Using OVP Models in SystemC TLM2.0 Platforms

Imperas Software Limited
Imperas Buildings, North Weston,
Thame, Oxfordshire, OX9 2HA, UK
docs@imperas.com

Author:	Imperas Software Limited
Version:	2.2.0
Filename:	OVPsim_Using_OVP_Models_in_SystemC_TLM2.0_Platforms.doc
Project:	Using OVP Models in SystemC TLM2.0 Platforms
Last Saved:	Tuesday, 06 October 2020
Keywords:	

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

1 Introduction

This document describes the use of OVP models in systemC TLM2.0 simulation platforms using the OP C++ API. Earlier releases of the Imperas simulator used the deprecated (still supported) ICM C++ API.

CpuManager and OVPsim are dynamic linked libraries (`.so` suffix on Linux, `.dll` suffix on Windows) implementing Imperas simulation technology. The shared objects contain implementations of the OP interface functions described in "OVPsim and CpuManager User Guide". The OP functions enable instantiation, interconnection and simulation of complex multiprocessor platforms containing arbitrary shared memory topologies and peripheral devices.

CpuManager is one of the commercial products available from Imperas. OVPsim available from www.ovpworld.org

It is assumed that you are familiar with C++, SystemC and TLM2.0 technology.

Please refer to the "OVPsim and CpuManager User Guide" for more details of OVPsim and CpuManager products.

1.1 Why use CpuManager or OVPsim?

OVPsim and CpuManager (hereafter referred to as just CpuManager) have access to a rich source of fast, qualified processor models and to a constantly growing list of peripheral models. Using CpuManager in your SystemC TLM2.0 simulation gives access to these high performance models, and to associated software development tools with very little extra effort.

1.2 Restrictions

CpuManager is a very high speed instruction-accurate processor and platform simulator. It is not intended for cycle-accurate or pin-level simulation. For this reason the TLM2.0 interface uses the TLM2.0 "loosely timed" (LT) model. Attempting to use other models will give incorrect results.

CpuManager allows the free-running of each processor for a large number of instructions rather than advancing all processors in lock-step. If your simulation uses TLM2.0 models which rely on lock-step operation you will need to reduce to one the number of instructions which are run in each step.

1.3 Compiling Examples Described in this Document

This documentation is supported by C++ code samples in an `Examples` directory, available as part of an OVPsim installation, by download from the www.ovpworld.org website or as part of an Imperas installation.

The example uses the OR1K processor model and tool chain. The model

is included as part of the OVPsim or Imperas installation. The toolchain is available by free download from the www.ovpworld.org website.

For Windows environments, Imperas recommends using MSYS2 (www.msys2.org).

SystemC TLM2.0 models can be used on Windows with MinGW/MSys (since SystemC release v2.3.0). It is assumed that users of this environment will be familiar with C++, SystemC, TLM2.0 and will have obtained this software from http://accellera.org/downloads/standards/systemc or similar.

All version of SystemC TLM2.0 since v2.3.0 have been used by Imperas, the latest version supported is SystemC release v2.3.3.

2 How CpuManager works with SystemC TLM2.0

2.1 Platform construction

An OVP model is provided with a TLM2.0 interface in the form of a C++ header file. This should be included in the TLM2.0 platform source. It defines a SystemC module class specific to the processor type which can be instantiated in your platform. This class is derived from a generic interface, `tlmProcessor` (itself derived from s*c_module* and the OP C++ class `processor`). OVP peripherals are instantiated in the same way.

If you wish to set global simulator attributes, the `tlmModule` object should be instantiated before any processor or peripheral models.

Before simulation starts, SystemC causes CpuManager to initialize all processor and peripheral models.

2.1.1 Naming

The TLM2.0 interface uses the SystemC method `sc_object::name()` to create a dot-separated hierarchical name, guaranteed to be unique, for the CpuManager instance.

2.2 Processor models

Each processor model is run in a SystemC thread. The thread executes a calculated number of instructions on the processor without advancing SystemC time. Each instruction may or may not cause TLM2.0 transactions to be propagated to other components in the platform. When the allotted instructions have completed, the thread calls SystemC *wait()* to advance time. The processor threads are executed in the order determined by the SystemC scheduler.

2.2.1 Instructions/Sec and Quantum size

To use OVP models, SystemC must instantiate one tlmModule object. This object keeps the quantum period which sets how long each processor model instance waits before running again.

Each processor model instance keeps a figure which controls the effective number of instructions per second (IPS) executed by the model. It uses this and the quantum period to decide how many instructions to run in each quantum.

The default quantum period is 1mS. The default IPS is 100,000,000. Thus, by default, a processor runs 100,000 instructions per quantum (this matches OVPsim's internal scheduler used in a non-SystemC environment).

To change the quantum period use the tlmModule.quantum() method in your platform constructor. The effective frequency of a processor instance is set by the `mips` parameter which is passed to the constructor of the tlmProcessor class.

2.2.2 Guidelines for setting quantum and MIPS

A processor's MIPS should be set to give the correct clock frequency with respect to other models in the simulation. Note that if no other models accurately represent time, then setting the IPS will not affect the behavior of the simulation, merely the reported statistics.

Setting the quantum period is a compromise; a smaller quantum yields a more accurate representation of reality, a larger quantum achieves higher simulation speed.

