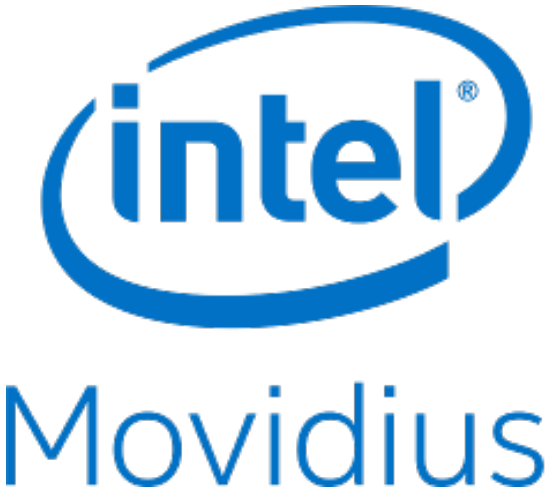


moviSimulator

Manual

Version 00.89.0 / 2018-03-30

Table of Contents

Copyright and Proprietary Information Notice

Movidius Ltd.
1730 South El Camino Real, Suite 200
San Mateo, CA 94402
http://www.movidius.com/

1. Introduction

MoviSim is a configurable simulator designed for heterogeneous multiprocessor systems. Its purpose is to simulate architectures, allowing them to execute real software applications rather than execution traces. The simulator allows the user to implement external models and to import them as dynamic libraries (via DLL files). These libraries are specified in an XML configuration file.

Other features available in this simulator are:

- Loading and execution of code for the processors in the system.
- Debugging facilities through a client-server interface which connects the simulator to a debugger.
- Profiling facilities: bus traffic analysis, memory access analysis, execution time/waiting time analysis, cache hit/miss analysis, register access analysis, function call analysis, code and data labels access profiling, assembly line profiling (some of them only available to moviDebug2).
- Data logging for later inspection.
- Using standalone computer vision models in PC simulation environment (Warp and Stereo support)

MoviSim can run in the following modes:

- As an independent application.
- As a separate application communicating with moviDebug2 in client-server mode using a TCP/IP socket for commands and messages.
- As a source of libraries that can be loaded dynamically inside another application for PC simulations (Warp and Stereo support)

1.1. Conventions used in this document

The moviSim parameters or source code in the examples in the present document are written in the Courier New font.

The words enclosed by < and > are not actually keywords, but values: <parameter> is one value specified as the parameter.

The { } denote a series of possible values, | separates the values of the series.

The [ ] denote an optional parameter.

The parameters in the moviSim command line can be separated by whitespaces (spaces, tabs).

2. Usage Examples

The simulator can be used in two separate work flows, with different approaches to how code is loaded and executed. These work flows are described as:

- Standalone simulator
- Client-server mode with debugger

2.1. Directory structure

The following directory and file structure is applied to both use cases. The simulator and other tools are located either in the `Tools` folder or in the `moviSim` folder (for releases numbered **xx.x4.xx.xx**).

[Tools]	- Tools directory
[common]	
[moviSim]	
[architectures]	- Architectures directory - contains XMLs
myriad.xml	- Myriad architecture XML
...	- Other XMLs
[win32]	- Folder containing Win32 specific files
[bin]	
moviSim.exe	- Windows simulator executable
moviDisplay.exe	- Windows executable for runtime averages display
moviAsm.exe	
moviLink.exe	
moviCompile.exe	
[models]	- Folder containing model libraries
shavesDll.dll	
leonDll.dll	
timersDll.dll	
...	
[modelLibrary]	- Library for building new model DLLs
moviIfLib.lib	- Interface library for building model DLLs
[lib]	
[linux32]	- Folder containing Linux32 specific files (hierarchy is the same as for Win32)
[bin]	
moviSim.elf	- Linux simulator executable
moviDisplay.elf	- Linux executable for runtime averages display
...	
[models]	
...	- Model libraries (same as for win32)
[modelLibrary]	- Library for building new model DLLs
moviIfLib.a	- Linux version of Interface library

[lib]	
[examples]	- Examples folder

NOTE: Some releases may have a restricted subset of the files and directories described here – the `modelLibrary` related files may not be present.

2.2. Standalone simulator

The `moviSim` simulator is compiled as a standalone application (`moviSim.exe`) for use in standalone mode. In this mode, the user runs the simulator with specific command line arguments (described in chapter 5). The arguments allow the configuration of the system, loading of code and data into memories, and several other profiling and logging features. Also, the runtime averages display can be enabled from a `commandLine.txt` file or the command line arguments (which have priority over those provided in the `commandLine.txt` file).

2.3. Client – server mode

For this workflow, the simulator is started without an `.elf` file specified with the `–l` command line argument. The debugger connects to the simulator via one TCP socket and implements a series of commands capable of debugging applications that are running in `moviSim`.

3. System Requirements

3.1. Windows

The simulator has been tested on systems running Windows XP or later.

3.2. Linux

The simulator has been tested on the following Linux distributions: CentOS 6.3, Ubuntu 64-bit, Red Hat 4.

4. MoviSim Components

4.1. Simulator core

The simulator core is a dynamic simulation environment, designed to allow the integration of different modules in order to construct a complete platform which can be used in conjunction with user applications or benchmarks to evaluate platform performance and cross-platform performance portability.

4.2. XML parser

The XML parser allows the system configuration to be loaded from a user-defined XML file. The format of the XML file is defined separately in chapter 6.1.

4.3. Generic models

The simulator contains an extensive range of configurable modules which can be used to construct target systems for performance-portability evaluation. These are configurable modules that are found in most SoCs, such as buses, bus bridges, memories, caches, etc.

4.4. Architecture specific models

A library of specialized modules can be defined by the user, specific to particular heterogeneous processor target architectures to be used for performance-portability experiments. These models are particular to a specific architecture. They include processor models (SPARC, SHAVE, ARM, PowerPC), custom memory models (CMX memory), interrupt controllers, UART, JTAG and other peripherals. The list of specific models is not exhaustive, as user modules can be created and added as dynamic libraries, using the simulator's external interface.

5. MoviSim Command Line Arguments

Table 1. `moviSim` command line options

Switch	Description
-h[elp]	Display the help screen for moviSim
-version	Display version string and exit
-useXML:<xmlFileName> -usexml:<xmlFileName>	Specify XML file
-cv:<architectureName>	Specify the architecture to be simulated
-q	Specify quiet mode
-leon -Leon	Specify Leon only mode
-l:<address>:<fileName.bin> -l:<targetName>:<fileName.elf>	Specify file to load
-s:<address>:<size>:<fileName>	Specify file to save
-nodasm	Disable disassembly
-nodma	Inhibit display of DMA messages
-darw -dawr	Enable display of register/memory reads/writes
-p -P	Disable register port-clash checking
-output:<dirName>	Specify output directory
-enBusLogs -enbuslogs -enCoreLogs -encorelogs	Enable bus / processor logging
-tcpip[:<portNumber>]	Enables client-server mode using default or specified port
-n	Enables sensitive mode
-err[:<fileName>]	Enables errors display at simulation end with optional error logging
-o(<name1>[<name2>...]):<fileName> -o{<name1>[<name2>...]}:<fileName> -o<name>:<fileName>	Enables logging of combined output of different models
-timeout:<numberOfSeconds>	Enable simulation timeout after specified time
-memInitVal:<hexValue>	Specify a value that is used to initialize memories
-j[:<numberOfThreads>]	Enable multi-threaded simulation
-maxLeonInstructions:<count>	Enable simulation termination after specified number of instructions
-simLevel:<option>	Configure simulation level

`MoviSim` allows the use of the command line arguments described in the table above. A more detailed description of each argument is provided in the following sub-chapters. There are more command line switches which might be referred to in the current document, but they are still in testing phase or might be subject to change. Please do not use switches other than the ones presented in the above table in production code.

5.1. Help

Syntax

`-h[elp]`

Description

Displays the help screen containing all possible command line arguments, their syntax and a short description for each. It can be used together with other switches, causing the displaying of the help screen at the start of execution.

NOTE:	Using <–q> together with <–h[elp]> suppresses the help screen display.

5.2. Display version

Syntax

`-version`

Description

Displays the version string, with the format <aa.bb.cc.dd> and causes moviSim to exit if it is used as standalone. When it is used together with other arguments, the version string is displayed and execution continues to parse the other command line arguments.

5.3. Specify verbosity level

Syntax

`-v:<level>`
`-verbosity:<level>`

Description

Specifies the verbosity level of the printed messages. There are 5 verbosity levels (0-4), as described in the table below.

Level	Enables display of:
0	No messages
1	- executed instructions
2	- executed instructions - port clash detection messages
3	- executed instructions - port clash detection messages - register reads and writes - memory reads and writes - DMA messages

4	- executed instructions - port clash detection messages - register reads and writes - memory reads and writes - DMA messages - stalls - Debug Unit register access (Shave processors)

NOTE: Arguments like –nodasm, -darw, -p etc. specified prior to the –v argument will be ignored as the same internal flags are used. Specifying these arguments after -v will override the internal flags and may enable display of messages additional to the specified verbosity.

5.4. Use XML file

Syntax

```
-useXML:<xmlFileName>
-usexml:<xmlFileName>
```

Description

Defines a specific XML file for the simulator to use. The XML file describes the architecture and is used for system configuration. The user should provide only the file name without the path. If this argument is not specified, the simulator uses the myriad1.xml file.

5.5. Specify architecture

Syntax

```
-cv:<architectureName>
```

Description

Defines a specific architecture for the simulator to use. The XML file that describes the architecture must exist and its name must be of the form architectureName.xml otherwise it cannot be loaded and an error will be generated. If this argument is not specified, the simulator uses the myriad1.xml file by default.

5.6. Quiet mode

Syntax

```
-q
```

Description

This argument forces the simulator to suppress the messages that are displayed about the code execution. Note that only the console display of the message is suppressed, not any disassembly

actions needed to compose the messages (the use of –nodasm is recommended to inhibit internal disassembly also if simulation speed-up is desired and no logging is needed). This option also disables the help display, so using –h[elp] has no effect.

