel port, it means that J-Box1 is in use. On the other hand, if the user defined the communication type as Ethernet, it means he uses J-Box2.

Default communication type is Parallel Port.

Notice:

1. This CLI must be entered before emulation session started.

Examples:

emu set communication type ethernet.

Related Commands:

start emu, stop emu

17.6.62 emu set daisy chain

emu set daisy chain [chain's length] core [selected core]

Using the JTAG interface, several cores can be chained together. It means that the JTAG signal is passed through all cores, when the TDO pin of each device is connected to TDI of next device. By accurate padding of the JTAG signal, both IR ("instruction register") and DR ("data register") signals, only the selected core in the chain will get the full signal.

This CLI enables using the daisy chain mechanism while using the debugger. User should define the number of cores he has in the chain, and the number of the selected core.

The use of this CLI must come before Emulation has started. After the **start emu** CLI will be entered, several steps will be made:

1. All cores will be put in by-pass.
2. Proper padding will be added by the J-Box to all IR and DR signals in the current session.
3. The debugger will be connected only to the selected core in the chain.

Notes:

1. This CLI is relevant only when using J-Box2.
2. This CLI is supported only for CEVA-X/CEVA-TeakLite-III/ CEVA-TeakLite-4 cores.
3. This CLI must be entered before emulation session started.
4. This CLI should be entered every time before the Emulation session. After the Emulation session was over (by using **stop emu**, **new session,** closing the debugger etc.) this CLI should be entered again if the user wants to use the daisy chain mechanism.

Examples:

Considering user has a Daisy Chain with 3 cores while CEVA-X/CEVA-TeakLite-III/ CEVA-TeakLite-4 is the second core. Use the CLI:

emu set daisy chain 3 core 1

Related Commands:

start emu, stop emu, emu set daisy chain ir length

17.6.63 emu set daisy chain ir length

emu set daisy chain core [selected core] ir length [length]

Using the JTAG interface, several cores can be chained together. It means that the JTAG signal is passed through all cores, when the TDO pin of each device is connected to TDI of next device. By accurate padding of the JTAG signal, both IR ("instruction register") and DR ("data register") signals, only the selected core in the chain will get the full signal.

The padding of the IR must be with the specific IR length. Each core has different IR length. In order to make the Daisy Chain mechanism work properly, the user must update the correct IR length of every core in the chain. Default length is 32 bit (CEVA-X/CEVA-TeakLite-III/ CEVA-TeakLite-4 IR length). This CLI enables to define the IR length (in bits) of a selected core in the Daisy Chain.

Notice:

1. This CLI is relevant only when using J-Box2.
2. This CLI is supported only for CEVA-X/CEVA-TeakLite-III/ CEVA-TeakLite-4 cores.
3. This CLI must be entered before emulation session started.

Examples:

Considering user has a Daisy Chain with 2 cores while CEVA-X/CEVA-TeakLite-III/ CEVA-TeakLite-4 is the second core, and the first core's IR length is 10 bit. Use the below CLI sequence:

emu set communication type Ethernet ; Daisy Chain mechanism works only with J-Box2.

emu set daisy chain 2 core 2 ; debugger should connect to the 2nd core in the chain.

emu set daisy chain core 1 ir length 10 ; Update the IR length in bits of the first core in the chain.

start emu ; after all configurations are done, the emulation session can start.

Related Commands:

start emu, stop emu, emu set daisy chain, emu set communication type

17.6.64 emu set data access

emu set data access DataAdressRange Alignment bit

Relevant for CEVA-X16xx, CEVA-TeakLite-III, CEVA-TeakLite-4 and CEVA-XC debuggers.

Legal alignments: CEVA-X16xx, CEVA-XC: 8, 16, 32 bit.
CEVA-TeakLite-III, CEVA-TeakLite-4: 16, 32 bit

On certain systems, there's a requirement to access certain data areas with specific alignment. As part of the emulation concept, the debugger injects certain instructions in order to access the data memory. If there's a restriction considering the access width for a specific area, only the proper aligned instructions should be used.

Using the "set data access" CLI, the user can define a block in the data memory, where all accesses will be aligned to the number of bits the user determined.

Default access width will be the largest option allowed (32 bit for CEVA-X16xx, CEVA-TeakLite-III, CEVA-TeakLite-4 & CEVA-XC)

Note:

1. The emu read code command is valid only in Emulation Mode.
2. This CLI must be used only after Emulation started.
3. It's recommended to use this CLI before mapping/loading COFF file
4. Overlapping is prohibited.
5. Block's length should be aligned with the data width access.

Examples:

Define a block from 0x2000 to 0x2008 where all accesses will be 16 bit align:

emu set data access [d:2000, 8] 16 bit

Define a block from 0x1000 to 0x3000 where all accesses will be 8 bit align (CEVA-X16xx /CEVA-XC only):

emu set data access [d:1000, d:2FFF] 8 bit

Related Commands:

start emu, stop emu, map

17.6.65 emu set data address mask

emu set data address mask AddressMaskValue

The **emu set data address mask** command is used to extend the breakpoint addresses. The OCEM (On-Chip Emulation Module) has a special mask register for extended data address breakpoints. Instead of breaking on a particular data address, it is possible to break on a range of addresses. The following statement is used to extend the breakpoint addresses:

(DataAddressBreakpointRegister ^ DataAddressBus) & mask) = 0

By clearing bits in the mask (selecting the appropriate mask value), one can disable/ignore bits in the data address breakpoint comparison.

Note:

The **emu set data address mask** command is valid only in Emulation Mode.

Examples:

emu set data address mask 0xFF00

Related Commands:

strat emu, set break, cancel break, disable break, enable break

17.6.66 emu set internal register

emu set internal register InternalRegId , InternalRegValue

The **emu set internal register** command is used to configure the Debugger in regard of some emulation-related issues. In general, there is no need to apply this command when using the original CEVA made Emulation Board.

A knowledge of the related DSP Development Board is needed in order to change these internal Debugger settings.

The table below explains each option:

Table 17-1: Emulation Setting Options

Internal RegId	InternalRegValue	Purpose
0	Code address	Set a new **Monitor program start address** in the code memory.*
1	Data address	Set a new **Mailbox start address** in the data Memory.*
2	Data address	Set a new **BIU registers start address** in the data memory.
3	PC Port base address	Set the PC I/O **port base address** (set automatically by the Debugger).*
4	**CEVA-X Only** Scan-chain integer identity	Set the scan-chain identity number (e.g. –0x50 for OCEM control scan-chain). See CEVA-X spec vol 1.
5	**CEVA-X Only** Value	Set the scan-chain value to the identity specified for 4 (above).

Internal RegId	InternalRegValue	Purpose
6	0 - The default behavior, reset after ILL detection. 1 - Disable reset after ILL detection. 2 - The Debugger will pop-up a dialog box after ILL detection to acquire user's decision.	Configures the **reset method** of the Debugger. For example, by default, after ILL detection, the Debugger will perform a reset. ILL is an event detected by the OCEM and points to an attempt to access the Mailbox area not through the TRAP routine. Disabling reset in such an event may help to debug the cause of the ILL setting. The command affects the reset execution for all the following events: ILL bit, ERR bit, Wrong PC, Hardware error and Div. KIT/PC error.
13	Positive integer value	**CEVA-X only Disable automatic development chip identification** and version checking. **0** - Automatic identification reading from the dev. chip (default value). **1** - Disable automatic identification. Take version of the dev. chip according to internal register set by the user ('emu set internal register 19').*
17	0 – **No** synchronization on simultaneous hardware accesses 1 – Synchronize simultaneous hardware accesses using **hardware** mechanism 2 – Synchronize simultaneous hardware accesses using **software** mechanism	When other applications need to access the target hardware simultaneously with the Debugger, accesses to the hardware must be synchronized for proper usage. This internal register sets the **synchronization method** of simultaneous hardware accesses. **hardware** synchronization means the FPGA is responsible for the synchronization while **software** synchronization means a MUTEX is used for synchronizing between simultaneous accesses. **Default** is 0 (**No** synchronization). **Note** that no synchronization is preferred since it accelerates the Debugger run-time in emulation mode.
19	Integer value	This range of emulation internal registers can be used by the user for **general purposes**.
33	---	Reserved (for Debugger internal use)
34	Integer value	Mailbox size – used internally

Internal RegId	InternalRegValue	Purpose
43	CEVA-X, CEVA-XC only File-I/O operation indication 0 Disable File-I/O operations 1 Enable File-I/O operations	Indication whether to enable File-I/o operations in case of working in multi core systems, in such cases only one debugger can perform File-I/O operations.

Note:

1. The above commands are valid in Emulation mode only.

2. * Commands that are relevant for the Emulation start (like Monitor address setting) and are marked with '*****' must be issued **BEFORE** the **start emu** command in order to take effect.

Examples:

1. Set the Monitor program start address to 0xF000:

emu set internal register 0, 0xF000

2. Set the Mailbox start address to 0xF800:

emu set internal register 1, 0xF800

3. Define the version of the development chip that Debugger will work with:

emu set internal register 13, 0x300 ; work with rev C

emu set internal register 13, 0x200 ; work with rev B

4. Set new OCEM address to be 0x4400:

emu set internal register 16, 0x4400

5. Set the OCEM data address breakpoint #1 low address value to 0x1000 (CEVA-X):

emu set internal register 4, 0x30 ; set scan-chain identity to OCM_DADD_LOW1

emu set internal register 5, 0x1000 ; set OCM_DADD_LOW1 to 0x1000

Related Commands:

start cevatl, start emu, emu-get_internal_register

17.6.67 emu set ip address

emu set ip address [ip address string]

In Emulation session, when Ethernet connection is chosen (J-Box2), the Debugger tries to initiate TCP connection to the J-Box using default IP address and port number.

This CLI updates the IP address the debugger uses in the Emulation session.

Important – this CLI doesn't configures the IP address of the J-Box itself only the debugger's inner parameter.

Notice:

1. This CLI is relevant only when using Ethernet connection (J-Box2)
2. This CLI must be used before Emulation session started.

Examples:

emu set ip address 192.68.52.55

Related Commands:

start emu, stop emu, emu set communication type, emu set tcp port

17.6.68 emu set jbox register

emu set jbox register [offset] [value]

3. This CLI is relevant only when using J-Box2.
4. This CLI must be entered after emulation session started.

Related Commands:

start emu, stop emu, emu get jbox registeremu set communication type

17.6.69 emu set mode

emu set mode update/noupdate

The **emu set mode** command is used to switch the Emulator from the **update** mode to the **noupdate** mode and vice-versa. In the **update** mode, all (non-I/O) mapped data memory locations are physically read every time the control returns to the Debugger. This happens after each **step**, **next** CLIs or error, or whenever a breakpoint of any type has been reached. When in **noupdate** (default) mode, ONLY the (non-I/O) mapped data memory locations displayed in the data windows will be physically read every time the control returns to the Debugger.

Notes:

1. The **emu set mode** command is valid only in Emulation Mode.

2. The default setup in no-update.

3. If fast response time is needed, the (default) no-update is recommended. Note that in this mode, if the user scrolls the data window when the user's application is running (i.e. during go), the displayed data might not be updated. When the application is not running, there is no difference between the update and no-update modes (except for the response time).

Examples:

emu set mode update

Related Commands:

start cevatl, start emu

17.6.70 emu set tcp port

emu set tcp port [port number]

In Emulation session, when Ethernet connection was chosen (J-Box2), the Debugger tries to initiate TCP connection to the J-Box using default IP address and port number.

This CLI updates the TCP port number the debugger uses in the Emulation session.

