CEVA-Toolbox™

CEVA-XM4™ Debugger API Reference Guide

CEVA-XM4

V15.1.0

July 2015

DSP Division

Documentation Control

History Table:

Version	Date	Description	Remarks
1.0	31/12/2013	Document creation	
10.3	1/10/2014	Update for CEVA-XCxxxx V10.3	
11.0.0.Beta	22/01/2015	Update for CEVA-XCxxxx V11.0.0.Beta	
15.1.0	19/07/15	Update for CEVA-XM4 V15.1.0	

Disclaimer and Proprietary Information Notice

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

1 General Overview

The CEVA Debugger API is a C++ interface providing complete debug capabilities of all CEVA cores (Simulation & Emulation). It consists of several different APIs which support its functionality.

1.1 Motivation

CevaDbg API provides a unified debugging interface to all of CEVA's cores. Since the API is composed mainly from pure virtual C++ classes it provides great modularity capabilities, and avoids binary compatibility problems. It enables a connection to many different user interface modules such as GUI, CLI, ESL system simulation and so on, independently from the debugger's implementation.

The API is generic and using it does not require any knowledge of the core's architecture (registers, resources, configurations, and so on). All of this information is provided through dedicated API functions. This feature makes the integration of new cores in the user interface very easy, since most of the work is done under the Debugger's hood.

The API is completely standard C++, and does not use any other libraries. Users can write standalone drivers for the debugger, without having concerns over compatibility issues.

1.2 Main Features

A list of the Debugger's API main features:

1. **Full source debug capabilities including**
    a. Simulation – ISS (Instruction Set Simulation).
    b. Simulation – CAS (Cycle Accurate pipeline Simulation).
    c. ESL Simulation (ISS & CAS) – with full core and MSS signals exposure.
    d. Emulation – Using CEVA-Jbox.
2. **Generic**

    Get registers and resources information without necessarily knowing which resources the core has.

    Access memory, perform mappings and configure memory behavior in a generic way.

    Configure each core according to its specific attributes.

    All CEVA cores share the same APIs and support them in the same manner.
3. **Event Driven**

    The Debugger API works in an event driven environment, it is not needed to check every cycle if an event has occurred, there are observer classes which can register on the various APIs in order to get a notification of a certain event (breakpoint, notification, external access, signal update etc.)
4. **Versioning**

    The Debugger API has a built in versions handling concept, implying that if a user has a working application which uses the API, if a new API version is being supplied by CEVA, the user does not have to recompile the application and still benefit from improvements in the current API version he uses, if the user wishes he can recompile the application to use new features that will be introduced in the newer version.

1.3 System Modeling

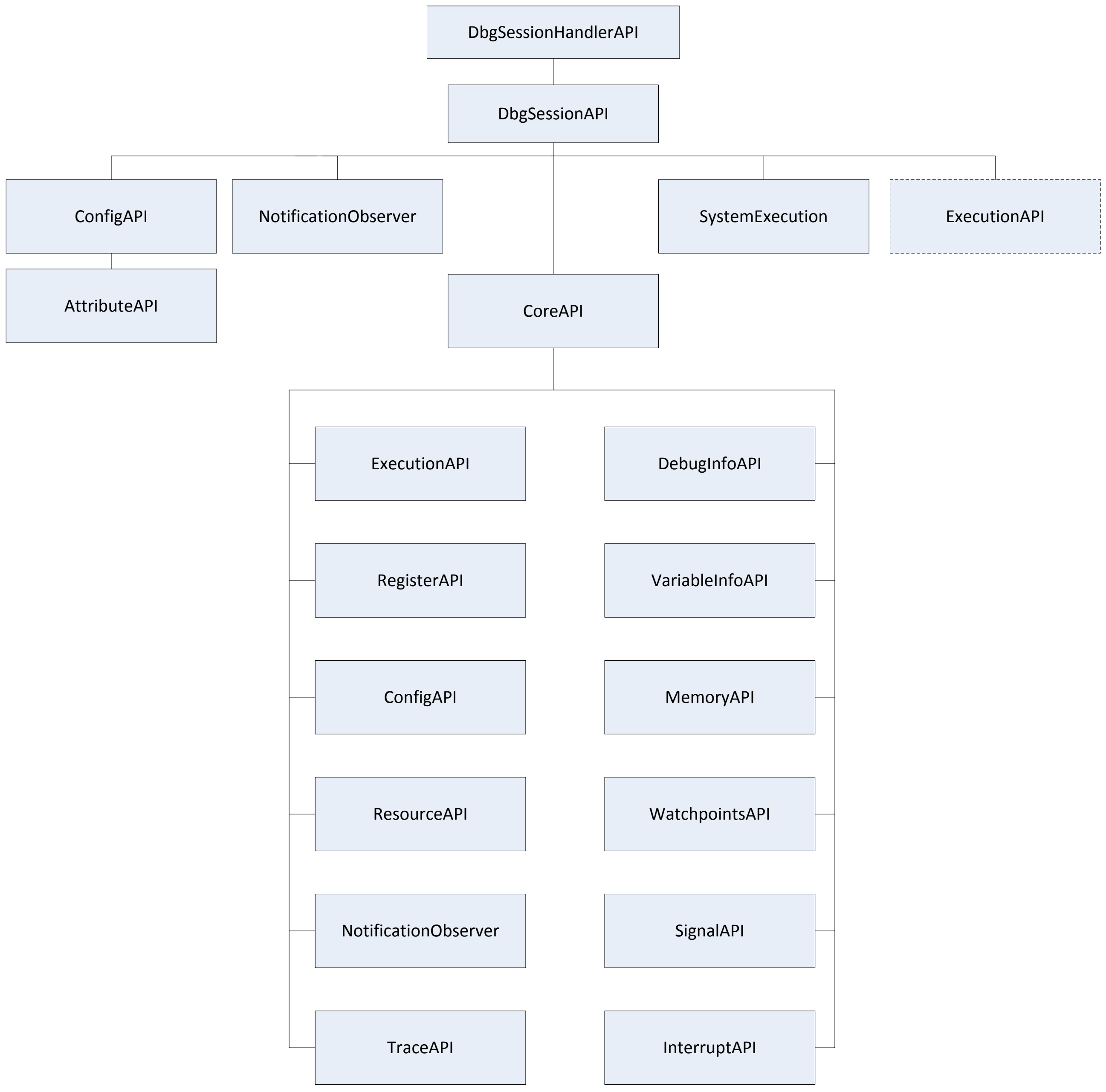
It is possible to load several cores in the same Debug session, in this way users can model a system composed from any of CEVA cores, and using the different APIs also define the connectivity between the cores, because the API is C++ based it is also possible to connect any other vendor core to the system and use the DBG API to define the connections of signals/buses between the cores.