2.2.2.1 Factors demanding a smaller quantum

To avoid gross functional errors, the quantum period for a processor must be shorter than the shortest time delay modeled in any peripheral device with which the processor interacts.
Similarly, if two processors are communicating using shared memory, the number of instructions per quantum should be less than the number of instructions taken to make each communication.

2.2.2.2 Factors demanding a larger quantum

A short quantum results in poor simulation performance. However, this is only of concern if you intend to simulate many instructions. As a guideline, the SystemC scheduler takes at best a few hundred instructions to start a processor's quantum, so as the instructions per quantum is reduced to this number, the performance will be dominated by the scheduler and not the model.

Scenario	Dominating factor	Quantum
Booting Linux. Functional (not cycle accurate) peripheral models.	Simulation speed of processor model	>= 1mS
Programming a graphics controller. Cycle-accurate GPU.	Simulation speed of GPU	<1uS
Developing a UART driver. Uart has 1mS character rate.	Accuracy of interaction	< 1mS

2.2.3 SystemC Stack Size

The `tlmProcessor` model requires an increased thread stack depth. The function `set_stack_size()` is used to override the default SystemC thread stack size.

2.2.4 Direct Memory Interface Memory Access

TLM2.0 allows comprehensive modeling of bus transactions, but each transaction takes significant simulation time. Direct Memory Interface (DMI) allows negotiation between two TLM2.0 models so that an initiator can directly access target memory, bypassing the TLM2.0 mechanism.

The TLM2.0 OVP interface uses DMI negotiation by default. In practice, a processor with a fixed program memory will execute one code-fetch via TLM2.0. The DMI hint in

that transaction will allow CpuManager to map the program memory into the processor's address space so that subsequent code-fetches do not use TLM2.0 transactions. DMI can be enabled or disabled by calling the `dmi()` method on a processor or memory instance. See section 4.4.3.

Note that a simulator that uses code translation must necessarily cache translated code in the simulator. If the original code memory is subsequently modified by a mechanism outside the simulator, the simulator must be notified so that the code can be re-translated. CpuManager supports the standard DMI invalidation mechanism.

When DMI is turned off on a processor, cached DMI regions are removed for its entire code and data address spaces

When DMI is turned off on a memory, the cached DMI region is removed from connected processors for the region that addresses the memory.

When DMI is turned on again, the models will re-negotiate DMI opportunity.

2.2.5 Simulation artifacts

CpuManager performs dynamic code translation for simulation efficiency. A processor will therefore pre-fetch each code location (up to the next jump or branch) once before it begins executing code. The pre-fetches use the TLM transport_dbg() method rather than the regular transport() method, to distinguish artifact accesses from the real accesses.

For this reason, bus and memory models **must support the transport_dbg() method**. The simplest way to do this is to give the transport() and transport_dbg() methods the same behavior. However if a model counts or otherwise reports bus traffic, it should not do so in the transport_dbg() method.

2.2.6 Delays in bus transactions

The LT (loosely timed) TLM model allows bus transactions to take time. It is legal for a CpuManager processor model to be stalled by a bus transaction that takes time – some part of a bus target calls SystemC wait, which stops the processor model in its bus access. Other scheduled tasks, including other processor models can run while this is happening. If a processor model is used in this way, it is not appropriate to use the supplied TLM processor wrappers – the user will need to write a wrapper that schedules the processor one instruction at a time.

2.3 Peripheral models

During simulation, peripheral models can be activated in three ways:

- TLM2.0 transaction. A TLM2.0 transaction by another model is propagated to this model which results in a bus read or write.
- Elapsed time. Each time SystemC advances time, it notifies CpuManager, which will activate any peripheral waiting for that time to occur.
- Net propagation. A write to a net (implemented using the SystemC *analysis port*) will be propagated by SystemC to any peripheral models connected.

CpuManager also requests notification at the beginning and end of simulation to trigger OVP peripheral model BHM simulation-start and simulation-end special events.

2.3.1 Delays in bus transactions

The LT (loosely timed) TLM model allows bus transactions to take time. It is NOT legal for a CpuManager peripheral model to be stalled by a bus transaction that takes time – CpuManager expects peripheral model code to run in "zero time" – scheduling another model while a peripheral model is stalled will lead to unpredictable behaviour and memory corruption.

2.4 Automatic generation of the TLM interface.

The Imperas Model Generator (igen) can be used to generate TLM interfaces for platforms, processors, MMCs and most peripherals. Please refer to Imperas_Model_Generator_Guide.doc.

3 OVP OP header and source files

The OP API, used by both CpuManager and OVPsim, is defined by several header files within the Imperas tool release tree or download from www.ovpworld.org :

Common Definitions

Standard types `ImpPublic/include/host/impTypes.h`

OP API Definitions

C API functions `ImpPublic/include/host/op/op.h`

C++ API functions `ImpPublic/include/host/op/op.hpp`
`ImpPublic/source/host/op/op.cpp`

OP API Link library

C API functions `bin/<host architecture>/libRuntimeLoader.so`
`bin/<host architecture>/libRuntimeLoader.dll`

C++ API functions[1] `bin/<host architecture>/libRuntimeLoader++.so`
`bin/<host architecture>/libRuntimeLoader++.dll`

OP TLM Archive

`bin/<host architecture>/tlm.a`

OP TLM Source and Headers

Each model in the published component library is provided with a TLM interface.