Important – this CLI doesn't configures the port number of the J-Box itself only the debugger's inner parameter.

Notice:

1. This CLI is relevant only when using Ethernet connection (J-Box2)
2. This CLI must be used before Emulation session started.

Examples:

emu set tcp port 1234

Related Commands:

start emu, stop emu, emu set communication type, emu set ip address

17.6.71 enable asm source debug

This command sets the behavior mode of the source debug commands (**step source**, **next source**, **step out source**). When using this command, which is the default option, debug information is taken from all source files, Assembly and C/C++.

Example:

enable asm source deubg

Related Commands:

evaluate , disable asm source debug

17.6.72 enable branch to self

This CLI is relevant for the CEVA-X, CEVA-XC only.

This command set the behavior of branch to self to enable. In enable mode, when reaching a branch to self, no message pops stating that we are in branch to self and the debugger keeps on running.

The default behavior of branch to self is disabled.

This behavior control allows the user to write an application that reaches at certain points with branch to self and enters an endless loop.

Example:

enable branch to self

Related Commands:
disable branch to self

17.6.73 enable break

enable break BreakpointAddress
enable break all

The **enable break** command is used to enable the breakpoint at the specified address, or enables all breakpoints. Breakpoints may be disabled using the disable break command. When a breakpoint is initially set, it is automatically enabled. For more details on setting/creating address breakpoints, see the explanation of the command set break.

Examples:

enable break MySeg.MyOffset+2
enable break C:1234
enable break all

Related Commands:

cancel break, disable break, set break

17.6.74 enable cfileio debug

enable cfileio debug [file fileName]

The **enable cfileio debug** command is used to disable the C file I/O debug messages mode. When specifying a file name all debug messages will be written into the specified file. Otherwise the messages will be redirected into the Log Window.

The default debug mode is disabled.

Examples:

enable cfileio debug ; enables C file I/O debug mde into the Log Window
enable cfileio debug file fileio ; enables C file I/O debug mde into the Log Window

Related Commands:
config cfileio, disable cfileio debug, evaluate

17.6.75 enable exception

enable exception internal read unmap/
external read unmap/
internal read uninit/
external read uninit/
internal write unmap/
external write unmap/
memory/
modulo/
stack/
xyram/
restriction/
bubble/
unmap/
all

The **enable exception** command is used to enable the specified exceptions. By default, all exceptions are enabled. Note that enable exception bubble is relevant only for Teaklite-III, and enabling the restriction exception enables the bubble exception as well. Disabling exceptions has the effect of masking certain error conditions. For more details on exceptions, see the **disable exception** command.

Examples:

enable exception modulo
enable exception stack
enable exception all

Notes:

1. The **debugger new session** CLI initializes messages and exceptions status with default values.
2. The **disable exception all** CLI doesn't affect the **unmap** exception due to compatibility reasons.

Related Commands:

disable exception, disable load coff reset, enable message

17.6.76 enable interrupt

enable interrupt nmi/int0/int1/int2/vint/all

The **enable interrupt** command is used to enable the simulation of the specified interrupt(s). The interrupt(s) can be disabled via the command disable interrupt. When interrupts are generated/created, they are automatically enabled. For more details on interrupts, see the command generate interrupt.

Note:

The **enable interrupt** command is valid in Simulation Mode only.

Examples:

enable interrupt nmi
enable interrupt all

Related Commands:

cancel interrupt, disable interrupt, generate interrupt

17.6.77 enable load coff reset

enable load coff reset

The **enable load coff reset** command is used to enable the core reset during loading of a COFF file. By default the core reset in the load coff instruction is enabled (reset is performed).

Examples:

enable load coff reset

Related Commands:

disable load coff reset, load coff, evaluate

17.6.78 enable message

enable message assert/
debug/
error/
fatal/
info /
warning/
dialog/
all

The **enable message** command causes messages of the specified severity to be displayed in a message box that the user must acknowledge before the system will process the next command. Messages can be disabled via the disable load coff reset command. By default, all levels of message severity are enabled. For more details on enabling/disabling messages, see the disable load coff reset and disable exception commands.

Examples:

enable message info
enable message all

Related Commands:

disable message , disable exception , disable load coff reset

17.6.79 enable mss_reg counter

The **enable mss_reg counter** command is used for enabling the counter of the mss_reg.

This command set the relevant bit in the mss pause register.

Notes:

mss_reg can only be of the profiler counter regs in the mss

Examples:

enable prof_nop_inst counter

Related Commands:

reset mss_reg counter, disable mss_reg counter

17.6.80 enable overwritten instruction nop padding

enable overwritten instruction [nop padding]

When downloading a section to the code memory in run-time, overwriting a pre-loaded code section, and where downloading starts from the second word of an instruction of the overwritten section, decoding of the binary contents of the code memory might be incorrect. This Debugger command was designed to prevent this. When **overwritten instruction nop padding** is enabled, the Debugger will automatically add a nop (on the remains of the overwritten instruction) before the new downloaded section to allow proper decoding.

Enabling **overwritten instruction nop padding** is useful when downloading a new section that starts at the second word of an instruction of the overwritten section while the Debugger is supposed to ignore the remains of the old section. In this case, if this command is not used, the Debugger may wrongly interpret the code.

Note that this is not in any way an application problem.

Examples:

enable overwritten instruction nop padding

Related Commands:

disable overwritten instruction nop padding, load section symbols

17.6.81 enable port

enable port PortAddress [input/output]
enable port PortAddress [input/output] file
enable port PortAddress [input/output] outwin
enable port PortAddress [input/output] user

The **enable port** command enables the connection of a memory-mapped I/O port to a file**,** Output Window or user-defined DLL function. If file, outwin or user are used in the command, only this connection will be enabled. Otherwise, all previously made connections to this address will be enabled. By default, the connection is enabled when it is created (via the command connect port). The connection can be disabled via the disable port command. For more details on I/O port connections, see the command connect port**.**

Notes:

1. The **enable port** command is valid only in Simulation Mode.

2. If no I/O Mode is given, both inputs and outputs are enabled.

Examples:

enable port 0x4F00
enable port PortSeg.Control output
enable port PortSeg.Control output file
enable port PortSeg.Control output outwin
enable port d:f7fe input user

Related Commands:

connect port, disable port, map memory, rewind port

17.6.82 enable printing tcl calls

The **enable printing tcl calls** command is used to enable the printing of TCL calls to the Debugger's CLI commands. By default, TCL call printing is enabled. For more information, see the disable printing tcl calls command.

Note:
The **enable printing tcl calls** command is relevant only in TCL mode.

Examples:

enable printing TCL calls

Related Commands:

start tcl, deb stop tcl, disable printing tcl calls

17.6.83 enable register

enable register ext ExtRegId **[input/output]**
enable register ext ExtRegId **[input/output] file**
enable register ext ExtRegId **[input/output] outwin**
enable register ext ExtRegId **[input/output] user**

Note: This CLI is **irrelevant** for the CEVA-X and CEVA-XC cores.

The **enable register** command enables the connection of an I/O-mapped external register to a file, Output Window or user-defined DLL function. If file, outwin or user are used in the command, only this connection will be enabled. Otherwise, all previously–made connections to this register will be enabled. By default, the connection is enabled when it is created (via the connect register command). The connection can be disabled via the disable register command. For more details on I/O port connections, see the connect register command.

Notes:

1. The **enable register** command is valid only in Simulation Mode.

2. If no I/O-mode is given, both inputs and outputs are enabled.

Examples:

enable register ext0
enable register ext2 input
enable register ext2 output file
enable register ext2 output outwin
enable register ext2 input user

Related Commands:

connect register , disconnect register , disable register, connect port, map memory

17.6.84 enable relative path to script

enable relative path to script

The **enable relative path to script** command configures the Debugger to treat all paths in a script file relative to this script path. This option can be used in order to enable users to run the same script from a few directories without changing the current directory.

Note: The IDE enables this switch by default.

Example:

enable relative path to script ; Enable relative path to script file.

Related Commands:

disable relative path to script, etp read, etp write

17.6.85 enable rtl version

[enable] rtl RtlVersion

The **enable rtl version** command enables the special RTL version restriction checking by the Debugger. When a specific RTL version is enabled, the Debugger will notify whenever the application performs a restricted sequence described by the specific RTL version's bug list. This will help users to detect and bypass problematic code for the specific core RTL version implemented in the target HW.

For enabling, use this CLI command or load a COFF file. Assembler using the Assembler's -rtlX command line switch.

Supported **RTL Version**s are as follow:

CEVA-X1620	– 0.5,1.0,1.1,1.1.5,1.2,1.2.1
CEVA-XC321	– 1.2.0,1.2.1
CEVA-XC323	– 1.1.0,1.1.1,1.2.0,1.2.1,1.3.0
CEVA-XC4210	– 1.2.0
CEVA-TeakLite-III	– 1.0, 1.2.4, 1.3.0, 1.3.2, 1.3.3, 1.3.4, 3.1.0, 3.2.0
CEVA-TeakLite-410	- 0.1, 1.1.0, 2.0.0
CEVA-TeakLite-420	- 0.1, 1.1.0, 2.0.0
CEVA-TeakLite-411	- 0.1, 2.0.0
CEVA-TeakLite-421	- 0.1, 2.0.0**Examples:**

enable rtl 1.2.0

Related Commands:

disable rtl version

17.6.86 etp read

etp read dataAddress clock cycleNumber | pcAddress

This CLI is relevant for the CEVA-X only.

The **etp read** command schedules an External Transfer Port (ETP) read operation to occur from a specified data memory address at a specific clock value or PC value. When cycleNumber or pcAddress has been reached a read from dataAddress will take place using the ETP protocol.

Note:

This CLI command is only relevant for Simulation mode.

Examples:

etp read d:2345 c:3000
etp read d:2345 clock 200

Related Commands:

etp write

17.6.87 etp write

etp write dataAddress dataValue clock cycleNumber | pcAddress

This CLI is relevant for the CEVA-X only.

The **etp write** command schedules an External Transfer Port (ETP) write operation to occur to a specified data memory address at a specific clock value or pc value. When the clockcycleNumber or pcAddress has been reached the dataValue will be written to the dataAddress using the ETP protocol.

Note:

This CLI command is only relevant for Simulation mode.

Examples:

etp write d:2345 0x200 c:3000
etp write d:2345 0x200 clock 200

Related Commands:

etp read

17.6.88 evaluate

The **evaluate** command enables the following items to be evaluated:

- System environment variables
- Numeric expressions
- Registers
- Address expressions
- Address range (Array)
- Debugger internal variables

Note:

The user should be aware that applying the evaluate command may result in side effects (e.g. as a result of referencing a readable I/O port).

This command can use the '?' character as a substitute to the 'evaluate' string.