It is also possible to remotely debug CEVA cores in such system using the Remote debug mechanism.

In this way after running an application composed of several CEVA cores in a C++ environment, the user can open the CEVA-ToolBox IDE to connect to that system and get full source debug visibility of any of the CEVA cores.

See also System Execution and remote debug

2 Class Overview



3 API Description

3.1 DbgSessionAPI

Instantiates a debug session, which can load as many cores requested from any CEVA supported cores.

User can simply load a core into the debug session if its name is known (for example: 'CEVA-XC323').

Class reference at "Debugger API Class Reference".

3.1.1 DbgSessionAPI_v2 changelog:

- " **startRemoteDebugServer**" – added argument to set the port number for the debug server to listen on.

Class reference at "Debugger API Class Reference".

3.1.2 Common Usages

- Query the DbgSessionAPI which cores are supported by the API.
- Get information about one of the supported cores by its index (for example, MainCore: CEVA-XC, SubCore: CEVA-XC4210).
- Load a core (by name or by index) to the debug session. (returns CoreAPI).
- Unload core from the debug session.
- Get the debug session ConfigAPI – configurations relevant for the whole session and not core specific.
- Register / Unregister a NotificationObserver – to receive notifications when an event occurred which is relevant to the whole session and not core specific.
- Reset session – reset the whole session, as oppose to resetting a core, this unloads all cores in the session and re-loads them, losing any loaded binary files or memory/registers initializations.
- Start a remote debugging server – used for remotely debugging this session and cores from a remote CEVA-ToolBox IDE.
- Handle the SystemExecution – The DbgSessionAPI initializes its own SystemExecution, which executes all cores in the session sequentially, and breaks execution if any of the cores stopped execution (breakpoint or other).

  It is possible for the user to utilize this SystemExecution to execute the system, or to implements his own SystemExecution and handle the execution by himself, notifying the DbgSessionAPI that the system execution has changed.

3.2 CoreAPI

Main Core interface holds an interface to everything underlying which is specific to the core currently loaded in the debug session, including simulation / emulation of that core, debug info, memory spaces, execution, input/output signals exposure and so on. Most CEVA DBG API handling begins with accessing and requesting information from the CoreAPI.

Class reference at “Debugger API Class Reference”.

3.2.1 CoreAPI_v2 changelog:

- ”**getFunctionalPortAPI**” – new method to retrive the ”FunctionalPortAPI” associated with this CoreAPI.
- ”**insertCli**” - updated to recieve another argument ”isDisableDeprecated” – gives the option not to run a cli if it is signed as deprecated.

  For example if ”step” CLI was marked as deprecated, calling ’insertCli(”step”, true)’ will not perform anything and will notify the user that the given CLI was deprecated. This method is for internal usage only, for inserting CLIs use with default ”isDisableDeprecated = false”.
- ”**getRegisterAPIByAddress**” **-** get a RegisterAPI instance of a register mapped to the address given.
- **Master / Slave port information** – to support SystemC API.

Class reference at “Debugger API Class Reference”.

3.2.2 Common Usages

- Query core name (which can be set to a custom name, for example, “Controller”).
- Load a binary (coff) file into the core (or just the symbols).
- Get DebugInfoAPI pointer for source level debugging of the core.
- Generically get information about core resources, first request the number of resources (or resources groups) and then recursively get all information about all resources and their sub resources, using the ResourceAPI.
- Similar to resources, all core registers can be fetched in a generic manner, the user does not need to know how many registers are available or their information (size, name and so on). This can be done recursively on all core registers and register parts, using the RegisterAPI.
- Core memories – query the core for how many memories (code/data/io and so on) the core has, and then get information on a specific memory space using the MemoryAPI.
- Get the core’s configuration and customize it, using the ConfigAPI, which enables changing simulation mode, memory sizes, number of blocks and all other core specific attributes.
- Control core breakpoints – set/cancel/enable/disable and so on using WatchpointsAPI.