The model interface class

`ImperasLib/source/<vendor>/<library>/<name>/<vsn>/tlm/<modeltype>.igen.hpp`

Required by a module: **tlmModule class**

```
ImpPublic/source/host/tlm/tlmModule.cpp
ImpPublic/include/host/tlm/tlmModule.hpp
```

Required by a processor interface: **tlmProcessor class**

```
ImpPublic/source/host/tlm/tlmProcessor.cpp
ImpPublic/include/host/tlm/tlmProcessor.hpp
```

Required by a peripheral interface: **tlmPeripheral class**

```
ImpPublic/source/host/tlm/tlmPeripheral.cpp
ImpPublic/include/host/tlm/tlmPeripheral.hpp
```

[1] These are the default libraries; these should be built with a toolchain that is consistent with those that built the SystemC libraries that are being used. See section 3.1.2 OP C++ Interface for building and using a local version.

Required by TLM platforms written by **igen:**

Generic Memory Model

```
ImpPublic/source/host/tlm/tlmDenseMemory.cpp (dense)
ImpPublic/include/host/tlm/tlmDenseMemory.cpp (dense)
ImpPublic/source/host/tlm/tlmMemory.hpp (sparse)
ImpPublic/include host/tlm/tlmMemory.hpp (sparse)
```

Generic Bus Decoder

```
ImpPublic/source/host/tlm/tlmDecoder.cpp
ImpPublic/include/host/tlm/tlmDecoder.hpp
```

Example processor

```
ImperasLib/source/ovpworld.org/processor/or1k/1.0/tlm/processor.igen.hpp
```

Example peripheral

```
ImperasLib/source/national.ovpworld.org/peripheral/16550/1.0/tlm/pse.igen.hpp
```

3.1 Building and Using Locally

When building a SystemC or SystemC TLM platform a C++ compiler/linker is used. The version used must be consistent across all libraries and executables built.

The TLM interface and the OP C++ function interface source are provided so that they may be built locally so that they use a consistent ABI to the SystemC library that is built.

A default version of the OP C++ Library is provided that may be used if compatible with the local SystemC library. If linkage errors occur this must be re-built locally.

The local build of the TLM interface archive and the OP C++ library are controlled by two environment variables, `IMPERAS_TLM_SUPPORT_ARCHIVE` and `IMPERAS_OP_CPP_LIBRARY` respectively. By setting these environment variables to the full path of the TLM archive and the OP C++ library the local versions will be used. They will automatically be built if not already available.

In the next sections we discuss building the TLM archive and OP C++ library locally so that they are compatible with the current user environment.

3.1.1 TLM Archive

The TLM archive is included as part of the build in the Makefile `ImperasLib/buildutils/Makefile.TLM.platform` and is built using the Makefile provided in the directory `ImpPublic/source/host/tlm` as shown below. The Makefile will be copied and the TLM archive built into the local directory as tlmSupport/tlm.a.

If you wish to control the location of the archive or if you wish to use an archive that you have already built the environment variable *IMPERAS_TLM_SUPPORT_ARCHIVE* can

be set to a specific TLM archive, for example

```
> export IMPERAS_TLM_SUPPORT_ARCHIVE=/home/user1/tlmSupport/tlm.a
```

You can build the archive separately:

```
> cp -r ImpPublic/source/host/tlm /home/user1/tlmSupport
> cd /home/user1/tlmSupport
> make
# TLM Compiling Build/Linux32/usr/tlmBusDynamicSlavePort.o
# TLM Compiling Build/Linux32/usr/tlmBusMasterPort.o
# TLM Compiling Build/Linux32/usr/tlmBusSlavePort.o
# TLM Compiling Build/Linux32/usr/tlmDMIslave.o
# TLM Compiling Build/Linux32/usr/tlmDecoder.o
# TLM Compiling Build/Linux32/usr/tlmDenseMemory.o
# TLM Compiling Build/Linux32/usr/tlmMemory.o
# TLM Compiling Build/Linux32/usr/tlmMmc.o
# TLM Compiling Build/Linux32/usr/tlmModule.o
# TLM Compiling Build/Linux32/usr/tlmNetInputPort.o
# TLM Compiling Build/Linux32/usr/tlmPeripheral.o
# TLM Compiling Build/Linux32/usr/tlmProcessor.o
# TLM Archive tlm.a
ar: creating tlm.a
# TLM NM tlm.nm.out
# TLM Support Library tlm.a
```

This creates the archive library tlm.a

The environment variable can be used as part of the main platform Makefile as shown in the following example which will use the example SystemC_TLM.

