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:	1.13.6
Filename:	OVPsim_Using_OVP_Models_in_SystemC_TLM2.0_Platforms.doc
Project:	Using OVP Models in SystemC TLM2.0 Platforms
Last Saved:	Thursday, 02 April 2015
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.

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 all the ICM interface functions described in "OVPsim and CpuManager User Guide". The ICM 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.

GCC Compiler Versions

Linux32	4.5.2	i686-nptl-linux-gnu (Crosstool-ng)
Linux64	4.4.3	x86_64-unknown-linux-gnu (Crosstool-ng)
Windows32	4.4.7	mingw-w32-bin_i686-mingw
Windows64	4.4.7	mingw-w64-bin_i686-mingw

For Windows environments, Imperas recommends using MinGW (www.mingw.org) and MSYS.

SystemC TLM2.0 models can be used on Windows with MSVC 8.0 and MinGW/MSys (since SystemC release v2.3.0). It is assumed that users of this environment with be familiar with C++, SystemC, TLM2.0 and will have obtained this software from www.systemc.org or similar.

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, `icmCPU` (itself derived from s*c*`_module`). OVP peripherals are instantiated in the same way.

If you wish to set global simulator attributes, the `icmTLMPlatform` 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 from a SystemC thread. The thread executes *IPQ* 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. Thus each processor executes a number of instructions at a time in a round-robin schedule.

2.2.1 Instructions/Sec and Quantum size

To use OVP models, SystemC must instantiate one tlmPlatform 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 icmTLMPlatform.quantum() method in your platform constructor. To change the effective frequency of a processor instance use the icmCpu.setIPS() method. See 4.4.1.5 for examples of changing these values.

2.2.2 Guidelines for setting quantum and IPS

A processor's IPS 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 behaviour 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 `icmCpu` 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 adds a significant time overhead to the simulation. 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 its address space so that subsequent code-fetches do not use TLM2.0 transactions. DMI can be disabled with an optional argument to the CPU model constructor.

DMI invalidation is provided to support the modification of TLM2.0 memory by an external source.

Processor and memory models can individually start or stop DMI during the simulation.

```
processor_instance.dmi(0);   // turns off DMI for this processor.
processor_instance.dmi(1);   // turns on DMI for this processor.

memory_instance.dmi(0);      // turns off DMI for this memory.
memory_instance.dmi(1);      // turns on DMI for this memory.
```

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 at the earliest 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_gdb() method**. The simplest way to do this is to give the transport() and transport_gdb() methods the same behaviour. However if a model counts or otherwise reports bus traffic, it should not do so in the transport_dbg() method.

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.

2.4 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 ICM header and source files

The ICM 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`

ICM API Definitions

Formatted output `ImpPublic/include/host/icm/icmText.h`
C API function `ImpPublic/include/host/icm/icmCpuManager.h`
C++ API functions `ImpPublic/include/host/icm/icmCpuManager.hpp`

ICM TLM2.0 Headers

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

The model interface class

```
ImperasLib/source/<vendor>/<lib>/<name>/<vsn>/tlm2.0/<modeltype>.igen.hpp
```

Required by a platform: **icmTLMPlatform class**

```
ImperasLib/source/ovpworld.org/modelSupport/tlmPlatform/1.0/tlm2.0/tlmPlatform.cpp
ImperasLib/source/ovpworld.org/modelSupport/tlmPlatform/1.0/tlm2.0/tlmPlatform.hpp
```

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

```
ImperasLib/source/ovpworld.org/modelSupport/tlmProcessor/1.0/tlm2.0/tlmProcessor.cpp
ImperasLib/source/ovpworld.org/modelSupport/tlmProcessor/1.0/tlm2.0/tlmProcessor.hpp
```

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

```
ImperasLib/source/ovpworld.org/modelSupport/tlmPeripheral/1.0/tlm2.0/tlmPeripheral.cpp
ImperasLib/source/ovpworld.org/modelSupport/tlmPeripheral/1.0/tlm2.0/tlmPeripheral.hpp
```

Required by an MMC interface: **icmMMC class**

```
ImperasLib/source/ovpworld.org/modelSupport/tlmMMC/1.0/tlm2.0/tlmMmc.cpp
```