- Control core execution using ExecutionAPI.

  Note: The core's ExecutionAPI controls execution of this specific core alone even if it is part of a system of cores loaded into the debug session. To control the system execution, see SystemExecution under DbgSessionAPI.

- Generate simulated core interrupts using InterruptAPI.

  Note: It is also possible to control the actual input signals of the cores (such as interrupt input signals) manually, for example by setting the value high, or using the InterruptAPI to generate interrupts at a certain frequency or number of times, and so on.

- Start / Stop the core tracing mechanism using TraceAPI. To get a detailed status of core registers, instructions and memory accesses every cycle.
- Register / Unregister a NotificationObserver – to receive notifications when an event occurred which is relevant to the specific core, for example: print out to console, error messages from the core, call back events and so on.
- Insert CLI (command line interface) – to support running scripts, including TCL commands.
- Call a CLI script file.
- Start, Stop or write to a debug log, saving to a file all printouts from the core, regardless if a notification observer has been registered.
- Enter / Exit emulation, connecting to a CEVA core on FPGA or Development board using CEVA-Jbox.
- Complete exposure of input/output signals of the core & MSS (memory subsystem), including interrupts, external ports and so on, using SignalAPI.

3.3 ConfigAPI

The ConfigAPI is used to act as an interface to get the attributes of the core, query it to get the amount of AttributeAPIs in the core and get each attributes' information in the list.

It is possible to load attributes automatically from XML file using the dedicated method:

```
loadAttributes(const char * pFileName)
```

This method loads attributes values from an XML file which can be found per core under <ToolsDir>\CevaDbgApi\AttributeXML.

It is possible to load an entire XML of attributes into the core to change many attributes or load as minimal as 1 attribute in a minimum XML format for example:

```
<CEVA-XC4210>
     <Attribute>
            <Name>AXI width</Name>
            <Value>128 Bit</Value>
     </Attribute>
</CEVA-XC4210>
```

Note that attributes in the XML need to be under a root node. In this example :

```
<CEVA-XC4210>
     <Attribute>
          …
     </Attribute>
     <Attribute>
          …
     </Attribute>
     ….
</CEVA-XC4210>
```

It is also possible to load separate XML configurations files for each attribute requested to load.

This XML file loaded into the cores' ConfigAPI will only change the current AXI width to 128 bits.

See also a list of all attributes per core:

CEVA-XC323 Attribute List

CEVA-XC4210 Attribute List

CEVA-XC4500 Attribute List

CEVA-TL3210 Attribute List

CEVA-TL3211 Attribute List

CEVA-TL410 Attribute List

CEVA-TL411 Attribute List

CEVA-TL420 Attribute List

CEVA-TL421 Attribute List

CEVA-XM4 Attribute List

Class reference at “Debugger API Class Reference”.

3.4 AttributeAPI

Used to get or change a specific attribute of a debug session or a specific core in the session.

Core specific attributes, for example:

- Execution mode – ISS (Instruction set simulation) / CAS (Cycle accurate simulation) / Emulation.
- Program arguments.
- Memory internal sizes, number of memory blocks in internal memory.
- Attributes of external ports.
- Hardware-specific configuration.

Note:

All configurations and attributes are generic and change from core to core, the user needs to query and iterate over all attributes in order to read or modify them.

Class reference at “Debugger API Class Reference”.

3.4.1 Common Usages

- Get attribute information – name, type, description, group of attributes associated with, read-only (or not), is valid (can be not valid by some other attributes value).
- Get / Set the attribute value to affect the core status, or HW configuration.

Note:

The AttributeAPI values are generic – meaning: each attribute can have a string value or number value or internal enum.

For example, the number of internal data blocks is an attribute, its type is list (*AttribDisplay_E_List*) in order to change the value the user can either change according to the name of the value:

pNumOfBlocksAttrib->setValue(“2”); // will change to 2 internal data blocks.

Or change according to the index in the list – the list of possible values can also be queried from the AttributeAPI:

pNumOfBlocksAttrib->setValue(1); // will change to 2 internal data blocks – given that the possible values are: 0 - 1 blocks, 1 - 2 blocks, 2 - 4 blocks, 3 - 8 blocks.

3.6 NotificationObserver

The Notification Observer is used to get notifications (call backs) from the core or debug session when a certain event occurs. Its main function "*handleEvent*" recievs an *Event* class object which can be used to identify the event and act accordingly.

An Event class object is comprised of the following:

- Type (Error/Warning/Info/Dialog box/Callback)
- Name
- Description
- Title
- Input/Output parameters – the event might send/recieve arguments from the event handler. For example:

  Dialog box event that requests a yes/no question from the user:

  - Event Type: EventType_E_DialogBox
  - Input Parameter 1 name: "Yes"
  - Input Parameter 1 name: "No"

  Now the event handler (in the notification observer) needs to set the value of the correct input parameter according to the user's selection to the dialog box shown.