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/SystemC_TLM .
> cd SystemC_TLM
> export IMPERAS_TLM_SUPPORT_ARCHIVE=/home/user1/tlmSupport/tlm.a
> make -C platform_cpp VERBOSE=1
# iGen Create OP TLM2.0 EXAMPLE PLATFORM platform.cpp
…
# TLM Depending Build/Linux32/usr/platform.d
…
# TLM Compiling Build/Linux32/usr/platform.o
…
# TLM Linking platform.Linux32.exe
g++ -o platform.<>.exe Build/<>/usr/platform.o   /home/user1/tlmSupport/tlm.a
/home/install/systemc-2.3.3/lib-linux/libsystemc.a -L/home/user1/Imperas/bin/<> -
lRuntimeLoader -lisl -lRuntimeLoader++ -lpthread
…
```

As can be seen above the locally built TLM archive is used in the linkage.

3.1.2 OP C++ Interface

The OP C++ interface is built and available in the installation directory `$IMPERAS_HOME/bin/IMPERAS_ARCH` as libRuntimeLoader++ shared object. It is included as part of the build in the Makefile `ImperasLib/buildutils/Makefile.TLM.platform` and is built using the Makefile provided in the directory `ImpPublic/source/host/op` as shown below:

```
> cp -r ImpPublic/source/host/op /home/user1
> cd /home/user1/op
> make
```

```
# C++ Depending Build/Linux32/usr/op.d
# C++ Compiling Build/Linux32/usr/op.o
# C++ Linking libLocalOP++.so
# Created C++ Support Library libOP++.so
```

To use this OP C++ library instead of the default, this library must be specified using the environment variable `IMPERAS_OP_CPP_LIBRARY`.

This is shown in the following example which will use the example SystemC_TLM.

```
> cp $IMPERAS_HOME/Examples/PlatformConstruction/SystemC_TLM .
> export IMPERAS_OP_CPP_LIBRARY=/home/user1/op/libOP++.so
> cd SystemC_TLM
> make -C platform_cpp VERBOSE=1
# iGen Create OP TLM2.0 EXAMPLE PLATFORM platform.cpp
…
# TLM Depending Build/Linux32/usr/platform.d
…
# TLM Compiling Build/Linux32/usr/platform.o
…
# TLM Linking platform.Linux32.exe
g++ -o platform.Linux32.exe Build/Linux32/usr/platform.o
<IMPERAS_HOME>/bin/Linux32/tlm.a <SYSTEMC_HOME>/systemc-2.3.3/lib-linux/libsystemc.a -
L<IMPERAS_HOME>/bin/Linux32 -lRuntimeLoader -lisl -m32 -L/home/user1/op -lOP++ -lpthread
```

When the platform was built using this OP C++ library is executed, the location of the library must be specified on Windows by adding the directory to the PATH, and on Linux by adding the directory o *LD_LIBRARY_PATH*.

On Linux *LD_LIBRARY_PATH* environment variable should be set

```
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/user1/op
```

And on Windows the directory should be included in the *PATH* environment variable.

```
> set PATH=%PATH%;C:\home\user1\op
```

Or in an MSYS shell

```
> export PATH=$PATH:/home/user1/op
```

4 Example Platform

An example TLM2.0 platform is provided in
`Examples/PlatformConstruction/SystemC_TLM`

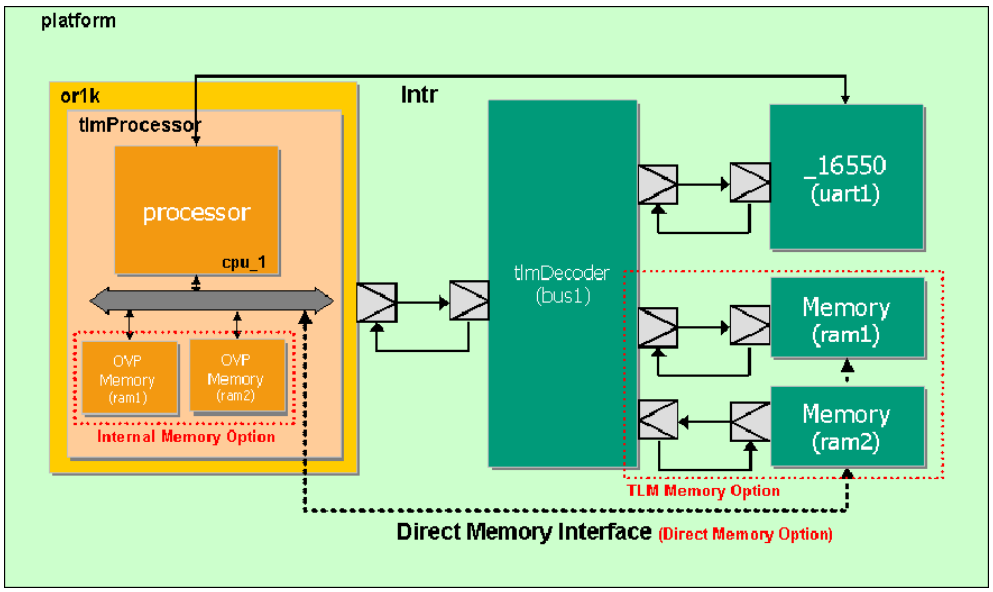

Figure 1: Example TLM2.0 Platform Block Diagram

The source files for this example are in
`Examples/PlatformConstruction/SystemC_TLM/platform_cpp`

The complete example platform has all memory mapped to TLM memory. At runtime this will be accessed using DMI, if possible.

The platform-specific code is in the included file `platform.sc_constructor.cpp`:

The model header files are included in the platform header:

```
#include "ovpworld.org/modelSupport/tlmDecoder/1.0/tlm/tlmDecoder.hpp"
#include "ovpworld.org/memory/ram/1.0/tlm/tlmMemory.hpp"
#include "ovpworld.org/processor/or1k/1.0/tlm/processor.igen.hpp"
#include "national.ovpworld.org/peripheral/16550/1.0/tlm/pse.igen.hpp"
```

In the platform class, platform components are instantiated (template parameters are supplied to template components).