5.7. LEON only mode

Syntax

```
-leon
-Leon
```

Description

This argument forces the simulator to suppress the search for other processor entry points if there is an entry point for the Leon processor in the .elf file specified for loading. It disables the execution starting for the other processors, even if they have an entry point set. This option is useful when the user wants to manage the other processors using the Leon processor (even if the application that gets loaded for each of the other processors has an entry point and could be run independently). It should not be used on the Myriad architecture if the user wants to load and start execution of SHAVE code without the LEON control code.

5.8. Load

Syntax

```
-l:<address>:<fileName.bin>
-l:<targetName>:<fileName.elf>
```

Description

This argument is used for loading code or data into the memory or register file(s) from a specified file. The user can load data and code binaries. The same argument can also be used for restoring or fast setting of register file content which may be saved from a previous run of the code. The supported file extensions include the standard .elf format and raw binary data (.bin). In order to correctly load an ELF file, the targetName of the processor for which this file was compiled and linked must be specified, otherwise the entry point will not be set correctly (ELF format does not include target information, so the target processor has to be specified based on the <ShortName> which can be found in the XML information). The number of shave cores is architecture dependent (12 cores for MA2100, MA2x5x; 16 cores for MA248x). Target names for the cores are described in the table below:

Core	Target Names
LEON OS	LOS
LEON RT	LRT
SHAVE 0	S0
SHAVE 1	S1

...	...
SHAVE n	Sn

When loading binary data, the starting address in memory must be specified. The simulator tries to load the entire contents of the binary file at the specified address. The binary file can have any extension (except `.elf`).

NOTE: The address for the binary file load for the `.elf` file load must be in hexadecimal encoding (`0x···`). Otherwise an error is generated.

5.9. Save

Syntax

`-s:<address>:<size>:<fileName>`

Description

This argument is used to save the content of registers or a memory area to a file. The `<address>` can be represented in hexadecimal (e.g. `0x10000000`) and `<size>` can be represented in decimal (e.g. 1024) or hexadecimal (0x400). At the end of the simulation, `moviSim` transfers `<size>` bytes starting with the one at address `<address>` into the file specified by `<fileName>`. The file name can have any extension and is saved in raw data format.

NOTE: The address that is specified must be in hexadecimal format (0x...). The size is allowed to be either a hexadecimal or decimal value.

5.10. Architecture marker

Syntax

`-a:<architectureId>`

Description

This argument is used when the system topology (described in `XML`) contains more than one `<Architecture>` tag. This command line argument applies to other arguments that are given after it. Using this argument before a load or save argument, for instance, instructs the simulator to execute that specific command on the architecture specified by its ID. The architecture IDs are described in the architecture description file (`XML`). If no architecture argument is explicitly specified, all switches apply to the topmost architecture in the system, without considering the sub-architectures or any subsequent architectures. For example, considering that `<myriadCluster.xml>` describes an architecture that instantiates several myriad chips, the following command line can be used:

```
moviSim.exe –usexml:myriadCluster.xml –a:0x01 –l:LOS:file1.elf
–s:0x10000000:1000:out1.bin –a:0x2 –l:LOS:file2.elf –s:0x40000000:2000:out2.bin
```

This instructs the simulator to: - use the `myriadCluster.xml` file for configuration

- load the file `file1.elf` onto the architecture with ID 0x1
- save some data from the same architecture to the `out1.bin` file
- load the file `file2.elf` onto the architecture with ID 0x2
- save some data from architecture 0x2 into the `out2.bin` file

WARNING:	In multiple architecture systems, components in different architectures can have the same names and device identifiers. Using this argument with an architecture ID that does not exist in the `XML` will result in erroneous behavior.

5.11. No disassembly

Syntax

```
-nodasm
```

Description

If this parameter is used, no disassembly code is printed for the executing processors. This causes a significant increase in code execution speed, as running the disassembler is quite taxing due to string processing. The use of this command line argument is recommended at all times, unless the user needs to analyze the code execution instruction by instruction, or to create processor execution logs.

5.12. Enable display of stalls

Syntax

```
-enStalls
-enstalls
```

Description

This argument enables the displaying and logging of stalls according to different stall sources. It can also be used together with `–v` to enable stall display on low verbosity levels.

5.13. Enable register and memory reads/writes

Syntax

```
-darw
-dawr
```

Description

This argument enables the displaying and logging of register reads/writes and memory accesses made by processing elements. The model must have the necessary implementation in order to use this properly. The enable flags are members of the `MsimModelArguments` structure described in chapter 6.

NOTE:	When not expressly checking register/memory reads/writes, the use of this switch should be avoided as printing large amounts of data to the screen affects simulation speed.

5.14. Disable register read/write port checking

Syntax

```
-p
-P
```

Description

Disables the detection of register read/write port clashes. The processor model must have the necessary implementation in order for this command line switch to have any effect. This gets translated into a value stored in the `readPortCheck` and `writePortCheck` members of the `MsimModelArguments` structure that gets passed to each processing model. The contents of the structure are described in detail in section 6.

NOTE:	When not expressly checking for port clashes, the use of this switch is highly recommended as it increases simulation speed.

5.15. Output directory

Syntax

```
-output:<directory>
```

Description

Specifies the directory in which the profiling and logging information is saved. If it is not specified, the simulator logs all requested information into the current working directory. If the specified directory does not exist, it is created.

5.16. Enable logs

Syntax

```
-enBusLogs
-enbuslogs
-enCoreLogs
-encorelog
```

Description

Log all bus transaction into files. Log all `PROCESSOR` messages into files. A log file is created for each bus or processor. For each processor, the simulator creates a `.out` text file. For each bus, the simulator creates a `.log` file. The file name is composed from `Bus_`/`Core_`, followed by the bus or processor name and the respective extension. If there are more than one architectures in the system, the log files are prefixed with `Axxxxxxxx_`, where `xxxxxxxx` represents the architecture ID specified in the `XML` file. This is done in order to make sure that each processor and bus has its own log file, even if names across separate architectures match (different instances of the same SoC for instance). Using `–nodasm` together with this command line switch causes processor instruction disassembly not to be logged. Also, if `–darw` is not specified, no register accesses are logged. These arguments cause the simulator to create all the logs for all available buses or processors. In order to specify only certain logs to be created, or to analyze synchronization be-tween different components, the user should consider the combined output file argument `-o` (see section 5.34).

NOTE:	Extensive logging impacts simulation speed.

5.17. TCP/IP communication

Syntax

```
-tcpip
-tcpip:<port>
```

Description

This command line argument puts the simulator in a client-server mode. This mode is used for communication with the `Movidius Debugger` and `Movidius Test Environment`. Optionally, a port number can be specified for the `TCP` connection. The simulator goes into the client-server mode by default (without specifying this argument), unless some load argument is present.

5.18. Sensitive mode

Syntax

```
-n
```

Description

Causes the simulator to stop execution and exit when the first error is detected during execution. The argument also enables the display of the error before exiting.

5.19. Error logging control

Syntax

```
-err
-err:<fileName>
```

Description

When used as `–err`, this argument instructs `moviSim` to display all errors at the end of the simulation. When a file is specified, all the errors that occurred during moviSim execution are logged to the file specified by `<fileName>`.

5.20. Transaction logging control

Syntax

```
-translog:<filename>
-transLog:<filename>
```

Description

Logs all transactions from the simulation into the file specified. The transactions have a standard format in the following form: `BTX:xxxxxxxxxx,mmmm,nnnn,iii,aaaaaaaaaa,sss,tttt`

Where:

- `xxxxxxxxxx` is the cycle number
- `mmmm` is master ID of the requesting component (`XML` value)
- `nnnn` is sub master Id (`LSU0 – 0x1`, `LSU1 – 0x2`, `IDC – 0x3`, `L1C – 0x4`, `DMA – 0x5`, `TMU – 0x6`, `NO_SUBMASTER – 0x0`)
- `iii` is the transaction ID and it represents the burst ID of the request (0 if not a burst)
- `aaaaaaaa` is the hex representation of the form `0xA0000000` of the address (will be extended to 64 bits in the future)
- `sss` is transaction size in bytes
- `tttt` is transaction type (read – `0xFE`, write – `0xFF`, read return – `0xFD`)

5.21. Camera model input file

Syntax

`-camera:<deviceId>:<fileName>`

Description

This argument specifies the input file for the camera interface. The camera model is specific to the `Myriad` architecture. Selection of different cameras is done via the `<deviceId>`, which is specified in the `XML` architecture description. The camera model needs to be configured before use. Its utilization is described in the `Myriad` specification.

5.22. LCD model output file

Syntax

`-lcd:<deviceId>:<fileName>`

Description

The argument specifies the output file for the `LCD` interface. The deviceId is used to differentiate between different `LCD` models specified in the `XML` architecture description file. The `LCD` model is specific to the `Myriad` architecture. The `LCD` model needs to be configured before use. Its utilization is described in the `Myriad` specification.

5.23. Profiling

Syntax

`-prof`

Description

This argument enables the logging of profiling information into a folder called `Profiling`. Optionally, the file name can be specified in the argument.

The simulator offers profiling information from the simulator by calculating the following:

- Execution cycles
- Stall cycles
- Number of executed instructions
- Number of reads/ writes:
    - On 8/16/32/64/128 bits
- Number of memory subsystem reads/writes for the processor cores:
    - On 8/16/32/64/128 bits
- Number of reads/writes done by the SHAVE DMA engine
- Number of reads/writes to each SHAVE register
- Cache statistics
- Power consumption estimation

5.24. Shave DCU register accesses

Syntax

```
-ddrw
```

Description

Enables the displaying and logging of `SHAVE` registers (`IRF`/`SRF`/`VRF`) reads/writes done through the slave interface.

5.25. Read before write check

Syntax

```
-enrbwcheck
-enRBWCheck
```

Description

Enables the simulator to check for reading uninitialized memory locations.

5.26. Output files flush interval

Syntax

```
-flushStep:<step>
-flushstep:<step>
```

Description

Enables the simulator output file flushing after <step> number of bytes written to a file stream. The default flush size is 1MB. By specifying a different value, the user can peek into the logs early during the simulation. For example, a flush step of 2048 would trigger a write to file every 2kB of file size written to the file stream. This feature is extremely useful when logging instruction execution, as the user can periodically peek into the file being logged with a text file viewer and examine the status of execution. Without it, there is no guarantee that the OS flushes the file stream often enough.