  Callback event that notifies that a coff binary file has been loaded:

  - Event Type: EventType_E_CallBack
  - Output Parameter 1: type of callback "CallBackType_E_LoadCoff".
  - Output Parameter 2: name of coff file (string).

Other functionalities the NotificationObserver supports is writing out to an output according to its name (STDOUT,STDERR) or getting an input line from the user (when using scanf for example).

Class reference at "Debugger API Class Reference".

3.7 SystemExecution

The SystemExecution class controls the execution of all cores loaded in the DbgSession. It can be used to execute those cores in the following manner:

The DbgSessionAPI implements its own "SystemExecution" class object which runs all cores in the session 1 cycle each time sequentially one after the other, starting from the coreAPI given as an argument to the SystemExecutions' "execute()" function, when one of the cores breaks execution (breakpoint, end of program, error and so on) all other cores stop execution.

This SystemExecution object will use another class object "*SystemExecutionWrap*" for running pre/post cycle operations, this class is implemented by the user and sent to the SystemExecution class for using it.

The SystemExecution object which the DbgSession uses can be replaced by a user implementation of it (which makes the SystemExecutionWrap redundant as the user controls all steps of execution).

Note:

The DbgSession uses the SystemExecution class object automatically to control the cores execution when remote debugging a running system. Hence, if the user wishes to replace the internal SystemExecution he must comply to the following:

- User can perform his own pre/post cycle operations within the "execute()" function.
- User can run the cores in any mode, step by step, different clock ratio for each core, run core in "run()" mode (needs to be synchronized with other cores and the remote debugger if connected). Etc.
- **User must return** "*Status_E_Error*" so that the remote debug can stop execution opon break from the system.

  Execute() method should return error if one of the specific cores in the sytem returned error status from its ExecutionAPI or if a core had stopped on breakpoint, branch to self, Simulation error (memory violation) or any other method which will break the execution.

  This is for the remote debugger (if connected) to be aware that the execution initiated by it should be aborted.
- **User must implement** "getCoreLastExecStatus" – this is used for the remote debugger to know which core caused the system to break or to identify the opposite – is the core that initiated the execution breaked the system or not. To know if a remote callback to abort execution is needed.

Class reference at "Debugger API Class Reference".

3.7.1 SystemExecution_v2 changelog:

- "**init**" – new method to init the user's class – will be called every "loadNewCore()" is called and added to the session.
- "**getCoreLastExecStatus**" - added argument to retrieve information for the debugger if the last execution was due to a breakpoint and if so which breakpoint id. For non-breakpoint need to return 0.
- "**run**" - new method for the user to distinguish a "run" command rather than a single "execute" – user can just call execute or implement a faster "run" until error is reached, such as "runNumOfCycles()".
- **"externalAbort"** – new method to inform the user's class that an external abort (from the debugger) is called.

Class reference at "Debugger API Class Reference".

3.8 ExecutionAPI

Main core execution interface (per core in the debug session).

Opon requesting the ExecutionAPI from the CoreAPI a license request is initiated for the specific core, if license does not exists the user will be asked to enter license information.

Execution functions will return a *Status_E_Ok* when the core has been executed. If the core breaked execution (by breakpoint/error and so on) it will be notified to the NotificationObservers (if any).

The Execution of the core is done in a **separate thread**, so the core might be still running after returning from the specific execution request, a notification of type "CallBack" and parameter "*CallBackType_E_ExecuteDone*" will be sent when the core finished its execution (even if there is a notification in the middle of execution).

Simulator clock counter:

When running in simulation mode **only**, there is an internal simulation "clock" counter, which counts the number of cycles that was exceuted.

Cycle Accurate mode: Executing a "stepCycle" will simulate driving a single clock into the core, and so "getClock" after a stepCycle will return 1 cycle more than before execution – for example stepping over a branch instruction with single stepCycle will take a few executions for the simulation to branch to the target address.

Instruction set mode: Executing a "stepCycle" will simulate steping over the current assembly instruction and fully execution it, which might be more than 1 cycle in case the instruction has an extra cycle count – for example stepping over a branch instruction will branch the simulation to the target address and increase the clock by number of clocks it takes to branch to the target.

The same behavior is true for running "**runNumCycles**" method – in ISS requesting to run 5 cycle "runNumCycles(5)" might result of a jump in "getClock" of more than 5 cycles in case the executed instructions have a greater latency than 1. Also it's possible for the simulator to run less than 5 cycles in case there was some interrupt (breakpoint / core interrupt / external access etc.).

Note:

The internal simulation clock counter is available in Simulation mode only. In Emulation "getClock()" will return 0 always and "setClock()" has no meaning.

Class reference at "Debugger API Class Reference".

3.8.1 ExecutionAPI_v2 changelog:

- "**isRunning**" – new method to identify the state of the core (running or suspended).

Class reference at "Debugger API Class Reference".