```
class simple : public sc_core::sc_module {

  public:
    simple (sc_core::sc_module_name name);

    tlmModule            Platform;
    tlmDecoder            bus1;
    tlmRam                ram1;
    tlmRam                ram2;
    or1k                  cpu1;
    _16550                uart1;
```

```
    params paramsForuart1() {
        params p;
        p.set("outfile", "uart1.log");
        return p;
    }

    params paramsForcpu1() {
        params p;
        p.set("defaultsemihost", true);
        return p;
    }
}; /* simple */
```

The `paramsForuart1()` function specifies configuration parameters for the uart model, in this case defining a file in which the uart's output is to be written.

The `paramsForcpu1()` function specifies configuration parameters for the processor model, in this case turning on the default semihost library to intercept system calls, sending text output to the simulator's output stream.

Calls to the sub-constructors are put before the body of the constructor.

```
simple::simple ( sc_core::sc_module_name name)
    : sc_module (name, paramsForRoot())
    , Platform ("")
    , bus1(Platform, "bus1", 2, 3)
    , ram1 (Platform, "ram1", 0xfffff)
    , ram2 (Platform, "ram2", 0xffff)
    , cpu1 ( Platform, "cpu1",  paramsForcpu1())
    , uart1 (Platform, "uart1", paramsForuart1())
{
```

Connections (binding) between components occur in the body of the constructor.

```
    // bus1 masters
    bus1.connect(cpu1.INSTRUCTION);
    bus1.connect(cpu1.DATA);

    // bus1 slaves
    bus1.connect(uart1.bport1, 0x90000000, 0x90000007); // Peripheral  (0)

    bus1.connect(ram1.sp1,0x0, 0xfffff); // Memory (1)

    bus1.connect(ram2.sp1,0xffff0000, 0xffffffff); // Memory (2)

    // Net connections
    uart1.intOut(cpu1.intr0);
```

The file `platform.cpp` defines `sc_main` which instances the simulator `session`, the standard command line `parser` and the platform class `simple`:

```
...
int sc_main (int argc, char *argv[] ) {
    session s;
    parser  p(argc, (const char**) argv);

    simple simple("simple");

    sc_start();
    return 0;
```

```
}
```

The TLM interface `tlmProcessor` is derived from the OP C++ `processor` class so that all methods of this class can be applied to the TLM processor instantiation. The `tlmPeripheral` is derived from OP C++ `peripheral` so you can similarly use its methods.

The SystemC library provides `main`() that calls `sc_main()` after the constructors and starts the SystemC scheduler.

4.1 Compilation

The above example[2] was compiled under Windows using MinGW/MSys[3] and on Linux using GCC.

To build the example, follow these steps:
On Windows

- Obtain and install MinGW/MSys

On Windows and Linux

- Obtain and install SystemC v2.3 source or above that support MinGW build
- Obtain and install the OR1K tool-chain (this example uses an or1k processor model)
- Obtain and install OVPsim or the Imperas professional tools and configure the Imperas environment as described in the Installation guide

All compilation is performed in the Linux or MinGW/MSys command shell

```
> cd <temp directory>
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/SystemC_TLM .
> cd SystemC_TLM/platform_cpp
```

Specify the locations of your SystemC and TLM2.0 releases. For Example

```
> export SYSTEMC_HOME=C:/SystemC/systemc-2.3.3
```

Compile the example platform and interfaces:

The Makefile includes standard Makefiles[4] that provide rules and targets.

```
# Common variables
include $(IMPERAS_HOME)/ImperasLib/buildutils/Makefile.common
# Target and Rules C to executable
include $(IMPERAS_HOME)/ImperasLib/buildutils/Makefile.TLM.platform
# Target and Rules iGen to C
include $(IMPERAS_HOME)/ImperasLib/buildutils/Makefile.TLM.igen
```

[2] The old ICM API example IMPERAS_HOME/Examples/PlatformsICM/SystemC_TLM2.0 includes the batch file platform_cpp/compile.msvc.bat that uses nmake in an MSVC command prompt to build the platform. This is no longer supported with the usage of MSYS/MinGW.
[3] Since SystemC release v2.3.0 support to build with MinGW on Windows has been included. At this time OVP moved all SystemC TLM2 examples and demos from building with MSVC to building with MinGW.
[4] The order of inclusion is important

Makefile.common	: Defines common variables
Makefile.TLM.platform	: Provides main target and rules to build executable from C source files
Makefile.TLM.igen	: Provides target and rules to generate C source from iGen input files

To build:

```
> make
# iGen Create OP TLM2.0 EXAMPLE PLATFORM platform.cpp
# TLM Depending Build/Linux32/usr/platform.d
# TLM Compiling Build/Linux32/usr/platform.o
# TLM Linking platform.Linux32.exe
```

To see the commands that are being performed, use VERBOSE=1