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

Generic Memory Model

```
ImperasLib/source/ovpworld.org/memory/ram/1.0/tlm2.0/tlmMemory.cpp
ImperasLib/source/ovpworld.org/memory/ram/1.0/tlm2.0/tlmMemory.hpp
```

Generic Bus Decoder

```
ImperasLib/source/ovpworld.org/modelSupport/tlmDecoder/1.0/tlm2.0/tlmDecoder.hpp
```

Example processor
```
ImperasLib/source/ovpworld.org/processor/or1k/1.0/tlm2.0/processor.igen.hpp
```

Example peripherals
```
ImperasLib/source/national.ovpworld.org/peripheral/16550/1.0/tlm2.0/pse.igen.hpp
ImperasLib/source/ovpworld.org/peripheral/SimpleDma/1.0/tlm2.0/pse.igen.hpp
```

4 Example Platform

An example TLM2.0 platform is provided in the Examples/Platforms/SystemC_TLM2.0 directory illustrating the use of an OVP processor model in a TLM2.0 platform.

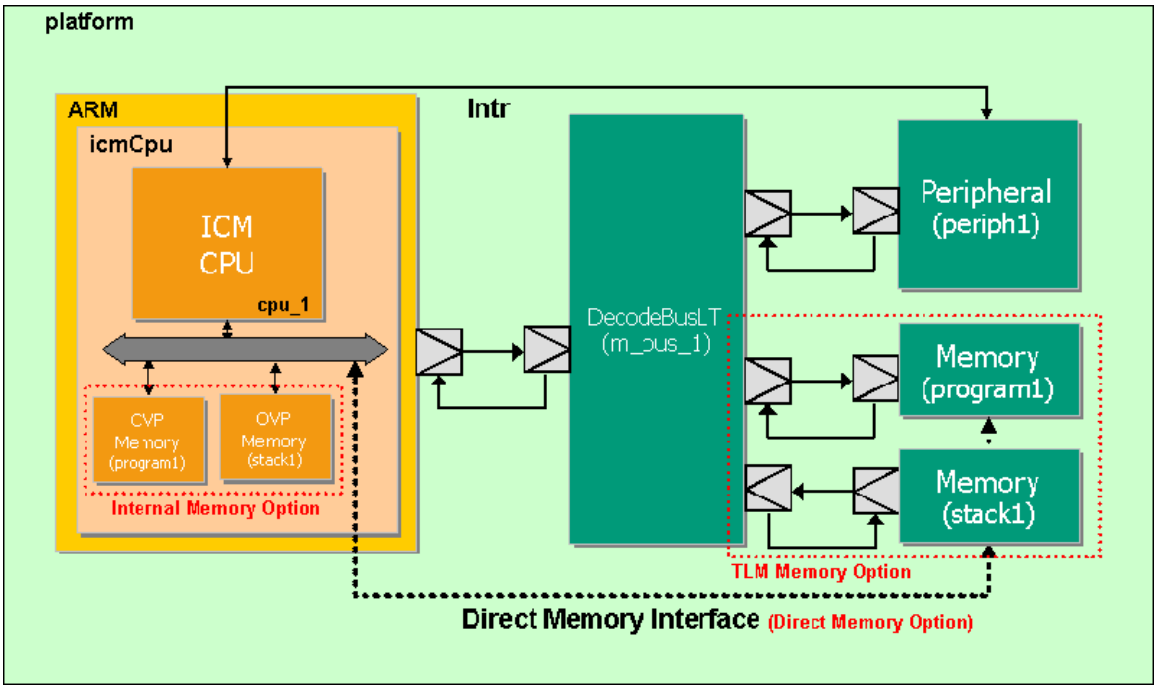

Figure 1: Example TLM2.0 Platform Block Diagram

The source files for this example are in
`Examples/Platforms/SystemC_TLM2.0/platform_cpp/platform.cpp`

The complete example platform has all memory mapped to TLM2.0 memory. At runtime this will be accessed using DMI, if capable.