3.8.2 Common Usages

- stepCycle – give a single clock to the core.
- stepOver – step over the next instruction (assembly level).
- stepOut - step out of the current function (assembly level).
- sourceStep – step into, run until the next source level instruction.
- sourceStepOver - step to the next source line without stepping into functions.
- sourceStepOut - step out of the current function (source level).
- Go – run the simulation until a breakpoint is reached.
- Run – run the simulation without stopping at breakpoints.
- Run to a specific address.
- Run a number of cycles.
- Stop run – if the simulation is still in execution this will abort execution.
- Reset the simulation / emulation.
- Get current internal clock.
- Get current Program Counter (PC).
- Set core internal clock to a value.
- Set the next execution program counter.
- Check if the application reached its end.

3.9 RegisterAPI

Interface for retreiving information of the core internal registers, getting or changing their values.

The RegisterAPI is completely generic and support any register type of the core, also, retrieving the registers APIs from the core is done either in a generic manner – requesting the number of registers in the core and retrieving their respective RegisterAPI, or by requesting a RegisterAPI if the name of the register is known.

RegisterAPI might have register parts which are also RegisterAPIs by themselves and can be accessed through the "parent".

All Registers are associated to a certain family (accumulator/pointers/general/VCU and so on).

Class reference at "Debugger API Class Reference".

3.9.1 RegisterAPI_v2 changelog:

- "**getRegisterAddress**" – get the address to which this register is mapped to.

Class reference at "Debugger API Class Reference".

3.9.2 Common Usages

- Get register information – name, family, size.
- Get or Set the register value, can be with array of bytes (according to size), or string value.
- Get access to register parts (RegisterAPI).
- Register an *ExternalAccessObserver* – if this is an external register and the user wants to get updated when this register has been accessed (read/write).

3.10 DebugInfoAPI

Interface for all source level debug information.

Can retrieve information on symbols, variables, Assembler / C source files.

Mainly returns an object of VariableInfoAPI.

Class reference at “Debugger API Class Reference”.

3.10.1 Common Usages

- Get Code/Data symbols information (name, address, size, type and so on).
- Get functions information – all functions or specific function.
- Get file information – address from file name & line or file name & line from address.
- Get call stack information – depth, function and so on.
- Get source level variable information (VariableInfoAPI).
- Get current scope local variables information.

3.11 VariableInfoAPI

Interface for retrievieng information on source level variables.

The API is generic for any type of variable (int, char, struct, array, pointers, and so on) in order to distinguish the type it is possible to use its name and number of parts.

For example:

Consider the following struct:

typedef struct T_Person_tag{

int id;

char name[10];

}T_Person;

And an array of that struct:

T_ Person people[2];

To get information on the variable "people":

*VariableInfoAPI *pPeople =*

(DebugInfoApi)pDebugInfo->getVariableInfo<VariableInfoAPI>("people", 0);*

Where 0 represents the current stack frame index.

Querying the *pPeople* for the type of the variable will return "*struct[]*".

Querying the *pPeople* for the type name of the variable will return "*T_ Person_tag*".

In the same manner the user can get information about each part of the structure (members), which are also VariableInfoAPI.

3.11.1 VariableInfoAPI_v2 changelog:

- "**isRegister**" – new method to identify if a variable is binded to a register of the core or resides in memory.

Class reference at "Debugger API Class Reference".

3.11.2 Common Usages

- Get name of the variable requested – as we can request a variableInfoAPI from the DebugInfoAPI according to address.
- Get variable type – int, char, struct, array, pointer and so on
- Get the variable type name – the specific type name (for example, struct name)
- Get the size of variable (in bits).
- Get the address of the variable.

- Get parts information (VariableInfoAPI for each part).
- Get & Set the value of the variable – with byte array.

3.12 MemoryAPI

Interface for all memory spaces information and operation of the core.

Class reference at “Debugger API Class Reference”.

3.12.1 Common Usages

- Get memory space various information – max address, name (for example, “code”/”data”/”io”), cell size, alignment.
- Retrieve information and modify memory configurations, using ConfigAPI and AttributeAPI. Memory attributes for example:
    - Read only
    - Shared
    - Read / Write wait states
    - Is port
- Add a new memory configuration name and modify its attributes.
- Map memory (internal/external) – necessary for read/write from a memory space, the mapped space is mapped according to a memory configuration (either pre-set or user).
- Retrieve information on mapped elements (can be mapped from a loaded binary file).
- Get / Set memory blocks values.
- Connect a memory cell(s) as port (for example, to file or console).
- Download / Upload memory to / from a file in various formats.
- Get / Set the disassembly string in a memory cell (code memory only).
- Get instructions extra information (code memory only) – some instructions have tooltips of extra information. CEVA-XC family ‘vgen’ instructions for example have extra information of what the instruction actually performs as it is decided in runtime only.
- Register an *ExternalAccessObserver* – in case this is an external memory blcok and the user wants to get updated when this block has been accessed (read/write). This is a simple way to simulate external memories when working in ISS (Instruction Set Simulation mode) where the external AXI/APB ports are not simulated.

  In CAS (Cycle Accurate Simulation) it is possible to simulate “external” memory in the internal memory region of the core using the observer as internal memory is not accessed via the external ports.

3.12.2 MemoryAPI_v2 changelog:

- ”**registerExternalAccessObserverIndex**” – same as ”**registerExternalAccessObserver”** but the return value is an index identification which then can be used to unregister the same observer.
- ”**unregisterExternalAccessObserver**” **–** new method to unregister external observer according to it’s index.