Options:

Evaluated Item	Returned Value
<Address>	Returns the value of the data/code/io address.
<Address, numCells>	Returns a WORD/DWORD value in of the data/code/io address.
<AddressRange>	Returns the values that are stored in the address range (array).
<register>	Returns the value of the register.
<vector name>:<vector unit number> (For CEVA-XC only)	Returns the value of the full vector.
<expression>	Returns the value of the expression. Could be used for simple mathematical expressions or for evaluating labels. See examples below.
env <environmentVariable>	Returns the value of the environment variable if it exists.
rtl	Returns the RTL version of the core.
current code segment	Returns the number of the current code segment.
current data segment	Returns the number of the current data segment.
code segments	Returns the number of code segments.
data segments	Returns the number of data segments.
fastfloat	Returns 'enable' if the fast floating point format is

Evaluated Item	Returned Value
	currently configured, ‘disable’ otherwise.
ieeefloat	Returns ‘enable’ if this format is currently configured, ‘disable’ if not.
asm source debug [mode]	Returns the current source debug mode. i.e. 'enable' or 'disable'.
emu debug message level	Returns the current emulation debug message level (e.g. 3).
emu debug message file	Returns the current output file name of the emulation debug messages (e.g. myOutFile.txt).
last cli	Returns the string of the previously executed CLI into the Command Window.
debugger mode	Returns the mode of the Debugger. i.e. ‘Emulation’ or ‘Simulation’. One of the following will be printed in case of Simulation: ‘Cycle Accurate Simulation’ ‘Cycle Accurate Pipeline Simulation’ ‘Instruction Set Simulation’
cevax type	Returns the CEVA-X type: CEVA-X1620a, CEVA-X1620, CEVA-X1640, CEVA-X1680
Load coff reset [mode]	Returns ‘Enabled’ if reset is enabled at load coff instruction, ‘Disabled’ otherwise.
cfileio debug [mode]	Returns ‘Enabled’ if the current C file I/O mode is on, 'Disabled' otherwise.
relative path to script	Returns ‘Enabled’ if relative path to script is enabled, 'Disabled' otherwise.
test mode (CEVA-X/CEVA-XC only)	Returns the CEVA-X test mode. i.e. ‘Regular' or 'Test'.
mss config (CEVA-X/CEVA-XC only)	Returns "enable"/"disable"/"disable code"/"disable data"
mss code size (CEVA-X/CEVA-XC only)	Returns the size of internal code memory.
mss data size (CEVA-X/CEVA-XC only)	Returns the size of internal data memory.
mss data blocks (CEVA-X/CEVA-XC only)	Returns '2' or '4'
mss data block width (CEVA-X/CEVA-XC only)	Returns '1' or '2'
mss data dma width (CEVA-X/CEVA-XC only)	Returns '1' or '2'
mss data bank width (CEVA-X/CEVA-XC only)	Returns '16' or '32'
mss external code width (CEVA-X only)	Returns '2' or '4'
mss code block size (CEVA-X/CEVA-XC only)	Returns '32' or '64'
mss data block size (CEVA-X/CEVA-XC only)	Returns '16' or '32'
ctag:<coreAddress> (CEVA-X/CEVA-XC only)	Returns miss/ hit
<writeBufferRegister> [value] (CEVA-X/CEVA-XC only)	Returns the value of <writeBufferRegister> as a list of values, space separated.
<writeBufferRegister> target (CEVA-X/CEVA-XC only)	Returns the data address which is the current target of <writeBufferRegister>
profiler [file]	show the current profiling trace file used, return the file name.

Examples:

evaluate D:5 ; ➔ D:5
evaluate D:5 + 3 ; ➔ D:8
evaluate symbol (?symbol) ; ➔ address of symbol
symbol + 5 ; ➔ address of symbol + 5
**D:6* ; ➔ contents of address d:6
**d:6 + 5* ; ➔ (contents of address d:6) + 5
evaluate 2+2 ; ➔ 0x4
?r0 ? 4 : 6 ; ➔ (r0 != 0) ? 4 : 6
r1 > r2 ; ➔ (r1 > r2) ? 1 : 0
c ^ 1 ; ➔ toggle value of carry flag
MySeg.MyOffset ; ➔ address of MySeg.MyOffset
**MySeg.MyOffset* ; ➔ contents of address pointed by MySeg.MyOffset
offset MySeg.MyOffset ; ➔ MySeg.MyOffset - MySeg.0
?current code segment ; ➔ 2
?current data segment ; ➔ 0
**m:200* ; ➔ contents of Mailbox offset address 0x200
evaluate env %ProjDir% ; ➔ value of environment variable "ProjDir"
?env %TKLTOOLS% ; ➔ value of environment variable "TKLTOOLS"
?segments ; ➔ **2 Code segments, 4 Data segments**
?code segments ; ➔ **2 Code segments**
?rtl ; ➔ 3.0.1
?fastfloat ; ➔ Enable
?ieefloat ; ➔ Disable
?asm source debug ; ➔ enable
?emu debug message level ; ➔ 3
?emu debug message file ; ➔ myMsgFile.log
?last cli ; ➔ step
?debugger mode ; ➔ Emulation
?[D0:0000,3] ; ➔ '12dc fe34 65a7'
?[C1:6000,C1:6003] ; ➔ 'e333 1c45 5555 0000'
?[d0:2000,label] ; ➔ print values from D0:2000 until label
? label@C2 ; ➔ address of label that is located in code segment #2
? label.3@D1 ; ➔ address of label+3 that is located in data segment #1
? label.(1+3)@D1 ; ➔ address of label+4 that is located in data segment #1
? cfileio debug mode ; ➔ 'Disabled'
? relative path to script; ➔ 'Disabled'
? test mode ; ➔ 'Regular'
? mss config ; ➔ 'disable code'

? mss code size ; ➔ '0x40000 (256KB)'

? mss data size ; ➔ '0x10000 (64KB)'
?ctag: 0x1000 ; ➔ 'hit/miss'
?wb_a ; ➔ '0x12 0x34 0x56 0x78'
?wb_a target ; ➔ '0x1000'
? profiler [file] ; ➔ prof1.frm

Related Commands:

Assignment Commands, config floating point format, enable asm source debug, enable rtl version, emu debug set message, enable relative path to script, config mss, ctag, define symbolName

17.6.89 exit

The **exit** command terminates the Debugger.

Examples:

exit

Related commands:

dos

17.6.90 fill

fill AddressRange FillValue

The **fill** command is used to fill part of the code or data memory with a specified value.

Examples:

1. Fill the data memory addresses 0x1000-0x2FFF with the value 0x5555:

fill [d:1000,d:2fff] 0x5555

2. Fill 32 words of the data memory starting from address 0x4000 with the value *0b0001100101001001:*

fill [D:4000, 32] 0b0001100101001001

3. Fill the code memory addresses 0x2000-0x20FF (which are mapped as data) with the value 0x1234:

fill [c:1000,0x100] 0x1234

Related commands:

copy, load coff, load coff symbols, Assignment Commands

17.6.91 generate interrupt

1. **Synchronous version:**

   generate [interrupt] nmi/int0/int1/int2 [after delay] [every freq] [for howMany] [during pendingPeriod]

   generate [interrupt] vint/vintc [after delay] [every freq] [for howMany] [during pendingPeriod] vaddress VectoredInterruptAddress

2. **Asynchronous/Event driven version:**

   generate [interrupt] nmi/int0/int1/int2 user

   generate [interrupt] vint user

The **generate interrupt** commands allow interrupts to be simulated while running the Debugger in Simulation Mode. Each of the interrupts nmi, int0, int1, int2 and vint can be independently simulated. ***Synchronous*** interrupts can be simulated with a predefined rate controlled by 4 parameters given in the command. ***Asynchronous/Event driven*** interrupts can be simulated with a user-defined (The UserIO DLL Functions) DLL function UserIO_PostExec() (for CEVA-X, CEVA-XC, CEVA-TeakLite-III and CEVA-TeakLite-4 the function UserIO_PreFetch() is used).

In the ***synchronous*** version of the **generate interrupt** command, the parameters of the Interrupt Sequencer are the following:

- Initial delay
- Frequency
- Loop counter
- Duration of each interrupt request

The simulated clock tick is equal to one DSP Core instruction cycle unit. Memory wait-states (also measured in instruction cycle units) cause execution times to be lengthened. Interrupts can be canceled, disabled, and enabled.

- **delay -** Start generate after <delay> cycles.
- **freq -** Generate interrupt every <freq> cycles.

- **howMany -** Generate <howMany> interrupts.
- **pendingPeriod -** Hold the request for <pendingPeriod> cycles.

Note: **vintc** is a vectored interrupt with context switching.

In the ***asynchronous/***event ***driven*** version of the 'generate interrupt' command, the simulation of the interrupts is initiated by the (UserIO DLL's) UserIO_PostExec function (for CEVA-X, CEVA-XC ,CEVA-TeakLite-III and CEVA-TeakLite-4 the function UserIO_PreFetch() is used) during certain conditions determined by the user. The function enables the simulation of the interrupts by assigning (during run-time) a value to its input parameter *Arg_W_lpwInSigsMask (according to the parameters that UserIO_PostExec gets), which determines which interrupts out of all will be enabled for use.

The **generate interrupt xxxx user** command will define which of the interrupts will be enabled for generation **through** the UserIO_PostExec() function.
More on interrupts generation can be found in the Simulating Interrupts section.

Notes:

1. The **generate interrupt** command is valid in Simulation Mode only.
2. The interrupt request duration method can be determined by the **interrupt request method** command, and may continue until the end of the duration or until an acknowledge (the duration parameter is ignored in this case).
3. If no "after" option is specified, the interrupt is generated immediately. If no "every" option is specified, the interrupt is generated each cycle. If both "for" and "every" options are not specified, the interrupt is generated only once. If the "for" option (only) is not specified, the interrupts are generated forever. If no "during" option is specified, the interrupt is requested for one cycle only. A maskable interrupt that wants to be simulated will be considered only if the IM0, IM1 and IM2 (in st0 & st2) Interrupt Mask bits will be active to enable the desired interrupt. A maskable interrupt that wants to be simulated will be considered only if the IE (Interrupt Enable) bit (in st0) is active.

Examples:

1. one-shot 'int2' after 15 clocks:

generate interrupt int2 after 15

2. forever 'int1' with period of 46 clocks starting immediately:

generate interrupt int1 every 46

3. 5 times int0 with period 40 clocks after 30 clocks hold each request for 5 cycles:

generate interrupt int0 after 30 every 40 for 5 duration 5

4. enable generating event driver interrupts through the UserIO DLL:

generate interrupt nmi user

Related commands:

cancel interrupt , disable interrupt , enable interrupt, interrupt request method, set input signals

17.6.92 get last cli

The **get last cli** command displays the previously executed CLI in the Log Window.

Alternatively, the **evaluate last cli** CLI can be used.

Examples:

The following CLI sequence will write the string 'step' as the last line in

the Log Window:

step
get last cli
step

Related commands:

etp read, etp write

17.6.93 go

go [ToAddress]

The **go** command starts executing the loaded program from the current pc until the optional specified address is reached, or until one of the following conditions occurs:

- executes at a breakpoint
- executes beyond the end-of-program
- executes at second word of two word instruction
- executes at a DW
- executes at unmapped memory
- ^pause command
- a simulation engine exception (when in Simulation Mode)

The **go** command can also be invoked by pressing the 'F5' key, or by pressing the 'Go' button on the toolbar (no default address in this case).

Note:

The **run** command ignores breakpoints; otherwise, its behavior is identical to the **go** command.

Examples:

go
go 0x345
go MyCodSeg.MyOffset – 1
go C2:1234 ; go until segment 2 address 0x1234 is fetched

Related commands:

reset, continue, pause, run, load coff, next, step, next source, step source, step out source

17.6.94 input

input DataAddress/extExtRegId

The **input** command is the recommended way to evaluate the contents of an I/O port or (for CEVA-TeakLite-III only) external register. Since this command causes the real or simulated I/O port to be read, it may cause side effects. Even if the port or external register is protected (see the protect command), the read operation will be performed.

Examples:

input D:1234
input Port.Control + 3
input ext3

Related Commands:

output , connect port, connect register, disconnect register, disconnect port, disable port, disable register, enable port, enable register, map, protect , protect , rewind, rewind , etp read, etp write

17.6.95 interrupt request method

interrupt request method 0/1

The **interrupt request method** command is used to determine the interrupt request acceptance method (pending). Two methods are available:

0 - Hold the request for the number of cycles defined by the duration parameter.