The model header files are included in the platform header:

```
#include "tlm.h"
#include "ovpworld.org/modelSupport/tlmPlatform/1.0/tlm2.0/platform.hpp"
#include "ovpworld.org/modelSupport/tlmDecoder/1.0/tlm2.0/decoder.hpp"
#include "ovpworld.org/memory/ram/1.0/tlm2.0/memory.hpp"
#include "ovpworld.org/processor/or1k/1.0/tlm2.0/processor.igen.hpp"
#include "national.ovpworld.org/peripheral/16550/1.0/tlm2.0/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);

    icmTLMPlatform        Platform;
    decoder              <2,3> bus1;
    ram                   ram1;
```

```
    ram                 ram2;
    or1k                cpu1;
    _16550              uart1;


    icmAttrListObject *attrsForuart1() {
        icmAttrListObject *userAttrs = new icmAttrListObject;
        userAttrs->addAttr("outfile", "uart1.log");
        return userAttrs;
    }
}; /* simple */
```

The `attrsForuart1()` function specifies configuration attributes (aka parameters) for the uart peripheral, defining a file in which the uart's output is to be written.

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

```
simple::simple ( sc_core::sc_module_name name)
    : sc_core::sc_module (name)
    , Platform ("icm", ICM_VERBOSE | ICM_STOP_ON_CTRLC| ICM_ENABLE_IMPERAS_INTERCEPTS )
    , bus1("bus1")
    , ram1 ("ram1", "sp1", 0x100000)
    , ram2 ("ram2", "sp1", 0x10000)
    , cpu1 ( "cpu1", 0    )
    , uart1 ("uart1", attrsForuart1())
{
```

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

```
    // bus1 masters
    cpu1.INSTRUCTION.socket(bus1.target_socket[0]);
    cpu1.DATA.socket(bus1.target_socket[1]);

    // bus1 slaves
    bus1.initiator_socket[0](uart1.bport1.socket); // Peripheral
    bus1.setDecode(0, 0x90000000, 0x90000007);

    bus1.initiator_socket[1](ram1.sp1); // Memory
    bus1.setDecode(1, 0x0, 0xfffff);

    bus1.initiator_socket[2](ram2.sp1); // Memory
    bus1.setDecode(2, 0xffff0000, 0xffffffff);


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

In `sc_main` the platform class in instantiated and default parameters are overridden:

```
...
int sc_main (int argc, char *argv[] ) {
...
    simple top("top");
    top.cpu1.setIPS(100000);       // 1000KHz

    top.uart1.setDiagnosticLevel(1);  // Set diagnostic level for Uart
...
```

The TLM2.0 interface `icmCpu` is derived from the icmProcessorObject class so that all methods of this class can be applied to the TLM2.0 processor instantiation. The `icmPeripheral` is derived from `icmPseObject` so you can similarly use its methods.

The application specified on the command line is loaded into the processor's memory using the `loadLocalMemory()` method. The `icmLoaderAttrs` value `ICM_SET_START` causes the processor's PC to be preloaded with the object file entry point, rather than the hardware reset vector:

```
    const char *exe;

    if (argc == 2) {
        exe = argv[1];
    } else {
        cout << "Usage: platform.exe <application>" << endl;
        cout << "       Please specify application" << endl;
        return 0;
    }
...
    top.cpu1.loadLocalMemory(exe, (icmLoaderAttrs)(ICM_LOAD_VERBOSE | ICM_SET_START));
...
```

The reset input of the cpu is asserted and simulation run for 1 uSec, then de-asserted and run to completion:

```
...
    cout << "Starting sc_main (reset asserted)." << endl;
    top.cpu1.reset.write(1);
    sc_core::sc_start(1, SC_US);  // Run the simulation for 1 uS with reset asserted

    cout << "*** De-assert reset. ***\n" << endl;
    top.cpu1.reset.write(0);
    sc_core::sc_start();          // Run to end of simulation with reset de-asserted
...
```

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

4.1 Compilation

The above example was compiled under Windows using MinGW/MSys[1] and on Linux using GCC. It also includes information on building with MSVC[2].

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

[1] 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.
[2] The example mentioned includes the batch file platform_cpp/compile.msvc.bat that uses nmake in an MSVC command prompt to build the platform.

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