Class reference at “Debugger API Class Reference”.

3.13 ResourceAPI

Core resources are exposed through this common interface in a generic manner, meaning they all supply the same information (in table form), the user can use the data as is or parse it to get the actual information.

For example:

CEVA-XC42xx has queue managers and there is a ResourceAPI for showing the top most task of a each of the queues.

A single queue is a ResourceAPI of type *ResourceType_E_Group* it has sub resources, each of type *ResourceType_E_Array* which represents columns in the table – such as:

- Index of the queue
- Enabled / Disabled
- Message type
- Download/Upload
- Internal address
- External address

Looking at an element in those sub resources at an array cell will give us a cell in the table.

Class reference at “Debugger API Class Reference”.

3.13.1 Available Core Resources

- Code cache
- Write buffer
- Queue Manager (XC42xx only)
- Buffer Manager (XC42xx only)

3.14 WatchpointsAPI

Interface for all types of breakpoints:

- Code memory (and source files + lines)
- Data memory (by address with/without condition value)
- Data value match – without a specific address
- Registers
- Clock

Watchpoints are set according to a given structure of information defining what type of watchpoint the user wishes to set with all parameters - *T_WatchpointInfo*.

Watchpoints break the execution of the core (breakpoints), and notify a *WatchpointsObserver* of which watchpoint was reached.

The user must register a *WatchpointsObserver* in order to handle the core break and stop or continue execution (the default is to stop).

Each watchpoint set returns a unique ID to be used as identification when a watchpoint was hit or when getting information of a watchpoint, or canceling one.

Class reference at “Debugger API Class Reference”.

3.15 SignalAPI

Interface for exposing the core internal input/output signals as well as all Memory Sub System (MSS) input/output signals (such as AXI/APB external ports), it is a crucial API when working in an ESL environment where CEVA core is part of a system, all handling of signals are done through the various SignalAPI. Each input or output signal has its own SignalAPI.

There are some SignalAPIs which are actually groups of signals, for example “Input Signals” is a SignalAPI which has sub signals that are the core input signals. In this manner all signals are divided in a tree fashion to groups->sub groups->nodes (*Core Signals* -> *Input Signals* -> *Interrupts* -> *int0*).

SignalObserver gives the option to register an observer (implemented by user) on a certain signal to get notified when its updated (for output signals). This way the user does not have to sample the value of the signal each clock using “getValue” but can do so only when the observer is called. Even with an observer registered it is still possible to get the value each clock.

See also a list of all signals per core:

CEVA-XC323 Signals List

CEVA-XC4210 Signals List

CEVA-XC4500 Signals List

CEVA-TL3210 Signals List

CEVA-TL3211 Signals List

CEVA-TL410 Signals List

CEVA-TL411 Signals List

CEVA-TL420 Signals List

CEVA-TL421 Signals List

CEVA-XM4 Signals List

Class reference at “Debugger API Class Reference”.

3.15.1 SignalAPI_v2 changelog:

- ”**registerUpdateObserverIndex**” – same as ”**registerUpdateObserver”** but the return value is an index identification which then can be used to unregister the same observer.
- ”**unregisterUpdateObserver**” **–** new method to unregister update observer according to it’s index.
- ”**getDestination**” – new method to get the destination type of the signal (input/output).

Class reference at “Debugger API Class Reference”.

3.15.2 Common Usages

- Get signal information – type, name, size.
- Get sub signals SignalAPI.
- Get / Set signal value.
- Register an observer to get notified when signals are updated.

3.16 TraceAPI

Interface for starting or stopping the trace, or for starting stoping the memory profiler.

A trace file can be generated using this API which will record, clock, registers, memory access & instruction information every clock in a binary formatted ".frm" file. The .frm file can be dumped to a text file using the 'cevafrm.exe' utility.

The trace can be started or stopped at any given time in the middle of the simulation.

It is also possible to start / stop the memory profiler. (see memory profiler documentation).

After starting the memory profiler it i possible to load a .prf (profiler setup file) using the CoreAPI function for calling a script file "*callCliScript*" and passing the .prf file name as a parameter.

Class reference at "Debugger API Class Reference".

3.16.1 TraceAPI_v2 changelog:

- "**startProfiling**" – added an argument to set the coff file associated with the profiling session.

Class reference at "Debugger API Class Reference".

3.17 InterruptAPI

Interface for generating simulated interrupts.

Interrups can be raised by toggeling their corresponding SignalAPI, but, can also be generated internally by the debugger using this interface, with options to keep the interrupt request high for a counted number of cycles, or frequency, with/without a delay of when to start toggling and so on.

It also provides information on what interrupts are available to toggling.

It is also possible to cancel the generation of an interrupt.

Class reference at "Debugger API Class Reference".

3.18 FunctionalPort API

Interface for setting Snoop Transaction for the XC4500 Data Cache module.

The ACE protocol has a dedicated snoop channel to maintain the cache coherency.
This interface replaces the use with the AXI Signals for Snoop Transaction.
Using this interface require setting the right AXI signals for the snooping as function parameters.

The function is non-blocking and in return the user immediately gets the response and data (depends on the snoop transaction type).

