iGen Model Generator Introduction

This document introduces the use of the Imperas Model Generator ***iGen*** ;
A component of the Imperas Professional tools

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

Author:	Imperas Software Limited
Version:	2.4
Filename:	iGen_Model_Generator_Introduction.doc
Project:	iGen Model Generator Introduction
Last Saved:	Friday, 26 February 2021
Keywords:	iGen Model Generator Introduction

Table of Contents

1 Preface 4
1.1 Notation 4
1.2 Related Documentation 4
1.3 Glossary / Terminology 5
2 Introduction 6
2.1 Prerequisites 6
3 Overview 7
3.1 Obtaining & installing iGen 7
3.2 Using iGen 8
3.2.1 Simplest usage of iGen is with a Makefile 8
3.2.2 Batch mode 9
3.2.3 iGen command line arguments 9
3.2.4 Listing all iGen API functions 9
3.2.5 Getting help on an iGen API function 9
3.2.6 Arguments to iGen scripts 10
3.2.7 Interactive mode 10
3.3 Use of Libraries 11
3.4 VLNV 11
4 Creating Components 13
4.1 Creating a testbench / harness / platform / module 13
4.2 Creating a peripheral model template 13
4.3 Creating a processor model 13
4.4 Creating an extension library template 13
4.4.1 Beginning the extension library 13
4.4.2 Formal parameters 14
4.4.3 Commands 14
4.4.4 Writing the template 15
4.4.5 Adding a standard header 16
5 Creating SystemC TLM2 interfaces 17
5.1 Choice of names 17
5.2 Generating a SystemC TLM2 processor interface 17
5.3 Generating a SystemC TLM2 peripheral interface 17
5.4 Generating SystemC TLM2 code from TCL 18
5.5 Specifics of the SystemC TLM2 Processor Interface 18
5.6 Specifics of the SystemC TLM2 Peripheral Interface 19
5.7 Specifics of the SystemC TLM2 MMC Interface 20
6 Using iGen to Create XML Interface Description 21
6.1 Default Model Interface 21
6.2 Configured Model Interface 21
7 Using iGen to Create TLM2 Platforms 22
7.1 Example platform 22
8 iGen Command Line Arguments 24
9 iGen TCL Commands 27

1 Preface

The Imperas simulators can use models described in C or C++ and the models can be exported to be used in simulators and platforms using C, C++, SystemC or SystemC TLM2.

This document describes the Model Generator, iGen, which executes scripts making calls to the iGen Command API. The scripts use TCL (Tool Control Language) as input and iGen creates C templates for simulation models and plugins, and creates interfaces for SystemC TLM2 simulation and creates virtual platforms, testbenches and modules using the OP C API and SystemC TLM2.

1.1 Notation

`Code` or `code`	Text representing code, a command or output from *iGen*.
keyword	A word with special meaning.

1.2 Related Documentation

There are several documents available as PDF:

Getting Started
- Imperas Installation and Getting Started Guide

Interface and API
- OVP Peripheral Modeling Guide
- OVPsim Using OVP Models in SystemC TLM2.0 Platforms

Also, in your installation there is also the online iGen Function API Command reference documentation. This is correct-by-construction Doxygen-like API documentation available at:

```
        IMPERAS_HOME/doc/api/igen/html/index.html
```

References to specific uses of iGen
- iGen Model Generator Introduction
- iGen Platform and Module Creation User Guide
- iGen Peripheral Generator User Guide

Usage of Modules and Peripherals created using iGen
- Simulation Control of Platforms and Modules User Guide
- Advanced Simulation Control of Platforms and Modules User Guide

1.3 Glossary / Terminology

OP API - OVP Platforms API - C API used for creating and controlling virtual platforms. 2nd generation API, replaces ICM API.

iGen - Imperas productivity tool that has a powerful script-based function API that is used to create C/C++/SystemC models and templates. Described in the iGen Model Generator Introduction, and for platforms, in the iGen Platform and Module Generator User Guide.