```
> cd <temp directory>
> cp -r $IMPERAS_HOME/Examples/Platforms/SystemC_TLM2.0 .
> cd SystemC_TLM2.0/platform_cpp
```

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

```
> export SYSTEMC_HOME=C:/SystemC/systemc-2.3.0
```

Compile the example platform and interfaces:

On Windows:
```
> mingw32-make -f $IMPERAS_HOME/ImperasLib/buildutils/Makefile.TLM.platform
```

On Linux:
```
> make -f $IMPERAS_HOME/ImperasLib/buildutils/Makefile.TLM.platform
```

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 MINGW shell, follow these steps:

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

On Windows:
```
> cd ../application
> mingw32-make
```

On Linux:
```
> cd ../application
> make
```

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

4.3 Running a platform

Run the platform giving the program as the first argument:

```
cd ..
platform_cpp/platform.$IMPERAS_ARCH.exe application/int.elf
```

The output from the run should be:

```
Constructing.
...
Info (ICM_AL) Found attribute symbol 'modelAttrs' in file
'.../Imperas/lib/Linux32/ImperasLib/ovpworld.org/semihosting/or1kNewlib/1.0/model.so'
Info (ICM_AL) Found attribute symbol 'modelAttrs' in file
'.../Imperas/lib/Linux32/ImperasLib/ovpworld.org/processor/or1k/1.0/model.so'
Using SystemC TLM2.0 Memory with DMI
Info (OR_OF) Target 'top.cpu1' has object file read from './application/int.OR1K.elf'
Info (OR_PH) Program Headers:
Info (OR_PH) Type           Offset     VirtAddr   PhysAddr   FileSiz    MemSiz     Flags
Align
Info (OR_PD) LOAD           0x00002000 0x00000000 0x00000000 0x0000cfcc 0x0000cfcc R-E
2000
Info (OR_PD) LOAD           0x0000efcc 0x0000efcc 0x0000efcc 0x00000868 0x00000874 RW-
2000
Starting sc_main (reset asserted).
Info (16550_BRS) top.uart1: baud rate=1152000  parity=N  data bits=5  total bits=7
character delay=6usec
*** De-assert reset. ***

TEST: main starts
TEST: Initialize:
TEST: Enable UART:
TEST: main send string
TEST: Send String: Hello World

Send char H (0x48)
Interrupt Handler 0x02 (1)
Character sent
Send char e (0x65)
Interrupt Handler 0x02 (2)
Character sent
...
Character sent
Send char d (0x64)
Interrupt Handler 0x02 (11)
Character sent
Send char
 (0x0a)
Interrupt Handler 0x02 (12)
Character sent
TEST: main done

Info: /OSCI/SystemC: Simulation stopped by user.
Finished sc_main.
...
```

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; that is it represents the instructions but not necessarily the configuration of the processor

A default processor model attribute function is provided in the processor TLM2.0 interface. If the configuration is to be changed an alternative function can be defined in the platform class definition.

This is then used in the processor instantiation

```
    , cpu1 ( "cpu1", 0, ICM_ATTR_DEFAULT, attrsForcpu1())
```

4.4.1.2 Instruction Tracing.

This option prints a line for every instruction executed. Add the ICM_ATTR_TRACE to the processor constructor. See `icmCpuManager.h` : `icmNewProcAttrs`

```
cpu1 ("cpu1", 1, ICM_ATTR_TRACE );
```

4.4.1.3 Instruction tracing with labels.

This option cross-references code labels found in the object file to addresses printed in each traced instruction. This is a global option to all processors being traced. Add `ICM_DISASSEMBLE_LABELS` to the platform constructor. See `icmCpuManager.h` : icmInitAttrs

```
Platform ("icm", ICM_VERBOSE | ICM_DISASSEMBLE_LABELS);
```

4.4.1.4 Enabling debugging and selecting processor to debug

The ICM kernel is initialized by the `icmPlatform` constructor:

```
  icmPlatform(
     const char  *name,
     icmInitAttrs simAttrs  = ICM_INIT_DEFAULT,
     const char  *protocol  = 0,
     Uns32        port      = 0
   );
```

The third and fourth arguments of the platform constructor are used for single processor debugging:

```
   , Platform ("icm", ICM_VERBOSE ,"rsp", 0 )
```

GDB Remote Serial Protocol (RSP) debugging as supported by OVPsim uses standard operating system sockets on the host running OVPsim and GDB.

If a NULL value is given for *protocol*, debugging is disabled. To enable debugging, specify the protocol used by the debugger connection (only "rsp" is supported at present).

The *port* argument specifies the socket port on which to accept a debugger connection. The special value of zero allows OVPsim to choose any free port on the host, otherwise the specified port number is used.

The processor to be debugged is indicated by calling the method debugThisProcessor() on the processor object:

```
  cpu1.debugThisProcessor();
```

4.4.1.5 Setting the quantum

The quantum is set for the whole platform. All processor models use this value. See ImperasLib/source/ovpworld.org/modelSupport/tlmPlatform/1.0/tlm2.0/tlmPlatform.hpp.

```
simple:: simple (sc_module_name name)
    : Platform("simple", …..)
{
    Platform.quantum(100, SC_US); // 100uS  quantum period
}
```

4.4.1.6 Setting the processor Speed

The relative speed of each processor can be controlled individually. Use the icmCpu method setIPS(). See:
ImperasLib/source/ovpworld.org/modelSupport/tlmProcessor/1.0/tlm2.0/tlmProcessor.hpp.

```
simple:: simple (sc_module_name name)
    , cpu1("cpu1", …)
{
    cpu1.setIPS(100000000);  // 100MHz clock freq.
}
```

4.4.1.7 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 ICM_ATTR_SIMEX option 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 icmCpuManager.h : icmNewProcAttrs.

```
, cpu1 ("cpu1", 1, ICM_ATTR_SIMEX )
```

4.4.1.8 Code interception

By default, OVP processor model interfaces install an intercept library to provide semihosting of basic libC functions. Low-level read and write functions used by, for example, printf() are intercepted by CpuManager which sends their output to the simulator output stream. To change this behaviour, you must copy and edit the processor specific TLM2.0 interface (ovpworld.org/processor/or1k/1.0/tlm2.0/processor.igen.hpp in the example). The function which supplies the path to the default semihosting library can be replaced with a null, or by one which returns a library of your choice.

```
class or1k: .... {

private:
  const char *getSHL() {
       return icmGetVlnvString(0, "mydir", "mylib", "myname, "1.0", "model");
  }

public:
  // constructor
  or1k ( ... ) : icmCpu(name, ID, "or1k", 0, getSHL(), ....)
  {
    ...
  }
};
```

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 setDiagnosticLevel(). See bhm.h : bhmGetdiagnosticLevel().

```
 uart1.setDiagnosticLevel(3);
```

4.4.3 Memory Options

By default an OVP processor model will use its TLM2.0 initiator socket for all code and data access, and then try to negotiate DMI where possible. If it is not necessary to simulate bus transactions for a certain memory region, it is possible to map parts of the address space to 'local' memory which will be managed by CpuManager. A processor will never produce TLM transactions within a range which is mapped to local memory.

The platform Examples/Platforms/SystemC_TLM2.0/platform.cpp shows how to use local memory for code and data instead of TLM memory. Search for USE_LOCAL_MEMORY in this file. Note that most compilers assume that instruction and data ports are connected to the same memory. For convenience, the m_localBus variable is provided in the processor model interface. This can be passed by reference to the mapLocalMemory method of each of the processor's bus master ports.

```
…
#ifdef USE_LOCAL_MEMORY
 cpu1.INSTRUCTION.mapLocalMemory(0x0, 0xFFFFF, cpu1.m_localBus);
 cpu1.DATA.mapLocalMemory(0x0, 0xFFFFF, cpu1.m_localBus);
#endif
…
```

If this platform is compiled without USE_LOCAL_MEMORY defined, and the platform run with the environment variable IMPERAS_TLM_CPU_TRACE defined, TLM transactions to the program and data memory (before DMI is established) can be seen. If

the platform is recompiled with USE_LOCAL_MEMORY, no transactions to this region will occur.

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 Modelling 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 systemc.org 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.

##