```
> make realclean
> make VERBOSE=1
# iGen Create OP TLM2.0 EXAMPLE PLATFORM platform.cpp
igen.exe --quiet --nobanner --excludem GPT_NH --excludem GPT_UFNR  \
        --op \
        --batch    platform.tlm.tcl \
        --writetlm platform.cpp \
        --userheader <IMPERAS_HOME>/ImperasLib/fileheaders/refESLA.txt \
        --overwrite
# TLM Depending Build/Linux32/usr/platform.d
mkdir -p Build/Linux32/usr
g++ -MM platform.cpp -I<IMPERAS_HOME>/ImpPublic/include/host -I<SYSTEMC_HOME>/include -
I<IMPERAS_HOME>/ImperasLib/source  -ggdb -Wno-long-long -Wall -Werror -
DSC_INCLUDE_DYNAMIC_PROCESSES -D_CRT_SECURE_NO_WARNINGS -D_CRT_SECURE_NO_DEPRECATE -
I<IMPERAS_HOME>/ImpPublic/include/host -I<IMPERAS_HOME>/ImpProprietary/include/host -Wall
-Werror -O0 -g -gdwarf-2 -m32  -D_GLIBCXX_USE_CXX11_ABI=0  -MT
Build/Linux32/usr/platform.o -MF Build/Linux32/usr/platform.d
# TLM Compiling Build/Linux32/usr/platform.o
mkdir -p Build/Linux32/usr
g++ -c -o Build/Linux32/usr/platform.o platform.cpp -
I<IMPERAS_HOME>/ImpPublic/include/host -I<SYSTEMC_HOME>/include -
I<IMPERAS_HOME>/ImperasLib/source  -ggdb -Wno-long-long -Wall -Werror -
DSC_INCLUDE_DYNAMIC_PROCESSES -D_CRT_SECURE_NO_WARNINGS -D_CRT_SECURE_NO_DEPRECATE -
I<IMPERAS_HOME>/ImpPublic/include/host -I<IMPERAS_HOME>/ImpProprietary/include/host -Wall
-Werror -O0 -g -gdwarf-2 -m32  -D_GLIBCXX_USE_CXX11_ABI=0
# TLM Linking platform.Linux32.exe
mkdir -p .
g++ -o platform.Linux32.exe Build/Linux32/usr/platform.o   -L<IMPERAS_HOME>/bin/Linux32 -
lRuntimeLoader -lisl -m32 <SYSTEMC_HOME>/lib-linux/libsystemc.a  -
L<IMPERAS_HOME>/bin/Linux32 -lRuntimeLoader -lisl -m32 -lRuntimeLoader++ -lpthread
test -f /usr/bin/execstack; \
        if [ $? -eq 0 ]; then \
            execstack --clear-execstack platform.Linux32.exe; \
        fi
```

4.2 Building an application

A toolchain to allow an application to be cross compiled for the OR1K processor can be obtained from www.ovpworld.org.

The application code cross compilation is supported for a MINGW shell on Windows. Refer to the document “OVPsim_Installation_and_Getting_Started” for installation and use information.

To build the application in a Linux or in an MSYS shell on Windows, follow these steps:

- Go to a copy of the application directory in the example
- Execute the provided Makefile

On Windows:

```
> cd ../application
> make
```

On Linux:

```
> cd ../application
> make
> cd ..
```

This will build the executable *int.OR1K.elf*

4.3 Running a platform

Run the platform, passing the application program:

```
platform_cpp/platform.$IMPERAS_ARCH.exe –program application/int.elf
```

The output from the run should be:

```
                    SystemC 2.3.3-Accellera --- Sep 30 2020 09:02:20
                      Copyright (c) 1996-2018 by all Contributors,
                                  ALL RIGHTS RESERVED
TEST: main starts
TEST: Initialize:
TEST: Enable UART:
Interrupt Handler 0x02 (1)
Character sent
TEST: main send string
TEST: Send String: Hello World

Send char H (0x48)
Interrupt Handler 0x02 (2)
Character sent
Send char e (0x65)
Interrupt Handler 0x02 (3)
... etc ...
Character sent
Send char l (0x6c)
Interrupt Handler 0x02 (11)
Character sent
Send char d (0x64)
Interrupt Handler 0x02 (12)
Character sent
Send char
 (0x0a)
Interrupt Handler 0x02 (13)
Character sent
TEST: main done
```

4.4 Platform Construction Options

OVP models can be instantiated in a TLM2.0 platform exactly like other models. However, there are many features in CpuManager which are available through the TLM2.0 interface. Some commonly used features are listed here. For details, refer to OVPsim_and_CpuManager_User_Guide.doc. The code examples are given in the context of the worked example platform.

4.4.1 Processor Options

4.4.1.1 Setting a variant

By default, a processor model will execute as an Instruction Set Architecture (ISA) model which represents the instructions but not necessarily the configuration of the processor. The `variant` parameter is used to select a different configuration; refer to the processor model specific documentation for the list of supported variants.

In the example, parameters can be set in the function paramsForcpu1 (note that the or1k processor model does not have any variants):

```
    params paramsForcpu1() {
        params p;
        p.set("defaultsemihost", true);
        p.set("variant", “variant1”);
        return p;
    }
```

4.4.1.2 Instruction Tracing.

Instruction tracing can be enabled from the simulator command line:

```
> platform_cpp/platform.$IMPERAS_ARCH.exe \
    –program application/int.elf
    -trace
```

Tracing can also be enabled by setting a parameter on the model:

```
    params paramsForcpu1() {
        params p;
        p.set("defaultsemihost", true);
        p.set("trace", true);
        return p;
   }
```

4.4.1.3 Application debug

The or1k processor shipped with this OVPsim release is provided with a (rather old) GNU gdb debugger. This can be started and connected to the or1k processor model from the simulator command line:

```
> platform_cpp/platform.$IMPERAS_ARCH.exe \
    –program application/int.elf \
    -gdbconsole
```

This will pop up a console containing a gdb already connected to the simulator.

Like all command line flags, the same effect can be achieved by setting a parameter, this time on the root module:

```
    params paramsForRoot() {
        params p;
        p.set("gdbconsole", true);
        return p;
    }

simple::simple ( sc_core::sc_module_name name)
    : sc_module (name, paramsForRoot())
    , Platform ("")
    , bus1(Platform, "bus1", 2, 3)
    , ram1 (Platform, "ram1", 0xfffff)
    , ram2 (Platform, "ram2", 0xffff)
    , cpu1 ( Platform, "cpu1",  paramsForcpu1())
    , uart1 (Platform, "uart1", paramsForuart1())
{
   ...
}
```

4.4.1.4 Setting the simulation time slice (quantum)

The quantum is set for the whole platform. All processor models use this value. See ImperasLib/source/ovpworld.org/modelSupport/tlmPlatform/1.0/tlm/tlmPlatform.hpp.

```
simple:: simple (sc_module_name name)
    : Platform(“simple”, …..)
{
    Platform.setSimulationTimeSlice(0.0001);
}
```

4.4.1.5 Simulated Exceptions

By default an OVP processor model will notify the simulator if an exception (e.g. divide by zero or access alignment error) occurs. Set the `simulateexceptions` parameter to make the processor jump to its exception vector instead. This option is typically used when simulating an operating system. It is not used in a "bare metal" platform which has no code to handle the exception. See op.h and search for OP_FP_.

```
     params paramsForcpu1() {
        params p;
        p.set("simulateexceptions", true);
        return p;
    }

simple::simple ( sc_core::sc_module_name name)
    : sc_module (name, paramsForRoot())
    , Platform ("")
    , bus1(Platform, "bus1", 2, 3)
    , ram1 (Platform, "ram1", 0xfffff)
    , ram2 (Platform, "ram2", 0xffff)
    , cpu1 ( Platform, "cpu1",  paramsForcpu1())
    , uart1 (Platform, "uart1", paramsForuart1())
{
   ...
}
```

4.4.1.6 Loading intercept libraries

We have already seen that this example uses the default intercept library for the or1k processor. Commented-out code can be included to load a library.

```
    ( in the class definition )
    // extension            ext1;

    ( a private function in the platform class )
    const char *getExt1Path(void) {
       return opVLNVString(
           0,
           "ovpworld.org",
           "semihosting",
           "or1kNewlib",
           "1.0",
           OP_EXTENSION,
           true
        );
    }

    ( in the platform constructor )
    ,ext1  (cpu1, getExt1Path(), "ext1")
```

4.4.2 Peripheral Options

4.4.2.1 Peripheral diagnostics.

Most peripherals are capable of producing diagnostic output with different levels of detail. Call the PSE method diagnosticLevelSet (). See bhm.h : bhmSetDiagnosticLevel().

```
uart1. diagnosticLevelSet (3);
```

4.4.3 DMI

As mentioned previously, OVP processor and memory models have DMI enabled by default. To turn DMI on or off use the dmi method on the memory or processor models, as illustrated in commented code in the platform constructor:

```
simple::simple (sc_module_name name)
    : sc_module (name)
    , Platform ("", paramsForRoot())
    , bus1(Platform, "bus1", 2, 3)
    , ram1 (Platform, "ram1", 0xfffff)
    , ram2 (Platform, "ram2", 0xffff)
    , cpu1 ( Platform, "cpu1",  paramsForcpu1())
    , uart1 (Platform, "uart1", paramsForuart1())
{
    bus1.connect(cpu1.INSTRUCTION);
    bus1.connect(cpu1.DATA);
    bus1.connect(uart1.bport1, 0x90000000, 0x90000007); // Peripheral  (0)
    bus1.connect(ram1.sp1,0x0, 0xfffff); // Memory (1)
    bus1.connect(ram2.sp1,0xffff0000, 0xffffffff); // Memory (2)
    uart1.intOut(cpu1.intr0);

    // By default DMI is turned on. Use these to turn it off
    //ram1.dmi(0);
    //ram2.dmi(0);
    //cpu1.dmi(0);
```