5.27. Clock dividers enable

Syntax

```
-enClkDiv
-enclkdiv
```

Description

When using this argument, the base clock divider is activated. For each component in the system, the base clock is divided by the value specified in the XML using the <ClockDivider> tag. The default value for the divider is set to 1. Other values must be expressly specified in the XML file. Even if present, the <ClockDivider> tag is ignored if this command line argument is not used.

5.28. Runtime averages enable

Syntax

```
-enAverages
```

Description

This causes the simulator to enable runtime averages calculation for elapsed clock cycles, average stall cycles, average power consumed and average bus bandwidth consumption. The averages are calculated each 1000 cycles by default. Stall and energy averages are calculated only for active processors, so once the processors get halted (the execution is completed), the last calculated value is retained. However, using this switch has some impact on overall simulator performance.

5.29. Runtime averages sampling step

Syntax

```
-samplingStep:<step>
-samplingstep:<step>
```

Description

This command line switch causes the simulator to display runtime averages for elapsed clock cycles, average stall cycles, average power consumed and average bus bandwidth consumption at the clock cycle intervals specified by <step>. For instance, using the argument: -samplingStep:2000 causes the sampling and display of runtime averages to be done at 2000 cycles intervals.

5.30. Logging to specified EVCD file

Syntax

`-evcd:<evcdFileName>`

Description

This argument is used to enable the logging of data in the EVCD format to the file specified by <evcdFilename>. Currently, the feature is only for the SHAVE processors – the simulator logs the following information: IRF/SRF/VRF changes, IP changes, SHAVE stall signals and clock enable signal, bus bandwidth average and average power consumed. It is based on the messages that are sent by the simulator to the messaging center, so -darw should be used also in order to save changes made to the IRF/SRF/VRF registers. Also, –nodasm disables the logging of IP changes.

5.31. Enable EVCD signal logging

Syntax

`-evcdAdd:<pattern|"all">`

Description

Enables all EVCD signals with names matching the specified pattern. The keyword all enables all available signals.

Example

-evcdAdd:SHAVE0 enables all signals that contain a SHAVE0 substring in their name.

NOTE:	Command line arguments are acted upon in the order they are parsed, so argument order is important.

5.32. Disable EVCD signal logging

Syntax

`-evcdRemove:<pattern|"all">`

Description

Disables all EVCD signals with names matching the specified pattern. The keyword all disables all available signals.

Example

-evcdDelete:SHAVE0.VRF disables all signals that contain a SHAVE0.VRF substring in their name.

NOTE:	Command line arguments are acted upon in the order they are parsed, so argument order is important.

5.33. List available EVCD signals

Syntax

`-evcdSig`

Description

This argument displays all available EVCD signal names and their state (disabled by default).

Example

`-evcdAdd:SHAVE0 –evcdRemove.SHAVE0.VRF -evcdSig` adds all SHAVE0 signals to EVCD logging, then removes the VRF signals and then displays the list of signals.

NOTE:	Command line arguments are acted upon in the order they are parsed, so argument order is important.

5.34. Specify combined output files

Syntax

```
-o(<name1><name2>…):<fileName>
-o{<name1><name2>…}:<fileName>
-o<name>:<fileName>
```

Description

This argument allows grouping of logs from different components into a single log file, allowing the user to investigate synchronization between components. It can also be used to create a log of a single processing element. In that case, it is not mandatory to use parentheses or curly parentheses (`-oSHAVE0:<fileName>` is also accepted). This argument is affected by the `–a` argument that precedes it (if any). The component names that can be used with this command line option can be found in the `.xml` file describing the architecture at the `<Name>` tag for every component. The most commonly used are:

- SHAVE0 – SHAVEn, where n is the number of SHAVE processors – 1
- LEON
- UART0

Also, the following additional names which enable special functions are available:

- GLOBAL outputs all global simulator messages to the log file
- ERROR outputs all simulator error messages for the entire system into the log file

Examples

`-o(SHAVE1+SHAVE2):shave1_2.log` instructs the simulator to log all messages from SHAVE1 and SHAVE2 into the `shave1_2.log` file.

`-oSHAVE0:shave0.log` instructs the simulator to log all messages coming from SHAVE0 into the specified file.

`-o(SHAVE1+LEON+ERROR):file.txt` adds all messages from SHAVE1, Leon and all error messages from the

entire system to the log file.

`-o{SHAVE0+UART0}:file.log` instructs the simulator to log all messages coming from `SHAVE0` and all `Uart0` messages (coming usually from printf calls inside the code) into the specified file.

NOTE:	In `Linux` machines, the terminal has a special meaning for parentheses (`(` `)`), so the user should use either the curly parentheses (`{` `}`) (e.g. `-o{SHAVE1+SHAVE2}:shave1_2.log`) or the `\` character preceding the parentheses (e.g. `-o\(SHAVE1+SHAVE2\):shave1_2.log`).

5.35. Inhibit display of DMA transfer messages

Syntax

`-nodma`

Description

Inhibits display of all `DMA` transfer messages.

NOTE:	By default, the simulator shows slice `DMA` transfer messages. `DMA` refers to the slice `DMA` component.

5.36. Enable simulation timeout

Syntax

`-timeout:<numberOfSeconds>`

Description

Enable simulation timeout after the specified number of seconds has elapsed. If this is not specified, the simulation runs until all processors go into a halted state.

NOTE:	This switch has no effect when running in client-server mode.

5.37. Enable simulation termination after N instructions

Syntax

`-maxLeonInstructions:<count>`

Description

Enable simulation termination after the specified number of instructions have been executed by the Leon processor. If this is not specified, the simulation runs until all processors go into a halted state.

NOTE:	This switch has no effect when running in client-server mode.

5.38. Specify memory initialization value

Syntax

```
-memInitval:<hexValue>
```

Description

Specify the value to be used in order to initialize all memories in the simulated architecture. For instance, using `-memInitVal:0x00` will initialize all memories to 0.

NOTE:	The `<hexValue>` needs to be a hexadecimal number of the form `0xHH`. Only 2 hexadecimal characters are considered, as initialization is done at a byte level.

5.39. Enable multithreaded simulation

Syntax

```
-j
-j:<numberOfThreads>
```

Description

Enables the multithreaded execution of the simulation. If the `<numberOfThreads>` parameter is specified, the simulation will use the specified number of threads, otherwise it will default to 4 threads for simulation.

NOTE:	The simulation speed improvement can be best observed if also `–q` and `–nodasm` arguments are used. Otherwise, the synchronization of the messages display coming from different threads will cancel most of the speed gains.

5.40. Enable timeout on Leon number of instructions

Syntax

```
-maxLeonInstructions:<count>
```

Description

Sets a timeout that triggers when a specific number of executed instructions is reached. The countdown is counting the number of instructions executed by the Leon processor (`LeonOS` on `Myriad 2`).

5.41. Set simulation level

Syntax

```
-simLevel:<option>
```

Options available for simulation level:

`ca`	Enable `moviSim` in cycle accurate mode.

`fast`	Enable `moviSim` in fast model mode.

Description

There are two simulation levels available for `moviSim`: cycle accurate and fast model mode. By default the simulator will start in cycle accurate mode and it is not required for the user to input this parameter. To activate the fast model mode the user will be required to input `-simLevel:fast` as an input command line parameter. In cycle accurate mode the simulator send transaction through simulated buses between components based on architecture description from the `XML` file, and in fast model mode this layer is abstracted and direct communication between components is acieved.

NOTE:	The fast simulation is a work in progress, some corner cases may be not tested yet.

6. Architecture Specification

The simulated architecture can be constructed (and configured) via XML files. The simulator searches for the architecture XML file in the moviConfig directory. In order to allow more flexibility, the user can also use sub-XMLs for the configuration of a specific user-defined component, provided as an external .dll (using the <XmlFile> tag). The user should place these sub-XMLs into a folder inside the moviConfig folder for generalized approach purposes. This is not mandatory, as the XML tag and value are passed to the model library and can be specified at will. It is the same for the other tags inside the <Configuration> tag of a model.

6.1. XML files

6.1.1. MoviSim XML specification

The XML specification file is based on the classic XML description language and uses a number of predefined tags and keywords that are described in later sections of this document.

The XML specification file allows moviSim to be fully configured for heterogeneous multiprocessor systems. The intention is to provide a means of simulating the performance of prototype hardware running real software long before the actual hardware is available. In this way, performance portability can be evaluated at the hardware platform design stage. The ability to simulate the performance of multiple hardware configurations allows performance trade-offs to be made in order to arrive at optimal performance/power/cost for a particular product. These trade-offs would include the number and type of processors to be used, number and type of internal memories, width of buses and ports, the internal bus/NoC (Network-on-Chip) structure and of course the width and type of external memory and peripheral interfaces to support particular requirements. Rather than building a large library of components up-front, the simulator provides a limited, but complete set of building blocks from which other components of arbitrary complexity can be built. This allows the simulator to be built quickly, to be optimized for performance, and to be easily maintained without introducing huge support requirements.

In the future, the XML is generated by a User Interface which contains all the models, features and configurations required. The same interface provides means for displaying graphs from the gathered profiling information and provide analysis tools for the developer.

6.1.2. XML specification file overview

The XML file is composed as a hierarchy of <Architecture> tags, which contain either other <Architecture> tags and/or model descriptions. A number of generic components can be specified in the XML file:

- Generic bus model (AMBA/AXI based)
- Generic memory model
- Generic cache model
- Generic Bus bridge model
- Generic Signal model

There are several specific components that need to be considered:

- CPUs (SHAVE, SPARC, ARM, PowerPC etc.)

- Combinational Memory (MYRIAD/SHAVE CMX memory subsystem)

Each component is described by a numerical identifier (device ID) which has to be different for each device within the respective architecture tag. The same device ID is allowed only within different <Architecture> tags.

6.2. Configurable elements

The architecture can be specified in an XML specification file, using a number of configurable elements.
The XML file structure contains an optional tag that specifies which XML version was used. The whole architecture description is placed into a tag named <Architecture>.
A bus section is placed at the beginning of the <Architecture> element. This section contains all the buses defined in the current architecture. After the bus section, all the other structural elements follow.
The <Architecture> tag contents end with the declaration of the signals which connect the modules defined above.

```
<? xml version="1.0" encoding="US-ASCII" standalone='yes'?>
<Architecture name="MYRIAD" id=0x00000000">
<!-- Buses -->
<!-- Bridges -->
<!-- Modules -->
<!-- Signals -->
```

Each XML section and element is described in the following document sections.