1 - Hold the request until acknowledge (i.e. until the interrupt has been accepted). In this option the duration parameter is ignored.

Notes:

1. The **interrupt request method** command is valid in Simulation Mode only.3. **interrupt request method 0** is the default.

Examples:

interrupt request method 1

Related commands:

generate interrupt

17.6.96 load coff

load coff CoffFileName [append]

the **load coff** command loads an executable file into code and data memory. The following should be noted:

- All memory is unmapped at initialization.
- If the **load coff** command is issued prior to any **map** command, the Debugger automatically performs default mapping that matches the mapping of the .LNK file as reflected in the loaded COFF file. For more details on automatic memory mapping see the Memory Mapping chapter.
- However, the load will terminate prematurely if there is an attempt to load into unmapped memory, in the case any **map** command has been issued prior to the **load coff** command.
- The COFF file (generated by Compiler/Assembler/Linker) contains information according to the applications' compilation options. The Debugger automatically identifies and configures itself accordingly, thus saving manual Debugger configuration by the user through additional CLI commands. The extra information included in the COFF file and used by the Debugger includes:
  - Fast Floating Point (or IEEE) configuration - see config fastfloat .
  - RTL compatibility version – see enable rtl version.
- Upon load completion, the program is automatically decoded and disassembled and all data segments are initialized.
- COFF format executables can contain a symbol table and are symbolically disassembled, whereas Intel-hex executables are not symbolically disassembled.
- When the **append** option is NOT used:
  - The Simulator / Emulator is reset
  - All breakpoints are cancelled
  - The symbol table is deleted

- All input port connections are rewound
- The interrupt simulation counters are cleared for one-shots and otherwise reset to their maximum values.

Note:

- When the **append** option is used:
  - The Simulator / Emulator is not reset.
  - All previous breakpoints remain active.
  - The previous symbol table remains relevant and is updated with the new symbols from the new COFF. However, in case part or all previous loaded COFF file is overloaded, the symbols which were overloaded by the new COFF file are deleted.
  - All port connections remain unchanged.
  - Interrupt simulation configuration remains unchanged.
  - The title bar is updated with last loaded COFF name.
  - All sources which were in mixed mode are returned to the regular (non-mixed) mode.
  - I/O mode is updated according to the last loaded COFF.
  - Note that up to **5** COFF files can be loaded simultaneously using the **append** option (including the first loaded COFF file).
  - The **append** option can be useful when the application is partitioned into several COFF files, e.g. multi-tasking system, when each task is represented by a different COFF file.
- Besides the **load coff** command, the only other ways to modify the contents of the code memory is with the In-line Assembler via the command **assemble,** or by executing ***movd*** instructions, **copy** or **fill** commands, or by the **Assignment Command**. The data memory may be modified with the **fill** and **copy** command.
- In **Emulation mode**, the Debugger protects against loading to restricted program and/or data memory area. The restricted areas are:
  - BIU registers

- OCEM registers
- TRAP/BI interrupt vector
- Monitor area

The Debugger issues a warning/error message when the application is mapped to these addresses.

When loading a COFF file the Debugger will show a dialog-box with a progress percentage indication.

Examples:

load coff file1.a
load coff %MyProjectDir%\file2.a append

Related commands:
load coff symbols, Assignment Commands, copy, fill, map , rewind , reset, verify

17.6.97 load coff symbols

load coff symbols CoffFileName [append]

The **load coff symbols** command loads an executable file's symbol table (only) into the Debugger. The command enables symbolic debugging of a program that is already loaded (burned) in the HW - that is, only the COFF debugging information is missing.

When the **append** option is used, it is possible to add symbols and debug information of another COFF file in addition to the already loaded COFF file(s) symbols and debug information. When the **append** option is NOT used, all previous loaded COFF file(s) symbols are cleared prior to adding the new COFF symbols. **Note** that the maximal number of concurrent loaded COFF files is **5**.

Examples:

load coff symbols myprog.a
load coff symbols %MyProjDir%\myprog2.a append

Related commands:

load coff, assemble, Assignment Commands, copy, fill , map memory , rewind , reset , verify

17.6.98 load section symbols

load section symbols **SectionName**

The **load section symbols** is used in a debug session, to provide the Debugger with the current downloaded section name, so another source file, symbols and debug information associated with this section will be used instead of the original section information. When overlaying code sections, the Debugger cannot know which source file is associated with the current code memory contents (that may be changed/downloaded by ***movd*** after loading). This is why it always displays the same source file that is related to the originally loaded code section.

The **load section symbols** command will load debug information for the specified section, will change code window labels and display the correct source file when the program counter points to one of the (new) section addresses.

Examples:

load section symbols mySection

Related commands:

load coff ,assemble, enable overwritten instruction nop padding, disable overwritten instruction nop padding

17.6.99 map

map [memory] AddressRange MemoryClass MemoryAttributes
map [register] extExtRegId ExtRegAttributes

MemoryClass – memory class name (alias) as defined previously by the define memory command.

MemoryAttributes – readonly
{ior | iow | iorw}
{internal | external | onchip | offchip | user }
stack{code | data}

ExtRegAttributes – {ior | iow | iorw}
{internal | external | onchip | offchip}
register

The **map** commands map code, data and IO memory (for CEVA-X, CEVA-XC, CEVA-TeakLite-III and CEVA-TeakLite-4 only) memory, memory-mapped I/O ports and external registers. More precisely, they define the attributes of each memory location and external register. One of the attributes is whether or not the location or register is mapped (i.e. used in the application). By default, all memory and all external registers are unmapped.

Unmapped memory and external registers are denoted by asterisks (****) in the Data Memory Window and Registers Window**.** Before a program can be loaded, before the In-Line Assembler can be used, and before a file, Output Window or a user-defined DLL function can be connected to an I/O port or an external register, the memory or register must be mapped appropriately. Usually, memory mapping is done manually through **map** commands prior to loading the COFF file. However, if a **load coff** command is issued before any mapping command, the Debugger automatically maps memories according to the mapping done in the .LNK file as reflected in the loaded COFF file. For more details on automatic memory mapping see the Memory Mapping chapter.

The only attribute that must be explicitly specified is the memory class signified by the class alias. This alias must be defined by the define memory command before the alias can be used

in the **map** command. The recommended way to unmap memory or registers is by the **unmap** command. Correct and accurate mapping is useful for detecting unintended memory accesses in Combo-specific memory and external register configurations.

Memory attributes and external register attributes depend on whether the memory is in the program (code) space or in the data space and should be specified as explained below:

(1) ***Code memory*** has the following attributes:

⇒ **class** (signified by the class alias):
The attributes associated with a memory class are the wait-states and the class id, which is a number from 0 to 15 (Numbers 0, 1, and 2, are reserved). Number 0 signifies that the specified memory section is unmapped.

⇒ **bus type:**
Mapping code memory addresses as **external** (or **user** for the CEVA-X, CEVA-XC, CEVA-TeakLite-III and CEVA-TeakLite-4) will automatically cause the Debugger to call the UserIO_InExtCodeMemory function when reading from an external address, and the UserIO_OutExtCodeMemory function when writing to an external address.

Setting memory **internal** or **external** are additional memory attributes affecting whether or not the memory is respectively associated with the Combo chip. The default setting is **internal**.. Each mapped address in the low block must have a 0 wait state, and each mapped address in the high block must have the same programmable wait state. By default, the low block is assigned the value *internal* when it is not explicitly specified. The high block can only have the external attribute (and it should be explicitly specified since the default is internal).

(2) ***Data memory*** has the following attributes:

⇒ **class** (signified by the class alias):
The attributes associated with a memory class are the wait-states, and the class id, i.e. a number from 0 to 15. Numbers 0, 1, and 2, are reserved. Number 0 signifies that memory is unmapped.

⇒ **bus type:**
Mapping data memory addresses as **external** (or **user** for the CEVA-X, CEVA-XC, CEVA-TeakLite-III and CEVA-TeakLite-4) will automatically cause the Debugger to call the UserIO_InExtDataMemory function when reading from an external address, and the UserIO_OutExtDataMemory function when writing to an external address.

Another memory attribute is whether or not the memory is **internal** or **external** with respect to the combo chip. The default setting is internal.

⇒ **access type** (i.e. RAM/ROM or memory mapped I/O):
By default, all data memory is assumed to be RAM. ROM-type data memory should be assigned the **readonly** attribute. Before a memory-mapped I/O port can be simulated by connecting it to a file, Output Window or a user-defined DLL function, it must be mapped as one of **ior**, **iow** or **iorw**. A memory-mapped I/O port can have protected or unprotected read access. By default, when mapped, all I/O ports are protected. Protected ports and ports without read access are not displayed in the data window, in order not to cause a file read operation (in Simulation Mode) or a state change in Emulation Mode when the data window is normally updated after each execution command (go, step...) or after each window scrolling or screen refresh. For example, a port connected to a UART should be protected, but a port connected to a dip-switch need not be protected. To change protection, use the protect and unprotect commands. The Simulator will catch illegal types of memory access, e.g. an attempt to write to memory with a **readonly** attribute.

Note that all data memory is uninitialized by default. If the data sections were defined with a DW statement, then the data memory becomes initialized. To simulate truly uninitialized memory, the Linker must be instructed to locate the data section with the **noload** attribute. In this case, Debugger loader does not initialize the data memory with the data section contents. Moreover, in this case, when the application reads from the memory before it writes to this memory, an error will be reported. Uninitialized memory is denoted by "**????**" in the data memory window.

⇒ **stack area**

The Software Stack is configurable by the map command. Every time the Stack Pointer increments or decrements as a result of a **push** or **pop** instruction (or any other instruction affecting SP), the simulator will check whether or not the stack pointer is holding an address within the **stack** area. If the stack pointer holds an address value, which is outside the stack area, a "run-time exception" message will be displayed. See also the Message Management section.

(3) ***External registers*** have the following attributes: (Relevant only for CEVA-TeakLite-III)

⇒ **access type** (i.e. I/O mapped or register):

By default, all external registers are unmapped. Before an external register can be simulated by connecting it to a file, Output window or a user-defined DLL function, it must be mapped as one of **ior**, **iow** or **iorw**. Alternatively, it can be mapped as **register**, in which case it will be treated as an ordinary read/write register. An I/O-mapped external register can have protected or unprotected read access. By default, when mapped, all external registers are unprotected. Protected registers and registers without read access do not have their contents displayed in the register window in order not to cause a file read operation (in Simulation Mode) or a state change in Emulation Mode when the register window is normally updated after each execution command (go, step, next...) or after each window scrolling or screen refresh. For example, an external register connected to a UART should be

protected, but one to a dip-switch need not be protected. To change protection, use the protect and unprotect

commands. By default, all accesses to external registers are performed using zero wait-states. If wait-states are needed, use the set extreg wait command. All external registers must use the same number of wait-states.

(4) ***I/O memory*** (relevant for the CEVA-X, CEVA-XC, CEVA-TeakLite-III CEVA-TeakLite-4 only) has the following attributes:

⇒ **class** (signified by the class alias):
The attributes associated with a memory class are the wait-states, and the class id, i.e. a number from 0 to 15. Numbers 0, 1, and 2, are reserved. Number 0 signifies that memory is unmapped.

⇒ **bus type:**

Mapping data memory addresses as **user** automatically causes the Debugger to call the UserIO_InExtIOMemory function when reading from a user address, and the UserIO_OutExtIOMemory function when writing to a user address.