Note: For more information and for the expected behavior refer to section:
1.68 Snoop Port in the CEVA-XC4500_Arch_Spec_MSS

3.18.1 Common Usages

- Maintain the cache coherency for shareable, cacheable and valid addresses.
- Changing the state bits associated with cache line.
- Getting the Data in the cache line.

3.19 SystemC API

Interface for retrieving SystemC TLM2.0 interfaces of the given.

For more information information see the SystemC example.

4 API Workflows

4.1 Using the cevadbgapi.dll

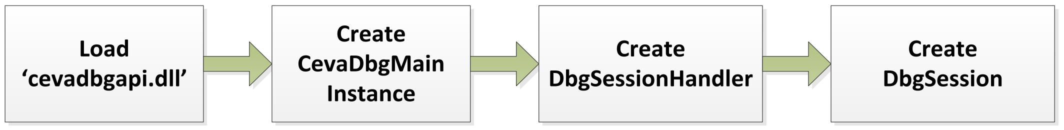
Before using 'cevadbgapi.dll' – make sure to load it from the correct and desired path.

There are 3 exported C functions for the cevadbgapi.dll :

1. 'createNewCevaDbgMain' – creates a new instance of class 'CevaDbgMain' which can then be used to create new debug sessions.
2. 'deleteCevaDbgMain' – deletes an instance of 'CevaDbgMain'.
3. '`getAPIVersion`' – get the version of the debugger & API.

Prior to loading a debug session the user needs to create a CevaDbgMain object which then is used to get an instance of 'DbgSessionHandlerAPI' – which is used to create the DbgSessionAPI instance. Either local or remote (most likely local session is needed).

Starting a DbgSession



Attention:

Always when trying to get an instance of any "API" class, a template reference to that class is needed, for example:

```
DbgSessionAPI  *pDbgSession
                = pSessionHandler->createNewDbgSession<DbgSessionAPI>();

MemoryAPI    *pMemoryAPI
                = pCore->getMemoryAPI<MemoryAPI>("data");
```

4.2 Debug Session

A debug session is the system to be debugged. A session can contain several cores from different types, with shared and non-shared memory.

4.2.1 Creating a new session

DbgSessionAPI is the class that represents a debug session. Each instance of DbgSessionAPI is an independent entity that is not connected to other sessions.

A new instance of DbgSessionAPI can be created by:

1. Load the ‘cevadbgapi.dll’
2. Get the function address for ‘createNewCevaDbgMain’
3. Create a new CevaDbgMain object
4. Create a new DbgSessionHandler object by calling createNewDbgSessionHandler() in CevaDbgMain class.
5. Create a new DbgSessionAPI object by calling createNewDbgSession<DbgSessionAPI>() in DbgSessionHandler class.

4.2.2 Loading a new core to the session

The instance of DbgSessionAPI returned by createNewDbgSession() is an empty session. To start working with the session, the user first needs to load a new core.

To find out which cores are available in the session (according to the cores licensed for the user) there are 2 functions DbgSessionAPI – getNumOfSupportedCores(), and getSupportedCoreInfo(). After the user finds the core he wants to load.

The core can be loaded by calling loadNewCore() with either the index of the selected core or the name of the sub-core to load, and that the core is licensed. loadNewCore returns a new instance of CoreAPI that is used to perform operation on this core.

```
CoreAPI *pCoreAPI = pDbgSession->loadNewCore<CoreAPI>("CEVA-XC4210");
```

A loaded core can be unloaded from the session, by calling unloadCore(). The CoreAPI is passed by pointer, and is reset to NULL after the core is unloaded.

4.2.3 Notifications observer for the Debug Session

A NotificationObserver can be registered on the DbgSession object to get callbacks when there are Session related events (such as errors, messages etc.).

It is important to register an observer before performing additional operations with the DbgSession although there is a ‘getLastError()’ method - notification events might have more information on what went wrong.

To register such observer:

```
pDbgSession->registerNotificationObserver(pObs);
```

(to unregister use the ‘unregisterNotificationsObserver’ method).

4.3 Core Configuration

Using ConfigAPI & AttributeAPI it is possible to configure the core before starting to debug and execute it.

There are two methods of changing core attributes:

Method 1:

Loading an XML Configuration file using ConfigAPI `loadAttributes` method. See ConfigAPI.

Method 2:

The many attributes of the core can be requested by the ConfigAPI, examined and modified in a generic matter.

Each attribute is of a specific type (list, string, number and so on), and the values it can receive are according to its type, this means that if the attribute is a list, its possibilities (1,2,3…getNumOfPossibleValues) are matched to its possible values.