6.2.1. Architecture element

The architecture element is the main element of the file. It is a mandatory element and all the other elements are embedded hierarchically within this <Architecture> tag. The element has two attributes

- <Name>, which contains the name of the architecture. In the example above, the name of the architecture is MYRIAD.
- <id>, which contains the architecture ID.

The user can create a hierarchy of architectures as described below:

```
<Architecture name="MULTIMYRIAD" id=0x00000000">
  <Architecture name="MYRIAD" id=0x00000001">
  <Architecture name="MYRIAD" id=0x00000002">
```

The depth of the architecture tree does not have a limit.

6.2.2. Bus element

A bus elements section must be present at the beginning of the <Architecture> element. As a rule, the <Bus> element must be defined before any master or slave that connects to the respective bus is defined in order to ensure that the bus connection is created properly. For ease of maintenance, it is recommended that all the buses in the system are defined at the beginning of the <Architecture> element in the XML file.

The bus element is used for specifying a generic bus block and can have the following configurable

parameters:

- width (8, 16, 32, 64 ... 1024 bits)
- master/slave interfaces (sockets) where other modules can be plugged
- arbitration model
- transfer protocol model: read and write latencies simulation for the bus, masters and slaves
- burst transfer mode
- pipelined transfer
- profiling information and monitors (bus traffic, bus load, power consumption, etc.)

From this block, several parameters are configurable via the XML file, using specific tags for each. Some of the tags are optional and there are default values implemented in case the user does not configure them in the XML. The bus element features the following tags:

- <Type> specifies the type of bus that is instantiated. Currently only AMBA/AXI buses are supported
- <Name> tag specifies a unique name that has to be given for each bus
- <DeviceId> specifies the unique device ID of this module within the architecture
- <ClockDivider> specifies a value that is used to divide the base clock for this component
- <Width> tag specifies the bus width in bits – can have the value 8, 16, 32, 64 ... 1024. This configuration is optional and the default value is 128.
- <Arbitration> tag specifies the type of arbitration that is used for this bus: RANDOM, ROUND_ROBIN, PRIORITY. Random priority means that the masters take precedence on a first-come, first-served basis. In case of two simultaneous requests, precedence of one over the other is not guaranteed. If PRIORITY arbitration is selected, each master on the bus must have the priority tag configured for its master interface. Otherwise, the resulting priority may not be the desired one. This configuration is optional and the default value is ROUND_ROBIN.
- <Burst> tag specifies if the bus supports burst transactions (True) or not (False). This parameter is optional and the default value is False.
- <PipelinedTransfer> tag is an integer representing the number of pipelined transactions. This parameter is optional and the default value is 0 (no pipeline).
- <SeparateDataBuses> tag specifies if the bus supports separate data buses for reads and writes (True) or not (False). This parameter is optional and the default value is False.
- <SimultaneousTransactions> an integer representing the number of simultaneous transactions serviced per cycle. The configuration is optional and defaults to 1.

Example

```
<Type>AMBA</Type>
<Name>SXI</Name>
<DeviceId>0x12345678</DeviceId>
<ClockDivider>0x1</ClockDivider>
 <Width>128</Width>
 <Arbitration>ROUND_ROBIN</Arbitration>
 <BurstEn>false</BurstEn>
 <PipelinedTransfer>0</PipelinedTransfer>
 <SeparateDataBuses>True</SeparateDataBuses>
 <SimultaneousTransactions>1</SimultaneousTransactions>
```

6.2.3. Bridge element

A <Bridge> element is used to specify a generic bridge – interconnecting elements on two different buses – and can be added to the XML file only after the two buses it connects have been added. The parser is not able to properly connect the bridge otherwise and outputs an error. This block implements a generic bridge that is capable of transferring transactions from one bus to another. The modeled bridge is unidirectional, so two such bridges are required to achieve bidirectional communication between two buses.

The <Bridge> element features several parameters that are configurable via the XML file using specific tags:

- <Name> a unique name that is mandatory for each bridge
- <ClockDivider> specifies a value that is used to divide the base clock for this component
- <MasterInterface> a tag that contains the description of a master connection to a specific bus. The master interface has the following sub-tags:
  - <Name> the name of the master interface, as it appears in the code. For the generic bridge provided by the simulator, this tag has the value masterIf
  - <Bus> the unique name of the bus this master interface connects to, used to identify the bus
  - <MasterID> a unique master ID, used for identifying the master across the bus, given as a hex number
- <SlaveInterface> a tag that contains the description of a slave connection to the specified bus. The slave interface has the following sub-tags:
  - <Name> the name of the slave interface, as it appears in the code. For the generic bridge provided by the simulator, this tag has the value slaveIf
  - <Bus> the unique name of the bus this slave interface connects to, used for identifying the bus
  - <SlaveID> a unique slave ID, used for identifying the slave across the bus, given as a hex number
  - <BaseAddress> the address in the memory map where the address space for this slave begins, given as a hex number
  - <MirrorBaseaddress> the address at which the base address is mirrored. If there is no mirroring, this can be 0
  - <MaxOffset> the maximum offset available, calculated as relative to the base address for this slave, given as a hex number
  - <PipelineDepth> specifies the latency through the bridge
  - <SplitBurstTransactionsEnable> specifies if burst transaction splitting into single transactions is enabled (True) or not (False)

Example

```
<Name>SXI2MXI1</Name>
<DeviceId>0x12345678</DeviceId>
<ClockDivider>0x2</ClockDivider>
  <Name>masterIf</Name>
  <Bus>MXI</Bus>
  <MasterID>0x11</MasterID>
  <Priority>0x1</Priority>
  <Name>slaveIf</Name>
  <Bus>SXI</Bus>
  <SlaveID>0x81</SlaveID>
```

```
  <BaseAddress>0x00000000</BaseAddress>
  <MirrorBaseAddress>0x0</MirrorBaseAddress>
  <MaxOffset>0x3FFFFFF</MaxOffset>
  <PipelineDepth>1</PipelineDepth>
<SplitBurstTransactionsEnable>False</SplitBurstTransactionsEnable>
```

6.2.4. Module elements

The XML file contains several module elements: A generic type PROCESSOR which is used for user-defined processor cores, specified via DLL Libraries. A generic type MODULE, which is used for user-defined peripheral modules, specified via DLL Libraries, and specific Module Elements like: CACHE, ICB, JTAG, MEM, CMX_MEM, CMX_CTRL, CPR. The module element contains several tags:

- <Type> tag can have the following values: PROCESSOR or MODULE for external components, CACHE, MEM for generic components and SHAVE, ICB, JTAG, CMX_MEM, CMX_CTRL, CPR or TIMERS for Movidius specific components
- <Name> tag contains a unique name of the element
- <DeviceId> tag contains a unique number of the element
- <ClockDivider> specifies a value that is used to divide the base clock for this component
- One or more <MasterInterface> tags if the module is a master on certain buses
- One or more <SlaveInterface> tags if the module is a slave on certain buses
- A <ImplementingDllName> tag, useful only if the module's functionality is implemented in a DLL. The tag specifies the DLL file name. The models provided by the simulator do not require this tag to be specified, as there is no DLL file for these models. The user must specify the name of the library without extension, for Windows/Linux compatibility, otherwise the simulator outputs an error
- A <Configuration> tag with additional information about the module. When used for external models (imported as .dlls), this configuration is not restrictive in terms of information, allowing any tags that are relevant for the external model to be defined. The information within the <Configuration> tag is taken by the XML parser and passed to the external module in the form of a configuration vector that stores all the elements within the tag

Example

```
<DeviceId>0x12345678</DeviceId>
<ClockDivider>0x2</ClockDivider>
<ImplementingDllName>file</ImplementingDllName>
```

Each type of module is described in the following sections.

6.2.4.1. CPUs

The simulator provides a CPU object type. This object is customizable to represent different system cores (e.g. LEON, ARM, SHAVE etc.). The CPU implements at least one Master interface for connection to a bus object. The internal makings of the CPU are core-specific and can be specified as a DLL Library.

6.2.4.1.1. SHAVE module element

The SHAVE processor is declared as a module in the architecture. As a result it has all the main tags of a module tag (see section above):

- <Type>: SHAVE
- <Name>: a unique name of the element
- <ClockDivider> specifies a value that is used to divide the base clock for this component
- <MasterInterface> tag
  - <Name>: sxiMasterIf
  - <Bus>: the unique name used for identifying the bus
  - <MasterID>: as a hex number
- <SlaveInterface> tag
  - <Name>: ahbSlaveIf
  - <Bus>: the unique name used for identifying the bus
  - <SlaveID>: as a hex number
  - <BaseAddress>: as a hex number
  - <MaxOffset>: as a hex number
- An <ImplementingDllName> tag which specifies the DLL file that contains the implementation
- A <Configuration> tag with additional information about the module

```
<Type>SHAVE</Type>
<Name>SHAVE2</Name>
<DeviceId>0x12345678</DeviceId>
<ClockDivider>0x1</ClockDivider>
   <Name>sxiMasterIf</Name>
   <Bus>SXI</Bus>
   <MasterID>0x2</MasterID>
   <Name>ahbSlaveIf</Name>
   <Bus>AHB</Bus>
   <SlaveID>0xC1</SlaveID>
   <BaseAddress>0x80160000</BaseAddress>
   <MirrorBaseAddress>0x0</MirrorBaseAddress>
   <MaxOffset>0x0000FFFF</MaxOffset>
<ImplementingDllName>shavesDll</ImplementingDllName>
```

6.2.4.1.2. PROCESSOR module element

The PROCESSOR type is used for all processor cores that are not Movidius IP. The tags necessary for the correct description of a processor are listed below.