⇒ **access type** (i.e. RAM/ROM or memory mapped I/O):

By default, all data memory is assumed to be RAM. ROM-type data memory should be assigned the **readonly** attribute. Before a memory-mapped I/O port can be simulated by connecting it to a file, Output Window or a user-defined DLL function, it must be mapped as one of **ior**, **iow** or **iorw**. A memory-mapped I/O port can have protected or unprotected read access. By default, when mapped, all I/O ports are protected. Protected ports and ports without read access are not displayed in the data window, in order not to cause a file read operation (in Simulation Mode) or a state change in Emulation Mode when the data window is normally updated after each execution command (go, step...) or after each window scrolling or screen refresh. For example, a port connected to a UART should be protected, but a port connected to a dip-switch need not be protected. To change protection, use the

protect and unprotect commands. The Simulator will catch illegal types of memory access, e.g. an attempt to write to memory with a **readonly** attribute.

Note that all IO memory is uninitialized by default. Uninitialized memory is denoted by "**????**" in the data memory window.

When using a paging/segments application, it is possible to map different address ranges to different segments/pages. The mapping CLI commands – **map memory**, show memory map and unmap memory were extended to enable the user to specify the segment number in which the mapping will take place. Mapping commands that do not specify a segment number will map all defined segments for the requested address range (as done before).

Notes:

- It is a good debugging practice to map *only* memory that is needed for the application, since:
  1. The Simulator can trap references to unmapped memory.
  2. Emulator performance is improved when the memory minimum possible size is mapped.
  3. With each mapping command, the Debugger allocates more memory that increases the memory usage of the Debugger and may slow the machine - make sure to map only the necessary memory area.
- The unmap command unmaps memory that previously was mapped by the **map** command – it does not free it from the Debugger's memory usage. Should the user not care to lose the entire memory contents, he may use the **debugger new session** command in order to free mapped memory.
- It is a good practice to use the **debugger new session** command at the very start of a new script file that contains mapping commands on a previously mapped area. Note that when using the debugger new session command, it will bring the Debugger to the state when it was first invoked, thus, memory that was already mapped will be lost along with its contents.

Examples:

map [C:C000, C:C800] my_mem_class1

; map all defined code segments at this range.

map [D3:8000, D3:A000] my_mem_class2

; map only data segment 3 at this range.

map [D:E200, 1000] my_mem_class3

; map all defined data segments at this range (length 1000 words).

map [C2:7000, 1024] my_mem_class3

; map only code segment 2 at this range (length 1024 words).

map [D:0000,D:07FF] xram onchip

map [D:6000,D:7FFF] Rom readonly

map [D:8000,D:801F] IOport iorw

map [D:8020,D:803F] IOport external ior

map [D:F000,D:FFFF] fast stack

map register ext2 iorw

map register ext1 iow

map ext3 register

map [C:0000,C:00FF] mycclass_incode code

; same as "map program [C:0000,C:00FF] code"

map [C:0000,0x20] mydclass_incode data

; same as "map program [C:0000,0x20] data"

map [U:0000,U:2000] myUnifiedClass

; define memory as unified between data and program address spaces

map [IO:0000,IO:20] ioClass user

; define memory as IO memory connected to the UserIO dll

Related commands:

connect port, connect register, define memory, enable exception, protect, set extreg wait, unmap,unprotect, debugger new session, emu config devchip

17.6.100 map program

map program AddressRange data/code

The **map program** command maps a code memory address range and instructs the Debugger to treat it as either data or code. When code memory is mapped as code, it is disassembled, checked for restrictions and presented in the Code Window as instructions. When code memory is mapped as data, it is not disassembled and is presented as constants in the Code Window.

When one or more ***movd*** instructions are executed in Emulation Mode, the Debugger gets an indication from the OCEM (MVD bit in STATUS1 register). However, no information regarding the number of times the ***movd*** instruction being executed is available, nor is the target address in the code memory. Therefore, the Debugger has to compare all of the mapped code memory to the code memory image it has in order to find out what has been changed by ***movd***.

The Debugger disassembles and checks for restrictions only those areas inside the program memory that contain code. Data areas are presented as constants. The **map program** command enables the user to define a code memory address range as code or data. In order to refer to [c:xxxx,c:yyyy] address range as code, the following CLI should be applied:

map program [c:xxxx,c:yyyy] code

If the user wishes modified code memory areas to be treated as data tables, he should apply :

map program [c:xxxx,c:yyyy] data

Then, if the ***movd*** instruction was writing new values to that memory location, it will not be disassembled, unless the command ***map program [c:x,c:y] code*** is issued prior to applying the ***movd*** instruction.

In order to get "**DW**" sections inside the code memory it should be specified in the COFF file as a data section inside code memory, or the ***map program [c:xxxx,c:yyyy] data*** command should be used.

Now, all the mapped code memory is disassembled and checked for restrictions, whether it is not mapped as data inside the code section.

Carefully check user mappings and unmap any unused code memory or mark it as data inside code memory, using ***map program [c:xxxx,c:yyyy] data*** CLI.

Notes:

- It is a good debugging practice to map ***only*** memory that is needed for the application, since:
  1. The Simulator can trap references to unmapped memory.
  2. Also, Simulator and Emulator **performance** is improved, when the (possible) minimum memory is **mapped**.
  3. With every **mapping** command, the Debugger allocates more memory, which increases the memory usage of the Debugger and may slow the machine - **make sure to map only the necessary memory area**.
- The **unmap** command unmaps memory that was mapped by the map command - it doesn't free it from the Debugger's memory usage. If the user does not care to lose the entire memory contents, he should use the **debugger new session** in order to free mapped memory.
- **It is a good practice to use the debugger new session command at the very start of a new script file that contains mapping commands on previously mapped areas**. Note that when using the **debugger new session** command, it will bring the Debugger to the state when it was first invoked. Thus, the memory that is already mapped will be lost along with its contents.

examples:

map program [C:0000,C:00FF] code
map program [C:0000,0x20] data

Related Commands:

map , debugger new session

17.6.101 next

The **next** command belongs to the Assembly Execution group. The behavior of this command is identical to the **step 1** command when the instruction at the current PC is not from the **call** family, that is, **call**, **calla**, and **callr**. Otherwise, its behavior is identical to the **go pc+2** command for the (two word) **call** instruction, and **go pc+1** for the (one word) **calla** and the **callr** instructions respectively. In other words, after the execution of this command, the PC will point to the next sequential instruction which appears in the next code line after the current instruction.

The **next** command can also be invoked by pressing the **Alt+F10** keys, or by pressing the **Assembly next** button on the toolbar.

Examples:

next

Related commands:

reset, continue, pause, go, run, next, step, next source, step source, step out source

17.6.102 next source

The **next source** command is useful for stepping through the user's original source files (Assembly or C/C++) only when the files are assembled and compiled with debug information (**-g** for Assembly files and **–g/g0/g1/g2/g3** for C/C++ files). The behavior of this command is identical to the **step source** command when the source code statement to be executed is not a call to a function. Otherwise, its behavior is identical to the **go** command when a hidden breakpoint is set in the source code statement immediately following the function call. If this command is given in **Assembly mode**, i.e. the COFF file is compiled without the debug information, it is equivalent to the **go** command.

The **next source** can also be invoked by pressing the **F10** key, or by pressing the **source next** button on the toolbar.

Examples:

next source

Related commands:

step source, step out source, go, load coff, pause , run, step , next, continue

17.6.103 output

output DataAddress , Value
output DataAddress , Value

The **output** command is the only possible way to directly write a value to an I/O port or external register.

Examples:

*output Port.Control , 6*8*
output d:5f00 , 0xffff
output ext3, 0x5a5a

Related commands:
connect port, connect register, disconnect port, disconnect register, disable port, disable register, enable port, enable register ,map, protect, rewind, etp read, etp write

17.6.104 pause

When used, with or without a caret (^), the **pause** command will suspend the execution of the commands **call**, **go**, **run,** and **wait**. Following this command (even without a caret), no further commands will be processed until another caret command is entered. Generally, the safest command that will allow execution to be resumed is **^continue**.

Note:

Execution of a script file can be suspended by the (^)pause command.

Examples:

^pause

Related Commands:

call , continue , go , run , wait

17.6.105 protect

protect [port] PortAddress
protect [register] extExtRegId
protect data memory

The **protect** command changes the read access attribute of a memory-mapped I/O port or for CEVA-TeakLite-III only an I/O-mapped external register. Protected ports or register values will not be displayed in the data memory or register windows, respectively, each time the Debugger is stopped by the user or by a breakpoint. If a protected port or external register is connected to a file, the file will not be read each time the screen has to be refreshed. Unprotected ports and registers will be read for each screen refresh. For more information, see the map command.

The **protect data memory** command protects reserved data memory areas from loading (i.e. BIU, OCEM, MMIO, etc'). When protect data memory is enabled and attempting to overwrite these data memory areas (for example by loading a COFF file), a warning will be issued to inform the user of the problem. By default these data memory areas are protected. Note that there are reserved section names for protecting the MMIO areas so the Debugger will not issue a warning when mapping these areas in the Linker script files. The reserved section name is **__MMIO_SECT**. Note also that the protection of the reserved data memory areas includes protection of sections with the 'noload' Linker attribute.

Note:
It is good debugging practice to leave all input ports and readable external registers in protected state. However, if it is essential to view the updated values each time the simulator stops running the application program, perform unprotect command to the port or external register. Then, if such an unprotected input port or external register is connected to a file or user-defined DLL function, each time the screen has to be refreshed, the file will be read (file pointer advances one line), or the DLL function will be called. If the unprotected input port or external register is ***not*** connected, then each time the screen is refreshed, a dialogue box will pop-up and request a value from the user !

Examples:

protect port d:2b00
protect port IOPorts.Uart1
protect register ext2
protect ext0
protect data memory

Related commands:

connect port, connect register, map , unprotect

17.6.106 refresh

The **refresh** command regenerates all the open windows. Usually the Debugger performs this operation automatically when necessary.

Examples:

refresh

17.6.107 reset

reset [**boot**]

The **reset** command performs the following:

- Resets the CPU (see the specific DSP Core architecture specification).
- The pseudo-register clock is cleared.
- All ports and external register connections are reset.
- If in profiling mode, the profiler database will be reset.
- When in Emulation Mode, the Development Chip is physically reset, and the Monitor Program is restarted.
- Following the above, if the **boot** option is specified, the program counter is initialized with an address other than zero following reset (see the specific DSP Core architecture specification). This is done in order to allow the execution of a ***br*** instruction to a BOOT routine (for automatic downloading of programs into program RAM from a slow external EEPROM).

The **reset** command is invoked also by pressing the **Alt+F5** keys, or by pressing the 'Reset' button on the toolbar.

Note:

When the Emulator is NOT running and the reset button is pressed on the Dev. KIT board, the Debugger loses control over the monitor program. The only way to recover in this case, is by issuing the **reset** command.

Examples:

reset
reset boot

Related commands:

go , run , step , next , step source , next source , step out source, debugger new session

17.6.108 reset mss_reg counter

The **reset mss_reg counter** command is used for resetting the counter of the mss_reg.

This command set the relevant bit in the mss reset register.

Notes:

mss_reg can only be of the profiler counter regs in the mss

Examples:

reset prof_nop_inst counter

Related Commands:

enable mss_reg counter, disable mss_reg counter

17.6.109 rewind

rewind [port] PortAddress
rewind [register] extExtRegId

The **rewind** command causes the file which is connected to the input I/O port at the specified address, or (for CEVA-TeakLite-III only) to the specified (input mapped) external register, to be rewound to the beginning of the file.