OVPsim - Simulator for Open Virtual Platforms that executes platforms and models coded in the OVP APIs

CpuManager - Imperas commercial simulator

Platform / Module – (used interchangeably) - a collection of components connected together into a level of hierarchy in a system to be simulated. This is a program in C/C++ making calls into OP API and normally compiled into a shared object/dynamically linked library and loaded by the simulator at run time.

Testbench / Harness - program in C/C++ making calls into OP API to connect and control OVP components. Normally linked to the simulator to provide a .exe binary that can be executed. Used to instantiate one or more platforms/modules and control their execution. The main difference, from a platform/module, is that a testbench or harness includes a call to the function `main`() and may include the command line parser.

Root Module - used to describe the initial platform/module that instantiates one or more platforms/modules and controls their execution. Used in the testbench / harness.

2 Introduction

Imperas simulation technology enables high performance simulation, debug and analysis of platforms containing multiple processors and peripheral models. The technology is designed to be extensible: you can create your own platforms, new models of processors, and other platform components using interfaces and libraries supplied by Imperas. Platform models developed using this technology can be used both with Imperas simulation products and the freely-available OVPsim platform simulator.

iGen is an Imperas productivity tool that has a powerful script based function API that is used to create C/C++/SystemC models and templates.

2.1 Prerequisites

Since models and platforms for use with Imperas and OVP tools are written in C, an important prerequisite is that you must be proficient in the C language.

iGen uses the TCL scripting language, so you must have some basic understanding of TCL to at least be able to write scripts that call functions.

In this document examples are shown using a shell. This can be either Linux or Windows/MSYS and so you must be familiar with these shells.

3 Overview

iGen can produce a C template for a

- hardware design (module, platform, and harness/testbench)
- processor model
- peripheral model
- extension/interception library
- Memory Model Component (MMC)

iGen can produce a SystemC TLM2 interface for a

- hardware design (platform)
- processor
- peripheral model
- Memory Model Component (MMC)

The C code of a hardware module generated by iGen can often be used as is without modification as in most cases it is purely structural and iGen creates all the necessary C code.

In most cases iGen is not used for generating simulation harnesses and testbenches. These are written directly in C (though iGen can create an initial template).

A peripheral model C template will

- create the basic model files
- create bus ports, net ports and packetnet ports to connect the peripheral to the platform
- construct memory mapped registers

The peripheral template will have no behavior but can provide empty callback functions that can be filled in by the user.

An extension library template will

- construct the basic model files
- install user commands
- parse the arguments of user commands when they are called

The extension library template will have no behavior but will provide empty functions that can be filled in by the user.

The SystemC TLM2.0 platforms create instances of OVP CPUs and peripherals and other components and have been tested with all major SystemC TLM2.0 simulators.

The SystemC TLM2.0 interfaces for OVP CPUs and peripherals have been tested with all major SystemC TLM2.0 simulators.

3.1 Obtaining & installing iGen

iGen is available as part of the Imperas DEV and SDK packages. It is assumed you have downloaded and installed one of these.

3.2 Using iGen

iGen is used mostly in a batch mode using a Makefile. It can also be used directly with a command line or also interactively, but most often it is used with the make system.

3.2.1 Simplest usage of iGen is with a Makefile

In an Imperas installation Makefiles are provided to control the usage of iGen. The best place to start using them is to find an example of what you are working on, and copy how it is built.

For example, to build a hardware module:

```
> cp -r Examples/PlatformConstruction/simpleCpuMemory .
> cd module
> ls
Makefile
module.op.tcl
```

Look at the files - the `Makefile` includes one of the Makefile provided in the installation, in this case the library `Makefile.module`, and the iGen script file: `module.op.tcl` which calls the iGen API functions that define the module.