- <Type>: PROCESSOR
- <Name>: a unique name of the element
- <DeviceId> tag containing a unique number of the element

- <ClockDivider> specifies a value that is used to divide the base clock for this component
- <MasterInterface> tag containing:
    - <Name>: the name of the master interface
    - <Bus>: the unique name used for identifying the bus
    - <MasterID>: as a hex number
- <SlaveInterface> tag containing:
    - <Name>: the name of the slave interface
    - <Bus>: the unique name used for identifying the bus
    - <SlaveID>: as a hex number
    - <BaseAddress>: as a hex number
    - <MaxOffset>: as a hex number
- An <ImplementingDllName> tag which specifies the implementation DLL file
- A <Configuration> tag with additional information about the module

```
<Type>PROCESSOR</Type>
<Name>LEON</Name>
<DeviceId>0x4C303030</DeviceId>
<ClockDivider>0x1</ClockDivider>
   <Name>masterIf</Name>
   <Bus>AHB</Bus>
   <MasterID>0x30</MasterID>
   <Name>bus_rr_interface</Name>
   <Bus>AHB</Bus>
   <SlaveID>0xB8</SlaveID>
   <BaseAddress>0xC0000000</BaseAddress>
   <MaxOffset>0x00FFFFFF</MaxOffset>
<ImplementingDllName>leonDll</ImplementingDllName>
<XmlFile>leon.xml</XmlFile>
```

6.2.4.2. below elements

6.2.4.2.1. Cache module element

The Cache element is a particular case of the Module element.
The Cache model has configurable parameters described in the table.

Table 2. Cache model configurable options

Parameter Name	Static/Dynamic	Description	Range
Line Width	Static	Cache line width	8-1024 bytes
No. of lines	Static	Number of Cache lines	0-1M
No. of sets	Static	Number of associative sets	DM, 2-set, 4-set
Replace Policy	Dynamic	Replacement policy used	BA, LRU, MRU, PLRU, SLRU, LFU, AR, MQCA

Write Policy	Dynamic	Write policy used	WT, WB
No. of partitions	Static	Partitions the cache can be split into	0-1023

From the Cache module, the following parameters are configurable via the XML file:

- <Type>: CACHE
- <Name>: a unique name has to be given for each element
- Functional <MasterInterface> containing:
  - <Name>: masterIf – this is predefined, as the cache model is generic
  - <Bus>: the unique name used for identifying the bus
  - <MasterID>: as a hex number
- Functional <SlaveInterface> containing:
  - <Name>: slaveIf – this is predefined, as the cache model is generic
  - <Bus>: the unique name used for identifying the bus
  - <SlaveID>: as a hex number
  - <BaseAddress>: as a hex number
  - <MirrorBaseAddress>: as a hex number
  - <MaxOffset>: as a hex number
- Control <SlaveInterface> containing:
  - <Name>: ctrlSlaveIf – this is predefined, as the cache model is generic
  - <Bus>: the unique name used for identifying the bus
  - <SlaveID>: as a hex number
  - <BaseAddress>: as a hex number
  - <MirrorBaseAddress>: as a hex number
  - <MaxOffset>: as a hex number
- <Configuration> tag with the following sub-tags:
  - <Size> cache size (in bytes)
  - <LineWidth> cache line size (in bytes)
  - <Sets> number of associative sets
  - <hitLatency> number of cycles it takes to service a hit
  - <IsWritethrough> specifies if the cache is writethrough (true) or not (false)
  - <ReplacePolicy> specifies the replacement policy used for associativity (BA, LRU, MRU, PLRU, SLRU, LFU, AR, MQCA)
  - <PartitionsNumber> specifies the number of partitions the cache is split into
  - <IsBypassed> enables (true) or disables (false) cache bypassing – a bypassed cache acts like a bridge between the master and slave interfaces. In this mode, the cache is able to split transactions if the functional slave interface bus is wider than the functional master bus
  - <BypassBase> enables the user to use a specific address space to bypass the cache while the

cache is still working for the rest of the slave interface address space

- <BypassMaxOffset> maximum offset from <BypassBase>

```
<Type>CACHE</Type>
<Name>L2CACHE</Name>
<DeviceId>0x12345678</DeviceId>
   <Name>masterIf</Name>
   <Bus>DXI</Bus>
   <MasterID>0x21</MasterID>
   <Name>slaveIf</Name>
   <Bus>SXI</Bus>
   <SlaveID>0x80</SlaveID>
   <BaseAddress>0x40000000</BaseAddress>
   <MaxOffset>0x0FFFFFFF</MaxOffset>
   <Name>ctrlSlaveIf</Name>
   <Bus>AHB</Bus>
   <SlaveID>0xBE</SlaveID>
   <BaseAddress>0x800F0000</BaseAddress>
   <MirrorBaseAddress>0x0</MirrorBaseAddress>
   <MaxOffset>0x0000FFFF</MaxOffset>
   <Size>0x20000</Size>
   <LineWidth>64</LineWidth>
   <Sets>2</Sets>
   <HitLatency>10</HitLatency>
   <IsWritethrough>false</IsWritethrough>
   <ReplacePolicy>LRU</ReplacePolicy>
   <PrtitionsNumber>1</PrtitionsNumber>
   <IsBypassed>false</IsBypassed>
   <BypassBase>0x48000000</BypassBase>
   <BypassMaxOffset>0x07FFFFFF</BypassMaxOffset>
```

6.2.4.2.2. Memory element

The Memory element is another particular case of the Module element. The element can describe memories as SRAM and DDR. The simulator provides a generic memory model that can be added to the system and configured to emulate multiple types of memory either on-chip or off-chip.

Table 3. Memory model configurable options

Parameter Name	Static/Dynamic	Description	Range
Size	Static	Memory size	0-1GB
Read Latency	Static	Latency of reads	0-1023
Write Latency	Static	Latency of writes	0-1023
Read ports	Static	Number of simultaneous read per cycles	1-1023
Write ports	Static	Number of simultaneous writes per cycle	1-1023
Read before write	Static	Define policy when 2 or more ports access the same location	Default is LRU

Error Checking	Static	Error detection mechanism	No checking, parity

From the Memory module, several parameters are configurable via the XML tags:

- <Type>: MEM
- <Name>: a unique name has to be given for each element
- <MemType>: type of memory modeled – SRAM, ROM, DDR
- <SlaveInterface>
  - <Name>: busInterface (this name is mandatory for the generic memory model provided in the simulator)
  - <Bus>: the unique name used for identifying the bus
  - <SlaveID>: as a hex number
  - <BaseAddress>: as a hex number
  - <MirrorBaseAddress>: as a hex number
  - <MaxOffset>: as a hex number
- <Configuration> tag with the following sub-tags:
  - <Size> represents the memory size (in bytes)
  - <ReadLatency> integer value
  - <WriteLatency> integer value
  - <ReadPorts> number of simultaneous reads per cycles
  - <WritePorts> number of simultaneous writes per cycles
  - <PortClashPolicy> READ_BEFORE_WRITE, WRITE_BEFORE_READ
- <ParityCeheckEn> enables (true) or disables (false) parity checking
- <InitFile> specifies a binary file to be loaded into the memory for initial values

```
<Type>MEM</Type>
<Name>LRAM</Name>
<MemType>SRAM</MemType>
<DeviceId>0x12345678</DeviceId>
   <Name>busInterface</Name>
   <Bus>AHB</Bus>
   <SlaveID>0xB1</SlaveID>
   <BaseAddress>0x90100000</BaseAddress>
   <MirrorBaseAddress>0x0</MirrorBaseAddress>
   <MaxOffset>0x00007FFF</MaxOffset>
   <Size>0x8000</Size>
   <ReadLatency>5</ReadLatency>
   <WriteLatency>5</WriteLatency>
   <ReadPorts>1</ReadPorts>
   <WritePorts>1</WritePorts>

<PortClashPolicy>READ_BEFORE_WRITE</PortClashPolicy>
   <ParityCeheckEn>false</ParityCeheckEn>
<Type>MEM</Type>
```

```
<Name>DDR</Name>
<DeviceId>0x12345678</DeviceId>
   <Name>busInterface</Name>
   <Bus>DXI</Bus>
   <SlaveID>0xA0</SlaveID>
   <BaseAddress>0x40000000</BaseAddress>
   <MirrorBaseAddress>0x48000000</MirrorBaseAddress>
   <MaxOffset>0x0FFFFFFF</MaxOffset>
<ReadPorts>1</ReadPorts>
<WritePorts>1</WritePorts>
<Size>0x1000000</Size>
<ReadLatency>30</ReadLatency>
<WriteLatency>30</WriteLatency>
```

6.2.4.2.3. DMA module

The module element is used to transfer data between two memory sections. This process is done by configuring the DMA module parameters. User can set the number of active channels using the <numberOfChannels> tag. The rest of the tags are also found in configurable elements tags:

- <Type>: DMA
- <Name>: a unique name of the module
- <MasterInterface> containing:
  - <Name>: the name of the master interface
  - <Bus>: the unique name used for identifying the bus
  - <MasterID>: as a hex number
- <SlaveInterface> containing:
  - <Name>: the name of the slave interface
  - <Bus>: the unique name used for identifying the bus
  - <SlaveID>: as a hex number
  - <MaxOffset>: as a hex number
  - <BaseAddress>: as a hex number
- <Configuration>: additional information about the module (in this case the number of channels)

```
<Module>
   <Type>DMA</Type>
   <Name>DMA1</Name>
      <Name>busSlave</Name>
      <Bus>MXI</Bus>
      <SlaveID>0x95</SlaveID>
      <BaseAddress>0x60000000</BaseAddress>
      <MirrorBaseAddress>0x0</MirrorBaseAddress>
      <MaxOffset>0x0000FFFF</MaxOffset>
      <Name>busMaster</Name>
      <Bus>MXI</Bus>
      <MasterID>0x14</MasterID>
```

```
<numberOfChannels>32</numberOfChannels>
```

DMA module requires setting certain control parameters. For details, see section TBD.

6.2.4.3. CMX Memory element

The CMX Memory element is another particular case of the Module element. The declaration of the CMX Memory element is analog to the declaration of the Memory element.