For more details on I/O port file connections, see the connect port command.

For more details on connections of files to external registers, see the connect register command.

Note:

The **rewind** command is valid in Simulation mode only.

Examples:

rewind 0xE000
rewind port d:c000
rewind ext3
rewind register ext0

Related commands:

connect port , connect register , disable port , disable register , disconnect port , disconnect register , enable port , enable register , map

17.6.110 run

run [CodeAddress]

The **run** command starts executing the loaded program from the current PC address content until the option specified address is reached or until one of the following conditions occurs:

- execute beyond end-of-program
- execute at a DW
- execute at unmapped memory
- ^pause command
- stop bit set or branch to self (when in Simulation Mode)
- simulation engine exception (when in Simulation Mode)

Note:

The **run** command ignores breakpoints. Otherwise, its behavior is identical to the **go** command.

Examples:

run
run 0x4567
run MyCodSeg.Label + 6

Related commands:

reset , continue , pause, go ,
load coff, next , step , , next source ,step source , step out source

17.6.111 run external cli

run external cli [all] [noblock] CliCommand

The **run external cli** command is used to execute a CLI in another CEVA-Toolbox Debugger. If multiple connections are opened, the CLI will be executed in the current active connected Debugger. If the '**all**' option is set, the CLI is executed in all the open Debugger connections. By default, the Debugger waits until the execution is completed followed by output display of the execution by the caller Debugger. If the '**noblock**' option is used, the caller Debugger is not blocked and the output is not displayed.

Examples:

start cevaxcdbg -ddeConnectName-secDbg_cevaxc4210 disable.dbg

connect external debugger secDbg_cevaxc4210

run external cli "load coff demo.a" ; To load coff at the external debugger

Related Commands:

connect external debugger, disconnect external debugger, transfer

17.6.112 set active external debugger

set active/current external debugger DebuggerUniqueName

The **set active external debugger** command is used to set the Debugger connection (to external Debugger) to active state in which other dedicated commands (to external Debugger) will relate to. Note that several Debuggers can be connected but only one connection can be active at a time.

Related Commands:

connect external debugger, disconnect external debugger, transfer, run external cli

17.6.113 set break

set break CodeBreakAddress [count NumTimesToSkipBeforeBreaking] [hw] [call ScriptFileName]

set break DataBreakAddress/RegisterName/ExtRegName [read/write] [value DataValue] [call ScriptFileName]

set break data [read/write] value DataValue [call ScriptFileName]

set break clock [clockValue] [call ScriptFileName]

The **set break** command sets (creates) a breakpoint at the specified address (program or data), on a data bus transaction or, for **CEVA-TeakLite-III only,** also on an external register transaction.

Counted code breakpoints may be set with a count number. The count number determines the number of times the Debugger skips accesses to the breakpoint code address before it breaks. In Emulation Mode, the count number range may be 0-254 (using 255 is equivalent to using 0), while in Simulation Mode the range may be 0-65535.

When in Emulation Mode, the user can force setting the breakpoint as a ***hardware breakpoint***. This is done by supplying the "hw" switch to the command. **Note** that the "hw" switch is mutual-exclusive to the counted breakpoints, as counted breakpoints are implicitly hardware breakpoints.

A Combined (on address and value) data breakpoint may be set if both a data address **and** a data value are specified in the command. Hence, a breakpoint is triggered only when both the address and the value being read or written match.

On data value match breakpoint may be set by using the **set break data** command. In such a case, the Debugger will break after a data value match is detected on the bus.

The call command can be attached to each of the above breakpoints, so when a breakpoint is reached a script file will be called. This enables the user to implement very complex

breakpoints by checking a complex condition inside the script file and issuing a **go** command if the condition is not met.

Breakpoints can be cancelled (deleted) via the command cancel break. Address breakpoints can be disabled and enabled via the commands disable break and enable break respectively.

For setting breakpoints on source files see the set source break command.

For more information on handling breakpoints, see the

Breakpoint Dialog Box Button at CEVA-Toolbox Debugger User's Guide Vol-I.

Notes:

1. For data address and register breakpoints, the bus transaction triggering the breakpoint can be limited to **read** or **write** operations and, if not explicitly specified, to both **read** and **write** accesses.
2. If the address is a code memory address, then the breakpoint must be on the first word of an instruction in memory that is mapped. If the address is a data memory address, then the breakpoint must be at a mapped data address with appropriate access.
3. When the pc gets to the address of the breakpoint, execution will stop ***BEFORE*** the instruction is executed. In data breakpoint, execution will stop ***AFTER*** the instruction, which read from or wrote to the address.
4. All breakpoints are cleared when a new program is loaded.

Examples:

set break C:ABCD
set break D:E001
set break MySeg.MyOffset+7
set break main
set break data read value 0x5555
set break extreg write
set break d:10fa write value 0x5a5a
set break 0x100 count 50
set break clock 1000 call DoThings.dbg

Related Commands:

cancel break,
cancel source break , disable break, enable break, set source break

17.6.114 set dde connect name

<serviceName>

The **set dde connect name** command sets the DDE server application service name.

Note:

The service names are not case sensitive.
This cli is valid in windows only.

Examples:

set dde connect name cevadebugger

17.6.115 set extreg wait

set extreg wait **NumberOfWaitStates**

Note: This CLI is **irrelevant** for the CEVA-X and CEVA-XC cores

The **set extreg wait** command is used to set the number of wait-states associated with all the external registers. By default, no wait-states are associated with external registers.

Examples:

2 wait-states are added for each external register access:

set extreg wait 2

Related commands:

map [register]

17.6.116 set input signals

set input pinId codeAddress value number [after delay] [during pendingPeriod] [skip skipCounter]

set input pinId clock clockValue value number [after delay] [during pendingPeriod] [skip skipCounter]

This command is relevant for the CEVA-X & CEVA-XC only.

The **set input signals** command is used to configure the core input signals.

The assignment of an input signal to a pending value can be triggered either by the program address (PC) value or by the clock value.

Optional parameters:

- **delay**: Initial delay (in cycles) after the trigger to activate the assignment. The default value is 0, which means no delay.
- **pendingPeriod**: Duration of the pending period. The default value is till the next assignment (indefinitely).
- **skipCounter**: Number of times to skip the trigger before activating the signal. The Default value is 0.

This command is relevant for Simulation mode only.

Examples:

1. ***set input int0 c:10 value 1 after 2 skip 3*** **->** Set int0=1 triggered two cycles after the fourth time the program counter is equal to ten. The value will stay till the next assignment.

2. ***set input int3 clock 300 value 1 during 4*** **->** Set int3=1 triggered when clock is 300. Leave the value pending for four cycles.

Related commands:

generate interrupt, etp read, etp write

17.6.117 set mld enable/disable

set mld enable/disable

For CEVA-XC only (Not includes CEVA-XC4xxx).

Turn on/off the MLD unit capabilities in the simulator – default is disabled – can only be done before load coff – loading a coff will change the configuration according to the info in the coff.

When turning on MLD capabilities the MLD unit registers will be shown in Vector Unit Registers window.

Examples:

set mld enable

Related commands:

set vdiv

17.6.118 set program arguments

set program arguments arg1 [arg2 …]

C/C++ application can get input arguments at startup using the *main(int argc, void *argv[])* syntax. The input arguments can be applied through the project settings, Debug tab at the General category, program arguments text box as well as through the **set program arguments** command.

The last updated input arguments will be kept in the **C/C++ program arguments dialog box** for later sessions (can be enabled or disabled from the dialog box).

When the program input arguments are enabled (in the dialog box), the Debugger will update the input arguments for the DSP application after every execution of the **load coff** or **reset** commands. **Note** that this command enables program input arguments in addition to updating new input arguments.

Note also that the new C/C++ Compiler's ***crt0.o*** (***/libs/crt0.c***) includes built-in support for the input arguments for the DSP application. For more details take a look on the ***crt0.c*** source file (search for ARGV_ARGC at the CEVA-Toolbox CEVA-XC Compiler User's Guide).

Examples:

set program arguments c:\mydir\myinputfile.inp 20 0x1234

Related Commands:

load coff, reset

17.6.119 set source break

set source break fileName lineNumber [count NumTimesToSkipBeforeBreaking] [call ScriptFileName]

The set source break command sets a breakpoint at a specific line number in a specific source file. The command supports counted breakpoints or calling Debugger script file upon receiving the breakpoint.

Counted breakpoints may be set with a count number. The count number determines the number of times the Debugger skips accesses to the breakpoint code address before it breaks. In Emulation Mode, the count number range may be 0-254 (using 255 is equivalent to using 0), while in Simulation Mode the range may be 0-65535.

The call command can be attached to the breakpoints, so when a breakpoint is reached a script file will be called. This enables the user to implement very complex breakpoints by checking a complex condition inside the script file and issuing a **go** command if the condition is not met.

For more information on handling breakpoints, see the set break command or Breakpoint Dialog Box Button at CEVA-Toolbox Debugger User's Guide Vol-I.

Examples:

set source break hello.c 12

Related Commands:

cancel source break, set break

17.6.120 set symbols location

set symbols location incode/indata

The **symbols location** command resolves the ambiguity related to symbols that are defined in sections that are mapped in both code and data memories. When this command is NOT used, the Debugger may pop up a dialog box for the user in order to resolve this ambiguity.

Examples of such ambiguities are:

- r0 = label ; when label is in the code and data memories.
- **add watch** label ; when label is in the code and data memories.

Note: that when dialog boxes pop-up is disabled (disable load coff reset), the Debugger assumes **indata** by default.

Examples:

set symbols location incode
set symbols location indata

Related Commands:

load coff, load coff symbols, load section symbols, disable load coff reset, enable message, cancel symbols location

17.6.121 set user

set user Param1 , Param2

The **set user** command is used to pass application-specific information from the Debugger to the user-defined I/O DLL library function. The first parameter can be viewed as a function id, and the second parameter as an input parameter of that function. Typical examples could be to set timeout values, to configure some I/O mode of operation, or to pass information about version control. When this command is entered by the user, the DLL function **UserIO_SetUser()** is called, which returns a pointer to a user defined string. When the parameters are 2 zeros, the DLL function should return its banner. When the returned pointer is not null, the string is shown in a message box on the Debugger's screen.

For more information, see The UserIO DLL Functions at CEVA-Toolbox Debugger User's Guide Vol-I.

Note:

The **set user** command is valid in Simulation Mode **only**.

Examples:

1. Show the banner of the user-defined DLL:

set user 0 , 0

2. User-defined operation:

set user 1 , 0x5a5a

17.6.122 set vdiv

set vdiv 2/4/8

This CLI is relevant for the CEVA-XC only.

The **set vdiv** command sets number of VDIV units in the simulator. The default value is 2 for all cores but CEVA-XC4xxx that the default value is 8 .

This command can be called only before loading coff file, because the loading of a coff file will change this configuration according to the info in it.

The number of VDIV units affects the maximum number of operations a vdiv/vsqrt/vsqrti instruction can perform (according to the voop field in modv0 register). Trying to perform more operations than the VDIV unit configuration will result in an error in the simulator.

For more information about VDIV unit please refer to the CEVA-XC Architecture Specifications

Example:

set vdiv 4

Related Commands:

set mld

17.6.123 show clock

The **show clock** command displays (in the Log Window) the clock count as displayed in the Registers Window**.** The clock is always decimal, and can be used to measure execution time in Simulation Mode. When Debugger log mode is **on**, the contents of the Log window are saved to a file, and therefore this command is helpful in creating tests whose results can be analyzed off-line.