```
}
```

5 Deviations from TLM2.0 LRM

5.1 Data Endian in TLM transactions

The contents of the data field in a TLM transaction is **target endian** rather than **host endian** as specified. This is due to an inconsistency in the TLM standard which makes efficient DMI otherwise hard to achieve.

5.2 Modeling of Interrupts

To model interrupt signals, the OVP interface to TLM2.0 uses the TLM analysis port rather than the SystemC net. The analysis port immediately propagates new values by function call, whereas the SystemC net requires the SystemC scheduler to cause propagation. As explained elsewhere in this document, for efficiency, the OVP processor model typically simulates thousands of instructions in one 'quantum' without the intervention of the SystemC scheduler. If these instructions change the value of an interrupt net, the effect of the change (on an interrupt controller for example) will not be seen until the end of the quantum when the SystemC scheduler is allowed to run. The delayed change will be unrealistic. TLM analysis port dos not suffer this delay so is often used for this reason.

6 Tracing TLM activity

The OVPsim TLM2.0 interface can generate messages, controlled globally by setting environment variables, or locally by calling methods on the models. Set the environment variables to a non-null value or refer to the model header files (described in this document) for the method prototypes.

Model	Environment variable	Method	Trace
Processor	IMPERAS_TLM_CPU_TRACE	traceQuanta	start of each time slice
		traceBuses	each bus transaction
		traceBusErrors	each incomplete bus transaction
		traceSignals	signal value changes
Peripheral	IMPERAS_TLM_PSE_TRACE	traceBuses	each bus transaction
		traceBusErrors	each incomplete bus transaction
		traceSignals	signal value changes
MMC	IMPERAS_TLM_MMC_TRACE	traceMasters	each bus master transaction
		traceSlaves	each bus slave transaction

Note that when DMI (Direct Memory Interface) is enabled (which is by default on all OVP processors), there are no transactions to be traced.

Tracing of bus transactions is verbose and will reduce simulation performance.

7 What can go wrong

The following are a list of problems that can be encountered while building or running a TLM platform:

7.1 Spaces in filenames

nmake and other MSVC tools will not accept spaces in file-names.
Either install OVPsim, SystemC and TLM in a path without spaces (ie not "Program Files") or ensure that all paths are enclosed in double-quotes.

7.2 OVPsim version incompatibilities

Ensure that library, models and TLM2.0 interfaces are from the same version of OVPsim.

7.3 Environment problems

Check the values of environment variables:

- SYSTEMC
- TLM_HOME
- IMPERAS_VLNV,
- IMPERAS_HOME
- PATH

7.4 Compiling OSCI SystemC 2.2.0 with later versions of gcc

The OSCI SystemC simulator is available as source from http://accellera.org/downloads/standards/systemc and may be used with OVPsim, but when compiling the version 2.2.0 source with the latest version of GCC the following errors may be encountered:

```
../../../../src/sysc/utils/sc_utils_ids.cpp: In function \u2018int
sc_core::initialize()\u2019:
../../../../src/sysc/utils/sc_utils_ids.cpp:110: error:
\u2018getenv\u2019 is not a member of \u2018std\u2019
../../../../src/sysc/utils/sc_utils_ids.cpp:111: error:
\u2018strcmp\u2019 was not declared in this scope
../../../../src/sysc/utils/sc_utils_ids.cpp: At global scope:
../../../../src/sysc/utils/sc_utils_ids.cpp:119: warning:
\u2018sc_core::forty_two\u2019 defined but not used
```

The solution is to add the following includes to the file `systemc-2.2.0/src/sysc/utils/sc_utils_ids.cpp`:

```
#include "string.h"
#include "cstdlib"
```

Additionally, when compiling the OVP tlm modules you may see the following errors:

```
In file included from
/systemc-2.2.0/include/sysc/datatypes/bit/sc_lv.h:49,
from /systemc-2.2.0/include/sysc/communication/sc_signal_rv.h:61,
from /systemc-2.2.0/include/systemc:74,
from /TLM-2008-06-09/include/tlm/tlm.h:21,
from
/Imperas/ImperasLib/source/ovpworld.org/modelSupport/tlmMMC/1.0/tlm2.0/t
lmMmc.hpp:25,
from
/Imperas/ImperasLib/source/ovpworld.org/modelSupport/tlmMMC/1.0/tlm2.0/t
lmMmc.cpp:21:
/systemc-2.2.0/include/sysc/datatypes/bit/sc_lv_base.h: In member
function \u2018sc_dt::sc_logic_value_t sc_dt::sc_lv_base::get_bit(int)
const\u2019:
/systemc-2.2.0/include/sysc/datatypes/bit/sc_lv_base.h:310: error:
suggest parentheses around arithmetic in operand of \u2018|\u2019
/systemc-2.2.0/include/sysc/packages/boost/bind/placeholders.hpp: At
global scope:
/systemc-2.2.0/include/sysc/packages/boost/bind/placeholders.hpp:54:
error: \u2018<unnamed>::_1\u2019 defined but not used
/systemc-2.2.0/include/sysc/packages/boost/bind/placeholders.hpp:55:
error: \u2018<unnamed>::_2\u2019 defined but not used
...
```

The 'defined but not used' errors are due to an old version of Boost used in the SystemC source. To correct that problem the file `systemc-2.2.0/src/sysc/packages/boost/bind/placeholders.hpp` may be edited and line 28 changed from:

```
#if defined(__BORLANDC__)
```

to:

```
#if defined(__BORLANDC__) || defined(__GNUC__)
```

The warnings about parenthesis can be fixed by editing the make file to remove `-Werror` so that the warnings do not stop the compilation or by editing the indicated file and adding parenthesis where indicated.

##