Several parameters are configurable via the XML file tags:

- <Type>: CMX_MEM
- <Name>: a unique name has to be given for each element
- Two <SlaveInterface> tags with the following sub-tags for each:
  - <Name>: slaveIfMxi and slaveIfAhb – these names are mandatory as this module is provided by the simulator framework
  - <Bus>: the unique name used for identifying the bus
  - <SlaveID>: as a hex number
  - <BaseAddress>: as a hex number
  - <MirrorBaseAddress>: as a hex number
  - <MaxOffset>: as a hex number
- <Configuration> tag with the following sub-tags
  - <Size>: represents the total size of the memory – hexadecimal value (in bytes)
  - <NumberOfSlices>: represents the number of slices the memory is split into. In order to function properly, the memory size must be dividable by the number of slices in such a way that the resulting size of one slice is a multiple of 8 bytes (which is the size of a SHAVE data port)
  - <SliceSize>: represents the size of a memory slice – hexadecimal value (in bytes)

```
<Type>CMX_MEM</Type>
<Name>CMX</Name>
   <Name>slaveIfMxi</Name>
   <Bus>MXI</Bus>
   <SlaveID>0x90</SlaveID>
   <BaseAddress>0x10000000</BaseAddress>
   <MirrorBaseAddress>0x0</MirrorBaseAddress>
   <MaxOffset>0x000FFFFF</MaxOffset>
   <Name>slaveIfAhb</Name>
   <Bus>AHB</Bus>
   <SlaveID>0xB0</SlaveID>
   <BaseAddress>0xA0000000</BaseAddress>
   <MirrorBaseAddress>0x0</MirrorBaseAddress>
   <MaxOffset>0x000FFFFF</MaxOffset>
   <Name>slaveIfAmc</Name>
   <Bus>AMC</Bus>
   <SlaveID>0xD0</SlaveID>
   <BaseAddress>0x10000000</BaseAddress>
   <MirrorBaseAddress>0x0</MirrorBaseAddress>
```

```
    <MaxOffset>0x000FFFFF</MaxOffset>
    <Size>0x100000</Size>
    <NumberOfSlices>8</NumberOfSlices>
    <SliceSize>0x20000</SliceSize>
```

6.2.4.4. Other module elements

6.2.4.4.1. Master elements

The module element allows the user to specify modules which act as masters on a certain bus by specifying the mandatory fields: type, name and master interface. An example of such element is the JTAG module:

```
<Type>JTAG</Type>
<Name>JTAG</Name>
    <Bus>AHB</Bus>
    <MasterID>0x32</MasterID>
```

6.2.4.4.2. Slave elements

In the case of modules that act as slaves on certain buses, the mandatory fields are: type, name and slave interface. Examples of such elements are `ICB`, `CMX_CTRL`, `UART` or `CPR`.

```
<Type>ICB</Type>
<Name>ICB</Name>
    <Name>bus_rr_interface</Name>
    <Bus>AHB</Bus>
    <SlaveID>0xBA</SlaveID>
    <BaseAddress>0x80010000</BaseAddress>
    <MirrorBaseAddress>0x0</MirrorBaseAddress>
    <MaxOffset>0x0000FFFF</MaxOffset>
<Type>CMX_CTRL</Type>
<Name>CMXCTRL</Name>
    <Name>slaveIf</Name>
    <Bus>AHB</Bus>
    <SlaveID>0xB6</SlaveID>
    <BaseAddress>0xAE000000</BaseAddress>
    <MirrorBaseAddress>0x0</MirrorBaseAddress>
    <MaxOffset>0x0000a1FF</MaxOffset>
    <NumberOfShaves>8</NumberOfShaves>
    <NumberOfMutexes>8</NumberOfMutexes>
<Type>UART</Type>
<Name>UART0</Name>
    <Name>bus_rr_interface</Name>
    <Bus>AHB</Bus>
    <SlaveID>0xB7</SlaveID>
    <BaseAddress>0x80000000</BaseAddress>
    <MirrorBaseAddress>0x0</MirrorBaseAddress>
    <MaxOffset>0x0000ffff</MaxOffset>
<Type>CPR</Type>
<Name>CPR</Name>
```

```
    <Name>slaveIf</Name>
    <Bus>AHB</Bus>
    <SlaveID>0xBB</SlaveID>
    <BaseAddress>0x80030000</BaseAddress>
    <MirrorBaseAddress>0x0</MirrorBaseAddress>
    <MaxOffset>0x0000FFFF</MaxOffset>
```

If the module is implemented into a DLL file, this can and must be mentioned in a specific tag. An example of such a module is TIMERS:

```
<Type>MODULE</Type>
<Name>TIMERS</Name>
    <Name>bus_rr_interface</Name>
    <Bus>AHB</Bus>

    <SlaveID>0xB5</SlaveID>
    <BaseAddress>0x80020000</BaseAddress>
    <MirrorBaseAddress>0x0</MirrorBaseAddress>
    <MaxOffset>0x0000FFFF</MaxOffset>
<ImplementingDllName>timersDll</ImplementingDllName>
```

More details on how to integrate an external DLL file with the simulator is provided in chapter 7.

6.2.5. Signal element

The signal elements can be specified in the signal section which is typically located at the end of the <Architecture> element. While it is not mandatory to place the signals in a grouped manner at the end of the XML file, it is highly recommended to do so in order to avoid certain mistakes that might cause the simulator to malfunction.

Signals are used to simulate connections between modules that do not go through the bus (e.g. an interrupt line, the reset input etc.). They can be viewed as events to which a module can be sensitive to. In order to be able to define a signal, both the source and the destination modules have to be defined in advance. A signal can be connected to one or more destination modules, but it is mandatory that it has a unique source module.

From the Signal component, several parameters are configurable via the XML file:

- <Type>: represents the type of the signal: bool (used for a single wire connection), unsigned_int (used for a connection of up to 32 wires), pointer (when two pointers inside the simulator need to be connected), specific interface type (SHAVE to CMX Interface for instance). The signals at source and destination must be coded using this specified type.
- <Source>: tag used to describe the source module with the following sub-tags:
- <Name>: the unique module name used for identifying the source element
- <SigName>: the name of the output signal at the source
- <Destination>: used to describe the destination module, with the following sub-tags:
    - <Name>: the unique name used for identifying the destination element
    - <SigName>: the name of the input signal at the destination

```
<Type>unsigned_int</Type>
```

```
<Name>LEON</Name>
<SigName>interruptAck</SigName>
<Name>ICB</Name>
<SigName>interruptAck</SigName>
```

7. Importing And Using External Models Within The Simulator

The simulator framework allows users to create their own models of components and import them into the simulator using the external DLL interface. In order to be used, the external model needs to conform to specific requirements. For this purpose, a library containing relevant structures is provided in the [moviSim\modelLibrary] folder.

7.1. DLL requirements

The external dynamic library must implement a standard interface for the simulator to be able to initialize the library and dynamically create model objects. This interface consists of the following functions:

- A function for library initialization:

```
void initLibrary(void* arguments, char* version)
```

Parameter arguments can be used to enable certain logging and displaying features. The pointer that is passed by the simulator infrastructure is the address of a structure of the MsimModelInterface type. This structure is defined in the moviTypes.h include file, which is part of the moviSim model support library and is provided for model development purposes only. The structure is defined as follows:

```
{
bool noMessages; //Quiet mode selected
bool echoTRFReads; //Echo TRF Reads
bool echoTRFWrites; //Echo TRF Writes
bool echoRFReads; //Echo IRF/SRF/VRF Reads
bool echoRFWrites; //Echo IRF/SRF/VRF Reads
bool echoMemoryReads; //Echo Memory Reads
bool echoMemoryWrites; //Echo Memory Writes
bool echoLeonRegReads; //Echo Leon register reads
bool echoLeonRegWrites; //Echo Leon register writes
bool echoLeonMemReads; //Echo Leon Memory reads
bool echoLeonMemWrites; //Echo Leon Memory writes
bool echoIDC; //Echo IDC decoding and disassembling
bool echoDCU; //Echo DCU reads/writes
bool readPortCheck; //Check read ports
bool writePortCheck; //Check write ports
bool enBusLogging;
bool enCoreLogs;
bool enProfiling;
bool icbDisable;
bool enableRBWChecks; //Enable read before write check for memories
bool functionProfiling; //Enable SHAVE functions profiling
bool dataLabelsProfiling; //Enable SHAVE data labels profiling
bool codeLabelsProfiling; //Enable SHAVE code labels profiling
bool asmLineProfiling; //Enable SHAVE asm line profiling
bool echoTransactions;



string outputDir;
string simDir;
```

```
string configDir;
string workingDir;
string tempDir;
//messaging functions pointers
sendInfoPtr sendInfoMessage;
sendErrorPtr sendErrorMessage;
sendWarningPtr sendWarningMessage;
sendNotePtr sendNoteMessage;
//function, line and label profiling vectors
vector<ProfilingVectors*> profVectors;
} MsimModelArguments;
```

Although most of these are targeted for the Movidius models, some of them can be used for enabling and disabling features, like register port clash checking or message printing, for other models created by the user. The simulator folders can be used if the user wants to create user-defined log files. Also, the pointer received as parameter is required for using the messaging system of the simulator (see chapter 7.4).

- A function for object creation:

```
void* createObject(string* name, vector<Configuration>* config);
```

Parameter `name` is a pointer to a string containing the unique name of the module. The parameter `config` is a pointer to a vector of Configuration items that have the following form:

```
{
string tag;
string value;
} Configuration;
```

Each entry corresponds to an entry in the `<configuration>` section specified for the respective module in the `XML` file that describes the architecture.

The `DLL` must use the moviIfLib library as a dependency in order to have access to several common structures that are required for the correct integration of the model with the simulator. The moviIfLib library and header files are provided together with the `Movidius` libraries in the moviSim package.

The user must include the `moviIfLib.h` file in the implementation of the module in order to ensure that all the required structures and classes are properly defined.

7.2. External model implementation requirements

In order to correctly link the external module into the simulator, several methods and structures are required. For connecting to a bus, a master or a slave interface object must be defined. The declarations for these objects can be found in the `simBusIf.h` file found within the model support library.

The runtime methods are encapsulated into a modelInterface class that must be virtually inherited by the model class.

These classes and methods are subject to change due to future optimization and improvements.

7.2.1. Bus connection classes and functions

The simBusIf.h file includes the definition for the busMasterInterface class and the busSlaveInterface class. These classes operate using a generic bus transaction structure, defined as follows:

```
struct bus_transaction
{
   //transaction info
   unsigned int master_id;
   unsigned long long address;
   unsigned int cycles;
   unsigned int data[MAX_BUS_SIZE / 32];
   unsigned int byte_en[4];
   unsigned int size; //in bytes
   bool is_read;
   //burst info
   bool isBurst; //if false - burst not enabled
   unsigned char *burstData;    //data to be transferred
   unsigned int burst_id;     //burst code
   T_BURST_TYPE burstType;    //burst type
   unsigned int burstSize;    //burst size of each unit
   unsigned int burstLen;    //length of burst data
}
```

The master interface is defined as a class that uses a transaction queue (masterRecQueue) for incoming read returns. The user must check for the read return in the queue and take the transactions out of the queue as they arrive. This is usually done in the execute function (see chapter 7.2.2.2).