Examples:

show clock

17.6.124 show connect

show connect port [DataAddress/DataAddressRange] [input/output]

show connect register [extExtRegId] [input/output]

The **show connect** command displays (in the Log Window) selected or all connected I/O ports or external registers. Ports and external registers can be connected to files or to user-defined DLL functions for input and/or output.

When reg/port enabled shows: (+)

When reg/port disconnected shows: (-)

For more information, see the map, connect port and connect register commands.

Examples:

1. Display all input ports in specified data range:

show connect port [d:1000, 256] input

2. Display all output ports:

show connect port output

3. Display all input and output ports:

show connect port

4. Display input and output connections of ext2:

show connect register ext2

Related Commands:

connect port , connect register, map

17.6.125 show interrupt

show interrupt [nmi/int0/int1/int2/all]

show interrupt vint

The **show interrupt** command displays (in the Log Window) all the active interrupts with their attributes as they were defined via the command generate interrupt.

Examples:

show interrupt int0
show interrupt all

Related Commands:

generate interrupt

17.6.126 show locals

The **show locals** command displays (in the Log Window) the local variables as displayed in the Locals Window. When Debugger log mode is **on**, the contents of the Log window is saved to a file, and therefore this command is helpful in creating tests whose results can be analyzed off-line.

Examples:

show locals

17.6.127 show memory classes

The **show memory classes** command displays (in the Log Window) all the memory classes and their attributes that were defined with the define memory command.

Examples:

show memory classes

Related Commands:

define memory, undefine memory

17.6.128 show memory map

show memory map [AddressRange]

The **show memory map** command displays (in the Log Window) the memory mapping (defined with the command **map**) for the optionally specified data address range. If no data range is specified, then the complete map of code and data memory is displayed.

For more information refer to the map and unmap commands.

Examples:

1. Complete map of code and data memory:
show memory map
2. Code memory map of specified area:
show memory map [C:4000,C:7FFF]
3. Show memory mapping of specified segment:
show memory map [d2:4000, 0x2000]

Related Commands:

define memory , map , undefine memory, unmap

17.6.129 Show message

show message assert/debug/error/fatal/info/warning/dialog/all

The **show message** command displays (in the Log Window) the message display status (i.e. disabled or enabled) of messages of the specified severity. Messages can be disabled or enabled via the commands **disable message** and **enable message** respectively. For more details, see the commands **disable message** and **enable message**.

Examples:

show message error
show message all

Related Commands:

disable load coff reset , enable message

17.6.130 show register

The **show register** command displays (in the Log Window) all the specific DSP Core registers from the Registers Window. When Debugger log mode is **on**, the contents of the Log window are saved to a file, and therefore this command is helpful in creating tests whose results can be analyzed off-line.

Examples:

show register

17.6.131 show register map

show register map [extExtRegId]

The **show register map** command displays (in the Log Window) the external register mapping (defined with the command **map**) for the optionally specified external register. If no register is specified, then the complete map of external registers is displayed.

For more information refer to the map and unmap commands.

Examples:

1. Complete map of code and data memory:

show memory map

2. Code memory map of specified area:

show memory map [C:4000,C:7FFF]

3. Mapping of external register ext0:

show register map ext0

4. Mapping of all external registers:

show register map

Related commands:

map , unmap

17.6.132 start cevax < MSSVersion >

start cevax < MSSVersion > [code InternalCodeMemorySize k] [data InternalDataMemorySize k]

This command supports working in relevant CEVA-X16xx MSS mode. By default the debugger is defined to work as CEVA-X1620 (or detected license). The User can change the CEVA-X16xx type to 1620, 1622, 1641 or 1643 by the new CLI. When the command is invoked, the debugger checks for a valid license. If a valid license exists then the debugger's title is changed as a validation mark.

The command also supports several internal configurations of the Internal Code and Internal Data memory sizes. If not specified the data and code memory sizes are automatically detected by the 'load coff' command.

Note: This product support is protected under a distinct IP license.

17.6.133 start cevatl3<MssVersion>

start cevatl3<mss version> [code InternalCodeMemorySize k] [data InternalDataMemorySize k]

This command supports working in relevant CEVA-TL321x MSS mode. By default the debugger is defined to work as CEVA-TL3210 (or detected license). The User can change the CEVA-TL321x type to 3210 or 3211 by the new CLI. When the command is invoked, the debugger checks for a valid license. If a valid license exists then the debugger's title is changed as a validation mark.

The command also supports several internal configurations of the Code and Data memory. The data and code memory size are automatically detected by the 'load coff' command.

Note: This product support is protected under a distinct IP license.

17.6.134 Start cevatl4 <MSSVersion>

start cevatl4<MSSVersion> [code <InternalCodeMemorySize> k] [data <InternalDataMemorySize> k] [<feature>]

- MSSVersion is one of 10, 11, 20 or 21
- InternalCodeMemorySize is one of 4, 8, 16, 32, 64, 128, 256 or 512
- InternalDataMemorySize is one of 6,8,12,16,24,32,48,64,96,128,192,256
- feature is one of audio, voice or compatibility

This command supports working in relevant CEVA-TL4xx MSS mode. By default the debugger is defined to work as CEVA-TL410 (or detected license). The User can change the CEVA-TL4xx type to CEVA-TL411, CEVA-TL420 or CEVA-TL421 by this CLI. The user can also select the CEVA-TL4xx feature (audio, voice or compatibility) using this CLI.

When the command is invoked, the debugger checks for a valid license. If a valid license exists then the debugger's title is changed as a validation mark.

The command also supports several internal configurations of the Internal Code and Internal Data memory sizes. If not specified the data and code memory sizes are automatically detected by the 'load coff' command.

Note: This product support is protected under a distinct IP license

Examples:

1. **Set the default: Internal CodeMemorySize, InternalDataMemorySize and feature:**
   start cevatl410

2. **Set the InternalCodeMemorySize to 128k, InternalDataMemorySize to 256k and the feature to voice:**
   start cevatl410 code 128k data 256k voice

17.6.135 start emu

start emu [PcPortAddress \ usb\ ethernet] [onthefly\debug]

The **start emu** command initiates Emulation mode of the Debugger. The command causes the memory map and breakpoints of the emulator to be reset (i.e. all memory is unmapped, all breakpoints are cleared). After this command the user must remap and reload his program and set back program breakpoints.

ethernet – **Relevant only for J-Box2**. When using the **ethernet** keyword, the communication between the PC and the J-Box will be created while using Ethernet connection.

PcPortAddress – The PC I/O port address for communicating with the Emulator. When applying the command without port address, default port address will be used 0x378.

Note that the –port <address> command line option can be used for setting the default port address.

Three Emulation modes are available:

1. **Normal**– In this (default) mode, the Debugger will reset the emulator HW, upload register contents and wait for the user command (define memory, map, load coff, go**…).**
2. **On-the-Fly** - Enables connecting the Debugger to a working application that is already loaded and currently running on the target hardware. This mode performs a warm/light connection by skipping the load Monitor and reset handshake and putting the Debugger in a waiting loop for the Monitor Idle signal (as after the **go** command was executed in the **Normal** mode).
3. **Debug** - Enters Emulation mode without any interaction with the hardware. This mode can be used for simple FPGA read/write when very basic problems appear with new hardware to which the Debugger cannot connect. This mode can be used as a very simple and low-level communication channel to such hardware. After entering this mode, the emu debug input command can be used for reading an FPGA register, and the emu debug output command can be used for writing an FPGA register.

The Debugger's title, in the title bar of the main window, is appended with **< Emulation Mode >** when in **Normal** Emulation mode, **< Emulation Mode – On-the-Fly >** when in **On-the-Fly** Emulation mode and **< Emulation Mode – Debug >** when in **Debug** Emulation mode.

To return to Simulation Mode, use the command. The title in the title bar will switch back to Simulation Mode.

For more information, see Emulation Mode at CEVA-Toolbox Debugger User's Guide Vol-I.

Notes:

1. If the Debugger command line option '-port <address >' is applied, using the **start emu** command without specifying a port address, will take the value according to the command line value used .

2. The **start emu** command is valid in Simulation Mode **only** for switching to Emulation Mode.

3. PCMCIA Paralle-Port connection (V9.1 and newer) should be used in conjunction with the JBOX.

Examples:

start emu
start emu 0x378
start emu 0x378 onthefly
start emu debug
start emu ethernet
start emu ethernet onthefly

Related Commands:

emu set internal register, emu get internal register,
emu set mode, emu set communication type

17.6.136 Start log

start log [LogFileName] [append]

The **start log** command starts recording all Log Window messages (both user-originated and system responses) to the optionally specified file or to the file ***SESSION.DBG*** if none is specified. Unless the **append** attribute is used, if the file already exists, it is overwritten. This file can be played back via the 'call' command. Recording is terminated via the stop log command. **Note** that the recording of system responses is prefixed with an exclamation mark (!) in the first column so that they will be ignored during playback.

Multiple log files can be opened by recurring the **start log** command. In this case, executed CLI commands will be logged to all opened log files simultaneously. When multiple log files are opened and the **stop log** command is issued without a log file name as an input, the last opened log file is closed. The **stop log** all command can be used to close all opened log files.

Note:

Default filename is ***SESSION.DBG***, default extension is ***.DBG*** or ***.LOG***

Examples:

1. Start log to SESSION.DBG:

start log

2. Start log to test.dbg, append to the file end:

start log %MyLogDir%\test.dbg append

Related commands:

call, stop log

17.6.137 start memory profiler

start memory profiler

The **start memory profiler** command starts memory profiler tracing and builds an information database which can be open in the CEVA-Toolbox IDE to show the profiling results.

This command can be used also in emulation mode for CEVA-X164x, CEVA-XC32x, CEVA-XC42xx, CEVA-TL321x and CEVA-TL4xx only. In this case the command activates the emulation profiler and the dedicated profiler window is opened.

Examples:

start memory profiler

17.6.138 start tcl

The **start tcl** command initiates TCL mode for the Debugger. When the **start tcl** command is issued, the Debugger enters TCL Mode, in which each command entered is interpreted as a TCL command.

When in TCL mode, the user can still use the CLI commands, in addition to the TCL commands. As some CLI commands are the same as TCL commands, we differentiate between CLI and TCL commands by the prefix **deb** for any Debugger CLI command.

For more information, refer to CEVA-Toolbox Debugger User's Guide Vol-III.

Examples:

start tcl
deb step

Related Commands:

deb stop tcl

17.6.139 start timer

start timer [timerName]

This CLI, combined with the "stop timer" CLI, enables the user to measure performance in his application.

This CLI can be used at any stage in the user's DBG script. When the user enters the "stop timer" CLI, the debugger will show the time elapsed between two calls in milliseconds resolution.

Note:

When using "start timer" without timer's name, the debugger will use its default timer.

Examples:

start timer
start timer timer01

Related Commands:

stop timer

17.6.140 start trace

start trace [TraceFileName]

The **start trace** command starts writing binary trace frames to the optionally specified file for each instruction that is subsequently executed. If a file of the same name already exists, then it is overwritten. To stop the trace, use the command **stop trace**.

Note:

If a file name is not supplied, a file name <basename of the loaded COFF file>.frm is created. Default extension is .FRM

Examples:

start trace trace.frm

Related Commands:

stop trace

17.6.141 step

step [NumberOfSteps]

The **step** command starts executing the loaded program from the current pc for one instruction or for the optional number of specified instructions.