For example the attribute of "Execution Mode":

Type: AttribDisplay_E_List

Num Of Possiblities: 3

Possibilities:

0 – Instruction Set Mode

1 – Cycle Accurate Mode

2 – Emulation Mode

To change the execution mode to cycle accurate:

```
// iterate over core attributes to find the correct attribute
uint64 numOfAttrib = pCoreConfig->getNumOfAttributes();

AttributeAPI *pExecMode = NULL;

for(int i = 0; i<numOfAttrib; i++)
{
	AttributeAPI *pAttrib = pCoreConfig-
>getAttribute<AttributeAPI>(i);
	const char *name = pAttrib->getAttributeName();

	if(strstr(name, "Execution Mode") != NULL)
	{
		pExecMode = pAttrib;
		break;
	}
```

```
// execution mode is not a straight forward number – we need the value
in the list that corresponds to the execution mode we want
if(pExecMode->getAttributeType() == AttribDisplay_E_List)
{
      uint64 numOfPossible = pExecMode->getNumOfPossibleValues();

      for(int i = 0; i<numOfPossible; i++)
      {
            const char *name = pExecMode->getValueName(i);

            if(strstr(name, "Pipeline Mode") != NULL)
            {
                  // found the value - set the mode according to index
                  pExecMode->setValue(i);
                  break;
            }
      }
}
```

4.4 System Execution and Remote Debug

4.4.1 Setting up the system

A system can be set up in any C++ development environment, such as Visual Studio, which interacts with the Debugger API in such way to load any type and amount of CEVA cores, the API calls can be integrated into any existing systems to create a debug environment for the CEVA cores inside the system.

A few points to consider before setting up the system:

1. The simulation mode of the cores – ISS / CAS, and if there be full pin level simulation in that case the simulation needs to run in ESL mode (set through dedicated AttributeAPI "ESL Mode").
2. How the cores / peripherals are connected? – the core can communicate to other cores via signals (SignalAPI) or via external access observer (in ISS only).

   This system connection scheme is best to be decided before hand and set up in code correctly to get correct system simulation.

The system connection is decided according to the system execution mode:

4.4.1.1 ISS system

In Instruction Set Simulation mode, external access to memory is done through the "*ExternalAccessObserver*" which is a user created class (inheriting from the APIs "ExternalAccessObserver" class) that can be registered on any memory region through the MemoryAPI. This will cause the observer to be called on any read/write on the observed region.

SignalAPI can also be used (in ESL mode only) to toggle externaly any input signals and get values of output signals from the core, such as interrupts and their corresponding acknowledge output signals.

Note: In ISS mode the external ports (AXI/APB/and so on) are not simulated.

4.4.1.2 CAS system

In Cycle Accurate Simulation mode, external access to memory is done through the dedicated external ports which are specific for each core (AXI/APB/etc.).

To connect to these external ports (Master/Slave) the user needs to set the ESL Mode to "On". This external ports which are specific for each core (AXI/APB/and so on).

It is possible at any given cycle to get/set value of any SignalAPI, but, it is also possible to register "*SignalObserver*" on any SignalAPI to get an event notification of when an observed signal gets updated.

The core signals (interrupts/boot/and so on) are also supported in CAS mode with ESL mode set to "On"

4.4.2 The System Execution

As mentioned in SystemExecution, the user can overload the default SystemExecution of the DbgSessionAPI, in that case the user controls the execution of each core (by calling its needed methods in ExecutionAPI) and can change the execution ratio for specific cores for example.

In ESL mode, the user can toggle the input signals of the core or MSS, in case of input signals they should be toggled in the pre-cycle execution stage, in case of output signals they should be sampled in the post-cycle execution stage.

If not overloading the default “SystemExecution” the user has the option to register the “SystemExecutionWrap” class and add implementation to the “preExecute()” / “postExecute()” methods which will be called by the “SystemExecution”.

SignalObserver’s *update()* method gets called in the middle of core execution, in this stage the user can sample other signals that might have changed value at the same time, for example, according to AXI3 protocol the “*arvalid*” and “*araddr*” are toggled at the same clock, so setting an observer on the arvalid signal is enough to sample at the same time the araddr signal.

Without the usage of observers the user can always sample signal values in post-cycle execution stage.

4.4.3 Remote debug using the Ceva-Toolbox IDE

4.4.3.1 General

It is possible to remote debug CEVA cores in a system “built” using the DBG API to get full source level debugging, including registers windows, memory views, breakpoints, disassembly and C/C++ source debug.

This is all done by connecting to a remote session server set up in the DBG API from the CEVA-Toolbox IDE.

4.4.3.2 Setting up a remote session server

In order to start a remote session server, a dedicated API call is present in the DbgSessionAPI:

```
startRemoteDebugServer(const char *pSystemName = NULL, int port = 0)
```

Note: This API call will start a **new OS thread** which will be listening for TCP connections from a remote debugger.

After this API call, the Main thread of the user’s program will continue, hence if the user wants to be able to connect from a remote debugger and control the simulation and debug from there, the Main thread of the application should be blocked, by mutex/scanf/endless loop or any other means.

4.4.3.4 Understanding the Remote Debugger Execution of The System

It is possible to control the simulation of the system either from the Visual Studio project using the Debugger API or from the external debugger.

Note: when executing the system from the remote debugger, the extra thread opened for listening to remote connections is the thread that will call either the “SystemExecution::execute()” or “SystemExecution::run()” methods.

If the user wishes to execute from a remote debugger, the user’s Main thread process should be halted in order to eliminate conflicts between the two threads as they will be sharing the same resources.

If the user wishes to execute from the system execution using the DBG API he can do so and the remote debugger will get refreshed automatically as there is a two-way connection between the debugged cores and the debugger.