For sending requests, the user must encapsulate the request in a T_BUS_TRANSACTION structure and use either the masterSend method (for single shot requests) or the masterSendBurst method (for sending burst requests). Burst requests can only be sent if the bus is configured to support them. If the bus does not support bursts, the results may be different than expected.

The class definition is presented below.

```
class busMasterInterface { public:    busMasterInterface();
   ~busMasterInterface();
   void AddBusInterface(void* ptr);
   // receive interface (for reads)
   void masterReceive (T_BUS_TRANSACTION trans);
   // send interface
   void masterSend (T_BUS_TRANSACTION trans);
   // burst send interface
void  masterSendBurst(unsigned  char  *data,  unsigned  int  master_id,  unsigned  int  address,
T_BURST_TYPE burstType = B_FIXED, unsigned int burstSize = 0, unsigned int burstLen = 0);
   deque<T_BUS_TRANSACTION> masterRecQueue;
   // pointer to bus (needs to be bound to call correctly)
   busInterface* busMInterface;
};
```

The slave interface is defined as a class that uses a transaction queue (slaveRecQueue) for incoming requests. The user must check this queue, typically in the execute function (see chapter 7.2.2.2), in order to service any incoming requests and take them out of the queue. The bus structure is waiting for responses to read transactions, so a response must be sent using the slaveReadReturn function, sending an updated

transaction structure as argument. No response is required for write transactions. The busSlaveInterface class definition is presented below.

```
class busSlaveInterface
{
public:
   busSlaveInterface(unsigned int base, bool rebaseEn);
   busSlaveInterface();
   ~busSlaveInterface();
   void AddBusInterface(void* ptr, unsigned int base, bool rebaseEn);
   void SetBase(unsigned int base);
   // receive interface
   void slaveReceive (T_BUS_TRANSACTION trans);
   // send interface
   void slaveReadReturn (T_BUS_TRANSACTION trans);
   deque<T_BUS_TRANSACTION> slaveRecQueue;
   // for reads - pointer to bus (needs to be bound to call correctly)
   busInterface* busSInterface;
   unsigned int base;
   unsigned long long mirrorBase;
   bool rebaseEn;
};
```

7.2.2. Interface classes and functions description

The modelInterface class is defined in the modelInterface.h file and has the following methods:

```
class modelInterface
{
public:
   //reset interface
   virtual void reset(void) = 0;
   //cycle execute interface
   virtual void execute(unsigned long long timeStamp = 0) = 0;
   //get signal pointer based on signal name
   virtual void * getSignalPtr(char * sigName) = 0;
   //get slave interface pointer
   virtual void* getSlaveInterface(char* busName) = 0;
   //get master interface pointer
   virtual void* getMasterInterface(char* busName) = 0;
   //get profiling info
   virtual void getProfilingInfo(void *) = 0;
   // set profiling info
   virtual void setProfilingInfo(void *) = 0;
   //check parameter interface
   virtual unsigned int checkParameter(void) = 0;
   //get halted state
   virtual unsigned int getHaltState(void) = 0;
   //run the simulation with a specific entry point - used for processors
   virtual void runSimulation(unsigned long long entry) = 0;
};
```

The user class must inherit this class and all of the methods must be implemented in the user's library.

7.2.2.1. Function "reset"

The function is called when a system reset occurs. This is similar to the activation of the reset pin in hardware. This function takes no parameters.

7.2.2.2. Function "execute"

The execute function is used to simulate the execution of the model on every clock cycle. The argument `timestamp` is optional and represents the current cycle number. This function is called by the runtime once per execution cycle, so care should be taken for it to be as optimized as possible, otherwise the speed of the simulator decreases.

7.2.2.3. Function "getSignalPtr"

This function is used by the runtime when connecting modules using signals. The parameter `sigName` represents the name given to the signal in the `XML` description file. The return value should be a pointer to the signal object identified by sigName.

7.2.2.4. Function "getSlaveInterface"

This function is used by the runtime when connecting modules to the bus using the slave interface. The parameter `busName` represents the name given to the bus this model is connecting to in the `XML` description file. The return value should be a pointer to the `busSlaveInterface` object that was instantiated for connecting to the respective bus as a slave.

7.2.2.5. Function "getMasterInterface"

This function is used by the runtime when connecting modules to the bus using the master interface. The parameter `busName` represents the name given to the bus this model is connecting to in the XML description file. The return value should be a pointer to the busMasterInterface object that was instantiated for connecting to the respective bus as a master.

7.2.2.6. Function "getProfilingInfo"

The function is used in order to collect profiling information from the external processor modules. The profiling information is provided using the infoPtr structure. The contents of the structure are tailored for the specific model that implements the function. For the `SHAVE` processor, the contents are defined as:

```
{
   string coreName;
   unsigned int cycleCount;
   double internalPowerConsumed;
   unsigned int instructionCount;
   unsigned int stallCount;
   unsigned int stallBruStarve;
      unsigned int stallBruMiss;
```

```
      unsigned int stallIdcFifoLow;
      unsigned int stallIdcAccess;
      unsigned int stallLsu0Access;
      unsigned int stallLsu1Access;
      unsigned int stallLsu0WaitData;
      unsigned int stallLsu1WaitData;
   unsigned int ddrReads8;
   unsigned int ddrReads16;
   unsigned int ddrReads32;
   unsigned int ddrReads64;
   unsigned int ddrReads128;
   unsigned int ddrWrites8;
   unsigned int ddrWrites16;
   unsigned int ddrWrites32;
   unsigned int ddrWrites64;
   unsigned int ddrWrites128;
   unsigned int cmxReads8;
   unsigned int cmxReads16;
   unsigned int cmxReads32;
   unsigned int cmxReads64;
   unsigned int cmxReads128;
   unsigned int cmxWrites8;
   unsigned int cmxWrites16;
   unsigned int cmxWrites32;
   unsigned int cmxWrites64;
   unsigned int cmxWrites128;
   unsigned int cmxIdcReadsBySlice[8];
      unsigned int cmxLsuReadsBySlice[8];
      unsigned int cmxLsuWritesBySlice[8];
      unsigned int cmxReadsByAXI;
      unsigned int cmxWritesByAXI;
   unsigned int cmxDMAReads;
   unsigned int cmxDMAWrites;
   unsigned int ddrDMAReads;
   unsigned int ddrDMAWrites;
   unsigned int irfReads[32];
      unsigned int irfWrites[32];
      unsigned int srfReads[32];
      unsigned int srfWrites[32];
      unsigned int vrfReads[32];
      unsigned int vrfWrites[32];
} ShaveProfilingInfo;
```

For other processors, a more generic structure is defined:

```
{
   unsigned int cycleCount;
   double internalPowerConsumed;
   unsigned int instructionCount;
   unsigned int stallCount;
} CoreProfilingInfo;
```

The profiling structure for caches is defined as:

```
{
   unsigned int cacheMisses;
   unsigned int cacheHits;
} CacheProfilingInfo;
```

The contents for these structures and structure types are subject to change, as more types of components and more profiling information is added to the simulator and is frozen at a later date.

7.2.2.7. Function "setProfilingInfo"

This is used for resetting the profiling counters to specific values or resetting them to 0. The parameter is of the same type as for `getProfilingInfo` (Chapter 7.2.2.6).

7.2.2.8. Function "checkParameter"

The checkParameter function is used for model self-checking. Each module should implement this function and use it to check the correctness of its signals, interfaces and configurations. A return value of 0 means that there is no error, while a different return value suggests an error. Each module should define a relevant error code list similar to the one below in order to better trace the error to its source.

```
{
   ERROR_FREE=0,
   MEM_INVALID_SIZE,
   MEM_INVALID_READ_PORTS,
   MEM_INVALID_WRITE_PORTS,
   MEM_INVALID_READ_LATENCY,
   MEM_INVALID_WRITE_LATENCY,
   MEM_ERROR_MEM_ALLOCATION,
   MEM_ERROR_INITCHECK_ALLOCATION,
   MEM_ERROR_PARITY_ALLOCATION,
   MEM_SBUSIF_ALLOCATION_ERROR,
   MEM_SBUSIF_INVALID_BASE_ADDR,
   BRIDGE_INVALID_INPUT_BUS_SIZE,
   BRIDGE_INVALID_OUTPUT_BUS_SIZE,
   BRIDGE_MBUSIF_ALLOCATION_ERROR,
   BRIDGE_SBUSIF_ALLOCATION_ERROR,
   BRIDGE_SBUSIF_INVALID_BASE_ADDR,
   CACHE_INVALID_SIZE,
   CACHE_INVALID_LINE_SIZE,
   CACHE_INVALID_LINE_NR,
   CACHE_INVALID_LINE_SIZE_NR,
   CACHE_INVALID_SET_NR,
   CACHE_INVALID_SET_LINES,
   CACHE_INVALID_BYTE_OFFSET,
   CACHE_INVALID_LINE_ADDR_BITS,
   CACHE_INVALID_MASTER_SIZE,
   CACHE_INVALID_SLAVE_SIZE,
   CACHE_MBUSIF_ALLOCATION_ERROR,
   CACHE_SBUSIF_ALLOCATION_ERROR,
   CACHE_SBUSIF_INVALID_BASE_ADDR,
   CACHE_ALLOCATION_ERROR,
```

```
    BUS_INVALID_WIDTH,
    BUS_INVALID_MASTER_NR,
    BUS_INVALID_SLAVE_NR,
    BUS_DUPLICATE_MASTER_ID,
    BUS_DUPLICATE_SLAVE_ID,
    BUS_DUPLICATE_BASE_ADDRESS,
    BUS_OVERLAPPING_ADDRESS,
    BUS_MIF_ALLOCATION_ERROR,
    BUS_SIF_ALLOCATION_ERROR
} T_ERRORS;
```