Create and compile the module by typing:

```
> make
# iGen Create OP MODULE module
# Host Compiling Module obj/Windows64/module.o
# Host Linking Platform object model.dll
```

This uses iGen to generate the C and compiles it to a shared object that will be used in other modules or the testbench / harness.

Note that the Makefiles are keyed to control their operation based on the filenames provided to them. It is recommended that you use the make system and the filenames specified.

Built in file names and the appropriate Makefiles:

File Name	**Makefile to use**
module.op.tcl	Makefile.module
pse.tcl	Makefile.pse
platform.tlm.tcl	Makefile.TLM.igen / Makefile.TLM.platform
platform.op.tcl / platform.tcl	Makefile.platform
platform.icm.tcl	Makefile.platform (deprecated)

Note there is also Makefile.harness which compiles C files, e.g. harness.c to an executable.

3.2.2 Batch mode

iGen can be used directly in batch mode, executing scripts provided on its command line:

```
shell > igen.exe --batch model.tcl --writec model.c

                IMPERAS IGEN (32-bit) version 20150315.0
         Copyright (c) 2005-2015 by Imperas Limited.
                     ALL RIGHTS RESERVED

This program is proprietary and confidential information of
Imperas Limited and may be used and disclosed only as authorized
in a license agreement controlling such use and disclosure.
...
shell >
```

Specifying an input file (--`batch`) and an output file (`--writec`) puts *iGen* into its batch mode. The input script should include commands to create just one platform / module / library / model.

3.2.3 iGen command line arguments

To see all the iGen command line arguments:

```
> igen.exe --help
```

For convenience, these are listed at the end of this document.

3.2.4 Listing all iGen API functions

To see all the iGen API functions:

```
> igen.exe --showcommands
```

Also, in your installation there is also the online iGen Function API Command reference documentation. This is correct-by-construction Doxygen-like API documentation available at:

```
       IMPERAS_HOME/doc/api/igen/html/index.html
```

3.2.5 Getting help on an iGen API function

To get help on an iGen API function, use the `igen --apropos` command line argument with the command or word of interest:

```
> igen.exe --apropos ihwnew
> igen.exe --apropos connect
```

This will show you the built in help for the command.

3.2.6 Arguments to iGen scripts

If required, arguments can be passed to the TCL script using the argument --*batchargv*. This flag can be repeated, each occurrence adding another argument. The TCL script can fetch the argument using the TCL variables *$argc* and *$argv*. e.g.:

```
# myscript.tcl

if { $argc != 3 } {
    puts "Expected three arguments"
    return
} else {
    set one   [lindex $argv 0]
    set two   [lindex $argv 1]
    set three [lindex $argv 2]
    puts "1:$one 2:$two 3:$three"
}
```

```
shell > igen --batch myscript.tcl --batchargv v1 --batchargv v2 --batchargv v3

                IMPERAS IGEN version 99999999
         Copyright (c) 2005-2015 by Imperas Limited.
                     ALL RIGHTS RESERVED

This program is proprietary and confidential information of
Imperas Limited and may be used and disclosed only as authorized
in a license agreement controlling such use and disclosure.

...
1:v1 2:v2 3:v3
...
shell >
```

3.2.7 Interactive mode

iGen can also be used interactively when learning the commands and their arguments, but not as a method of building models in a development environment.

Here `ihelp` is used to list the Imperas tcl commands:

```
shell > iGen.exe
...
iGen  > ihelp
...
...
ihwnew   -  Create a hardware design
...
...
```

All Imperas tcl commands begin with i and will accept the -help argument:

```
iGen  > ihwnew -help
NAME:
    ihwnew  - Begin creating a new hardware design
...
ARGUMENTS:
    -help
...
    -name <string> (mandatory)
```

```
        The VLNV name of the hardware design
...
```

3.3 Use of Libraries

In most situations, iGen does not need to refer to component libraries. However, when writing a TLM platform, information is required from the component specifications, so iGen must locate and load the components before writing the output file(s). Note that to do this, a simulator license will be used for a brief period. When searching for components, iGen uses the same rules as the Imperas simulator; looking in the library in the installation (referred to by the IMPERAS_VLNV environment variable) then using any other libraries specified by `–vlnvroot`. Note that it is an error to have a model with the same VLNV in more than one library.