When the value of the step is greater than one, execution will terminate prematurely when one of the following conditions occurs:

- execute at a breakpoint
- execute beyond end-of-program
- execute at the second word of a two-word instruction
- execute at a DW
- execute at unmapped memory
- ^pause command (when in Emulation Mode)
- stop bit set or branch to self (when in Simulation Mode)
- simulation engine exception (when in Simulation Mode)

This command can also be invoked by pressing the 'Alt+F11' keys, or by pressing the 'Assembly step' button on the toolbar (one instruction will be executed).

Note that in Simulation Mode, this command cannot be stopped by the **^pause** commands.

Examples:

step
step 100

Related commands:
reset, continue, pause, go, run,
load coff, next, next source, step source, step out source, step source <N>

17.6.142 step out

The **step out** command starts executing the loaded program from the current pc until the first instruction located (dynamically) following a **ret** type family instruction. Practically, this command can be used to step out of the current Assembly routine.

Execution will terminate prematurely when one of the following conditions occurs:

- execute at a breakpoint
- execute beyond end-of-program
- execute at the second word of a two-word instruction
- execute at a DW
- execute at unmapped memory
- ^pause command (when in Emulation Mode)
- stop bit set or branch to self (when in Simulation Mode)
- simulation engine exception (when in Simulation Mode)

This command can also be invoked by pressing the 'Shift+Alt+F11' key, or by pressing the 'Assembly step out' button on the toolbar.

Note that in Simulation Mode, this command cannot be stopped by the **^pause** commands.

Examples:

step out

Related commands:

reset, continue, pause, go, run,
load coff, next, step, next source, step source, step out source, step source <N>

17.6.143 step out source

The **step out source** command starts executing the loaded program from the current source file statement until the first source file statement that is located externally to the current function. Practically, this command can be used to step out of the current function back to the source file statement that follows the statement that called it. For this to occur correctly, the source files must be compiled with debug information (i.e. **–g** Assembler switch for Assembly source files, or **–g/g0/g1/g2/g3** Compiler switch for C\C++ source files).

Execution will terminate prematurely when one of the following conditions occurs:

- execute at a breakpoint
- execute beyond end-of-program
- execute at the second word of a two-word instruction
- execute at a DW
- execute at unmapped memory
- ^pause command (when in Emulation Mode)
- stop bit set or branch to self (when in Simulation Mode)
- simulation engine exception (when in Simulation Mode)

The ***step out source*** command can also be invoked by pressing the Shift+F11' keys, or by pressing the 'source step out' button on the toolbar.

Note:

This **step out source** command is used for source level debugging only, when the debugging mode is either **source mode** or **mixed mode**. When this command is given and no debug information exists, it is equivalent to the **go** command.

Examples:

step out source

Related commands:

reset, continue, pause, go , run,
load coff, next, step , step out, next source, step source, step source <N>

17.6.144 step source

step source

The **step source** command starts executing the loaded program from the current source file statement, until the next source file statement. For this to occur correctly, the source file must be compiled with debug information (i.e. **-g** Assembler switch for Assembly source files, or **–g/g0/g1/g2/g3** Compiler switch for C\C++ source files).

Execution will terminate prematurely when one of the following conditions occur:

- execute at a breakpoint
- execute beyond end-of-program
- execute at the second word of a two-word instruction
- execute at a DW
- execute at unmapped memory
- ^pause command (when in Emulation Mode)
- stop bit set or branch to self (when in Simulation Mode)
- simulation engine exception (when in Simulation Mode)

The **step source** command can also be invoked by pressing the **'F10'** key, or by pressing the **source step** button on the toolbar.

Note:
The **step source** command is used for source level debugging only, when the debugging mode is either **source mode** or **mixed mode**. When this command is given and no debug information exists, it is equivalent to the **go** command.

Examples:

step source

Related Commands:

reset, continue , pause , go , run ,
load coff, next, step, next source, step source, step out source, step step source <N>

17.6.145 step source <N>

The **source step <N>** CLI executes N source steps. This CLI is equivalent to writing the **step source** CLI for N times.

Examples:

source step 3 ; perform three source steps

Related Commands:

step, step out, step out source, step source

17.6.146 stop emu

The **stop emu** command stops Emulation Mode, and returns to (the default) Simulator Mode. The emulator is started with the **start emu** command. After this command, the user must remap, reload and reset his breakpoints.

Note:

This command is valid in Emulation Mode **only**.

Examples:

stop emu

Related Commands:

start emu

17.6.147 stop log

stop log [all]

stop log [LogFileName]

The **stop log** command stops recording the Log Window contents, and closes the file being used to capture the contents via the **start log** command. If no logging is in progress, the command has no effect other than to issue a warning. **Note** that a log file cannot be accessed for viewing or playback during recording, i.e. until recording has been stopped and the file is closed via this command.

Multiple log files can be opened by recurring start log commands. In this case, executed CLI commands will be logged to all opened log files simultaneously. When multiple log files are opened and the stop log command is issued without a log file name as an input, the last opened log file is closed. The stop log command can be used to close all opened log files.

Examples:

stop log

Related Commands:
strat log

17.6.148 stop memory profiler

This CLI is relevant for the CEVA-X, CEVA-XC, CEVA-TeakLite-III and CEVA-TeakLite-4 only.

" The **stop memory profiler** command stops the profiling database creation built by **start memory profiler**

Examples:

stop memory profiler

Related Commands:

start memory profiler

17.6.149 stop timer

stop timer [timerName]

This CLI, combined with the “start timer” CLI, enables the user to measure performance in his application.

This CLI can be used at any stage in the user’s DBG script. When the user enters the “stop timer” CLI, the debugger will show the time elapsed between the “start timer” call & “stop timer” in milliseconds resolution.

Note:

When using “stop timer” without timer’s name, the debugger will use its default timer.

Stopping timer without initializing it first is prohibited.

Examples:

stop timer

stop timer timer01

Related Commands:

start timer

17.6.150 stop trace

The **stop trace** command stops writing binary trace frames, and closes the file that was used to capture the data via the start trace command. If no trace was in progress, then the command has no effect other than to issue a warning. Note that a trace frame file cannot be accessed for viewing while a trace is in progress, i.e. until the trace has been stopped and the file closed via this command. When the trace feature has been activated, the emulator cannot run in true real-time mode.

Examples:

stop trace

Related Commands:

start trace

17.6.151 transfer

transfer [external] AddressRange to StartAddress

The **transfer** command is used to transfer code or data memory content from one location to another. If the **external** option is used, the transfer is performed to the current active connected external CEVA-Toolbox Debugger.

Notes:

- Memory destination and source can be of different memory types (code/data).
- When destination and source are of different memory types and the memory size of the source is bigger than the memory size of the destination, an error will be generated.

Examples:

transfer [D:5000,100] to D:F000

transfer [C:6500,100] to D:1200

transfer external [C2:1000,C2:2000] to C2:8000

transfer external [C2:4000,C2:5000] to D0:C000

Related Commands:

connect external debugger, disconnect external debugger, run external cli

17.6.152 undefine memory

undefine memory MemoryName

The **undefine memory** command is used to delete a previous definition of a memory class.

Notes:

- Predefined memory classes cannot be erased (e.g. xram, yram).

Examples:

define memory my_class
undefined memory my_class

Related Commands:

define memory, map, show memory classes, show memory map

17.6.153 unmap

unmap [memory] AddressRange
unmap [register] extExtRegId

The **unmap** command unmaps the specified memory address range or external register. At initialization, all memory and all external registers are unmapped. It is good debugging practice to unmap any memory not needed for the application, since the Simulator can trap references to unmapped memory. In addition, Emulator performance is improved when unneeded memory is unmapped.

When using a segments/paging application, it is possible to map different address ranges to different segments/pages. The mapping CLI commands – **map memory**, **show memory map** and unmap commands were extended to enable the user to specify a segment number in which the mapping will take place. Mapping commands that do not specify a segment number will map all defined segments for the requested address range (as done before).

Notes:

- It is a good debugging practice to map **only** memory that is needed for the application, since:
  1. The Simulator **only** can trap references to an unmapped memory.
  2. Emulator performance is improved when the minimum memory possible is mapped.
  3. With every mapping command, the Debugger allocates more memory that increases the memory usage of the Debugger and may slow the machine - **make sure to map only the necessary memory area**.
- The unmap command unmaps memory that was mapped by the map command; **It does not free it from the Debugger's memory usage**. Should the user not care to lose the entire memory contents, he may use the debugger new session command in order to free mapped memory.
- **It is a good practice to use the Debugger new session command at the very start of a new script file that contains mapping commands on previously mapped area**. Note

that using the **debugger new session** command will bring the Debugger to the state when it was first invoked, thus, the already-mapped memory will be lost along with its contents.

Examples:

unmap memory [C:0000,C:7FFF]
unmap [C:0+endofprogram+1,C:7FFF] ; endofprogram is a readonly pseudo-register
unmap [D1:0000, 1000] ; unmap only data segment 1, in the requested addresses.

Related Commands:

load coff symbols, load coff , map, show memory map

17.6.154 unprotect

unprotect [port] PortAddress
unprotect [register] extExtRegId
unprotect data memory

The **unprotect** command sets the specified memory-mapped I/O address to be not protected. At initialization, all input ports with **ior** access are protected.

The **unprotect data memory** command disables the protection of reserved data memory areas (BIU, OCEM, etc.). In this mode, when overwriting these data memory areas, no warning will be issued. By default these data memory areas are protected.

Note:
It is good debugging practice to leave all input ports in protected state. However, if it is essential to view the updated values each time the simulator stops running the application program, perform unprotect command to the port. Then, if such an unprotected input port is connected to a file or user-defined DLL function, each time the screen has to be refreshed, the file will be read (file pointer advances one line), or the DLL function will be called. If the unprotected input port is ***not*** connected, then each time the screen is refreshed, a dialog box will pop-up and request a value from the user !

Examples:

unprotect port d:4000
unprotect InPort.Codec
unprotect register ext3
unprotect ext4
unprotect data memory

Related Commands:
connect port, connect register, map, protect

17.6.155 use

use **AssemblySectionName**

The **use** command permits abbreviated symbol references for symbols in the specified section. **Note** that the full name of a symbol is in the form <Section>.<Offset>. When an abbreviated symbol reference is encountered, i.e. a symbol not prefixed with a section name, the default section name used is the one specified by this command. In order to specify the address of the start of a section, use the form <Section>.0.

Examples:

use MyCodeSection

17.6.156 ver

The **ver** command is used to print the Debugger version into the Log Window.

Example:

ver ➔ *prints 9.0 for V9.0 Debugger*

17.6.157 verify

verify CodeAddressRange

The **verify** command is used to verify that a particular ASIC that is being debugged in Emulation Mode contains the same software as the loaded COFF file. To use this command, the COFF file must be loaded first assuming to be identical to the contents of the chip's ROM.

Notes:

1. The **verify** command is valid in Emulation Mode **only**.

2. The command will fail if the ROM of the chip is protected!

Examples:

start emu
load c:\oak\mask\chip.a
verify [c:0000,c:1fff]

Related Commands:

start cevax < MSSVersion > load coff
load coff symbols

17.6.158 wait

wait NumberOfSeconds

The **wait** command causes the Debugger to stop processing CLI commands for the duration of the timer (in seconds) specified in this command. The timer is killed by any caret (^) command (e.g. ^**continue**). This command is useful for product demonstrations. The CLI better be used in a script before go or continue

Note:

The value (in seconds) must be in the range of 0..64.

Examples:

wait 60

Related Commands:

continue

18. Index

The following is the volume-II Index :