7.2.2.9. Function “getHaltState”

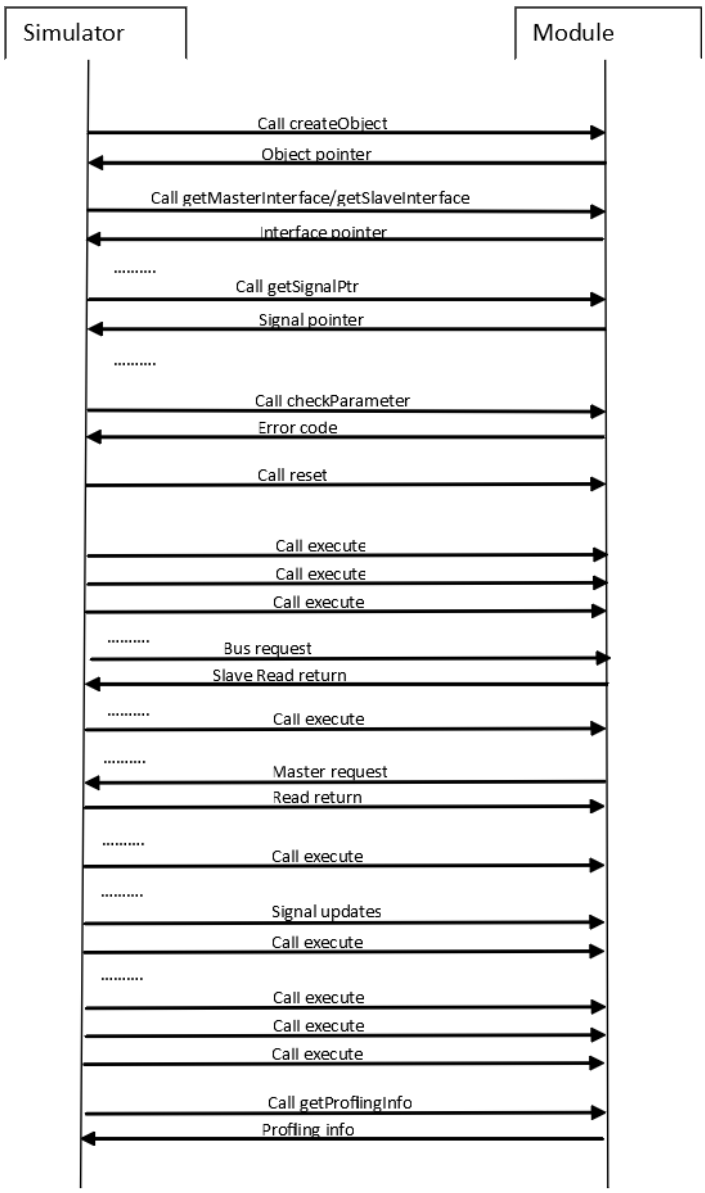
This function returns a boolean value according to the state of the current processing element. For each existing core `GetHaltState` checks the core state and if the core cycle count is active, it gets its internal state.

7.2.2.10. Function “runSimulation”

This function is called in the RunSimulation method with an `unsigned long long` parameter which is the entry point of the specified component.

7.3. Framework interaction with the model

The flow of the interactions between the `moviSim` framework and the external models is described below.



7.4. Messaging center interface

The simulator provides the means for sending 4 types of messages back to the simulator core that can be printed and/or logged to the screen/log files. In order to be able to use this feature, the user must first define a pointer to the MsimModelArguments structure and assign to it the value received as parameter in the initLibrary() function call. The pointer must be defined exactly as follows:

```
MsimModelArguments* modelArgumentsPtr;
```

NOTE:	The pointer name is important and must be the same for all libraries.

The user can send 4 types of text messages:

- Error message – the text prefix is `ERROR:` and the simulator can be instructed to log these messages (see `-e` command line argument) or to stop at the first error detected (`-n`).

- Warning message – the text prefix is `WARNING:`
- Info message – the text prefix is `INFO:`
- Note message – no prefix.

The simulator provides the following message sending macros that are included in the `moviMsg.h` file:

```
psSendInfo(deviceId, messageId, count, ⋯)
psSendError(deviceId, messageId, count, ⋯)
psSendWarning(deviceId, messageId, count, ⋯)
psSendNote(deviceId, messageId, count, ⋯)
```

There is one message ID reserved for each type of message. The available message IDs are:

```
ERR_EXTERNAL_DLL
WARN_EXTERNAL_DLL
INFO_EXTERNAL_DLL
NOTE_EXTERNAL_DLL
```

They are available once the user has included the `moviMsg.h` file.

In order to send an error message, the user would call the respective macro as follows:

```
psSendError(deviceId, ERR_EXTERNAL_DLL, 1, "Error message to display.");
```

where the deviceId is the one specified in the `XML` for this device, the message ID is the one for errors and the count is set to 1, as the `ERR_EXTERNAL_DLL` message only needs one additional char* parameter (the text message).

The same procedure is valid for warnings, infos and notes, using the respective message IDs.

Multiple parameter messages are not available for external models at the moment.

8. Using Computer Vision Models in PC simulation

8.1. Introduction

For external usage, out of MoviSim, of the computer vision(CV) algorithms and ma2x9x CNNBlock we provide a unified interface for fast prototyping and functional testing.

8.2. Prerequisits

The dynamic libraries automatically export the interface described in the subchapters below. They are used by moviSim and can be used standalone by the users also. The dymanic libraries have been build with Visual Studio 2015(Version 14.0.25431.01 Update 3) for Windows and under Centos 6 with gcc 6.4 for Linux.

NOTE:	The CV filters can only be run in full frame mode

8.3. Supported modules

So far Warp, Stereo modules for ma2x8x and DPUBlock for ma2x9x have the support for out of moviSim interface usage. Other modules are work in progress and will be available in a future release.

8.4. What is provided

8.4.1. Interface files

There are two header files that are needed to be included into your project in order to be able to use the API

`filtersConfiguration.h`	Provides structres for configuring CV filters, StereoConfiguration and WarpConfiguration and setting the **dpuMemoryAccess** for DPUBlock
`filterInterface.h`	Provides a C++ interface via FilterInterface virtual class.

The location of these files is `ToolsFolder\examples\moviSim\moviFilterApi\Includes`

8.4.2. Interface description

Each supported module described above, that is provided as a dynamic library, exports a 'createFilterObject' method

Windows	`extern "C" __declspec(dllexport) class FilterInterface* createFilterObject(const char* name);`
Linux	`extern "C" class FilterInterface* createFilterObject(const char* name);;`

Create Filter Object function returns a FilterInterface pointer that will be used to call well defined oprations

on the CV filter. The operations are described below

Member signature	Functional description.
`FilterStatus configure(void* filterConfiguration)`	After instantiating a model via the createFilterObject you will need to call the configure member that will pass the configuration to the CV filter model.
`FilterStatus start(FilterStartMode startMode)`	Start member will start the CV filter module with the configuration passed in the previous step.
`FilterStatus apbWriteRegister(size_t address, uintptr_t data, size_t size = 0)`	writes module register specified by 'address' using a 'data' pointer of given 'size'
`FilterStatus apbReadRegister(size_t address, uintptr_t data, size_t size = 0)`	reads module register specified by 'address' value returned in 'data' pointer of a 'size'
`void reset()`	Reset member to revert the model to a default initial state.

8.5. Steps in how to use

First you will need to load the dynamic libraries.

Than you will need to get the procedure address of createFilterObject function via GetProcAddress(library, "createFilterObject");

Call the creteFilterObject function to instantiate the CV model. This will return a derived class FilterInterface pointer.

Call configure member passing a pointer to the configuration. The filter configuration structures are described in filtersConfiguration.h file.

Call Start`to run the desired CV model.

In the output buffer(s) you have provided in the configuration structure you should have the processed data from the CV model.

In order to rerun you will need to call configure and than start member functions.

NOTE For DPU please see bellow: **Examples of usage for Keembay [DPU:CNN]**

8.6. Examples of usage for CV API [Warp, Stereo]

In `ToolsFolder\examples\moviSim\moviFilterApi\moviFilterApiTests\` there is a Visual Studio 2015 solution and a Makefile that provide an example from on how to use this interface. A couple of examples with different configurations are provided in order to view some of the features the CV API models provide. What you need to setup before you run these tests (by default both Warp and Stereo are run) is the location of the dynamic libraries and the resource folder. After the build of the Visual Studio solution or Makefile, call moviFilterApiTests executable for usage.

```
Usage: moviFilterApiTests ..\..\..\..\win32\models Resources where first argument is the
location of Warp and Stereo libraries and second is the resources folder, images, binary files,
etc.
```

8.7. Examples of usage for Keembay [DPU:CNN]

In `ToolsFolder\examples\moviSim\moviFilterApi\moviFilterApiTests\` there is a Visual Studio 2015 solution and a Makefile that provide an example from on how to use this interface. In example files: dpuResources.h, dpuFilterApiTest.h and dpuFilterApiTest.cpp is an example how to configure and use the DPUBlock.

NOTE See example files in order to understand the usage;

8.7.1. Set up CMXNN memory, write DPUBlock registers and start DPUBlock:

1. define on your side a memory buffer(do not exceed 2M); in the example that is provided the name of the buffer is: **cmxNnMemory**
2. implement 2 functions named **void dataRead(uint64_t address, uintptr_t *data, size_t size);** and **void dataWrite(uint64_t address, uintptr_t data, size_t size);**
3. create a dpuMemoryAccess* dataReadWriteAccessPtr = &dataReadWriteAccess; and set in dataReadWriteAccess the callbacks with the newlly defined functions at point 2.(see initDataAccess();)
4. See the flow on **ApbWriteReadSimulation():**
5. call **dpuFilter→configure((void*)dataReadWriteAccessPtr);** to register the callbacks in DPUBlock;
6. set input data in newlly defined **cmxNnMemory** that will be consumed by DPUBlock;
7. call **dpuFilter→apbWriteRegister();**, see: **dpuApbWrites()**, in order to setup DPUBlock registers(input data, configuration, output buffer address in **cmxNnMemory**)
8. when all the needed registers are set, call: **dpuApbWriteStartRegister();**, this will **start** DPUBlock;
9. results are expected to be written in provided output buffers from **cmxNnMemory**

NOTE because of no full setup yet present, no functional examples are provided;

NOTE for any problems contact moviSim team and CNN software model team

NOTE see for DPU features support: ToolsFolder\examples\moviSim\moviFilterApi\moviFilterApiTests\Resources\releaseNotes.txt

9. References

[1] starpu.pdf – INRIA

[2] Myriad Specification – Movidius Ltd. internal documentation

10. Glossary

SoC System-on-Chip

NoC Network-on-Chip

EBI External Bus Interface

DMA Direct Memory Access

CIF Camera Interface

FIFO First In First Out