3.4 VLNV

The SPIRIT consortium (a group of vendors and users of EDA tools that defined standards for exchange of design information), now part of Accellera (www.accellera.org) stipulates that a model should be identified by Vendor, Library, Name and Version tags. They are to be used as follows.

Vendor	The URL of the organization supplying the model (NB. not the component being modeled). Using this should guarantee unique specification of models. Large organization might consider using sub-domains, e.g. performance.imperas.com, services.imperas.com.
Library	Classification of the component e.g. processor, microcontroller, peripheral.
Name	The component identifier, often combining a functional and numeric identifier e.g. UART16550. Note that OVP models often package many variants as one model, using configuration parameters to select the variant at load-time. As a guideline, a model can conveniently include a variant if its external interface is similar (i.e. has a subset of identical bus and net ports).
Version	The revision (not the variant).

Imperas models are specified using VLNV; all models are shipped in a directory structure exactly matching the VLNV. e.g.
<processor vendor="ovpworld.org" library="processor" name="or1k" version="1.0" />

is located in a directory called `<root>/ovpworld.org/processor/or1k/1.0`

NOTE: Since a Windows file-system is case-insensitive, it is important when creating models that case is not used to distinguish VLNVs

The conformance of VLNV and directory names allows Imperas and OVP tools to access the same model hierarchy as the Imperas Simulator.

4 Creating Components

4.1 Creating a testbench / harness / platform / module

This is covered in the document:

- iGen Platform and Module Creation User Guide

4.2 Creating a peripheral model template

This is covered in the documents:

- iGen Peripheral Generator User Guide
- OVP Peripheral Modeling Guide

4.3 Creating a processor model

This is covered in the documents:

- OVP Processor Modeling Guide
- Imperas CPUGenerator Guide

4.4 Creating an extension library template

An extension library specification is a machine (and human) readable specification of the interface to an extension library (to create an extension library, refer to the Imperas binary Intercept Technology User Guide). The specification must define:

- VLNV of the extension.
- Name of the shared object (.dll/.so) which implements it
- Symbol of the model's attributes table
- Configuration parameters specific to this model

The following *iGen* commands are used to create an extension library template:

Command name	**Action**
`imodelnewsemihostlibrary`	Start a new semihost library
`imodeladdformal`	Add a formal parameter
`iadddocumentation`	Add a plain text field
`imodeladdsupportedprocessor`	Document that the library supports this processor
`imodeladdcommand`	Add a command specification.
`imodeladdformalargument`	Add a formal argument to a command.
`imodeladdenumeration`	Add an enumeration to a formal argument

4.4.1 Beginning the extension library

imodelnewsemihostlibrary	**Start a new extension library specification**
`-name`	name of the new extension library
`-vendor`	VLNV vendor of new extension library
`-library`	VLNV library of new extension library
`-version`	VLNV version of new extension library

imodelnewsemihostlibrary	**Start a new extension library specification**
`-header`	Optional path to Copyright header to be included in each C file.

This command begins the specification and describes its location on the library. The optional –header argument allows the user to specify a text file to be included at the start of the output file.

4.4.2 Formal parameters

Formal parameters are used to configure an extension library. They can be set by the simulator or the platform that loads the library. They are referenced in the platform by hierarchical name.

imodeladdformal	**Add a formal parameter to the extension library.**
`-defaultvalue`	The default value of a numeric type.
`-help`	
`-max`	Maximum allowed numeric value
`-min`	Minimum allowed numeric value
`-name`	Name of this formal parameter
`-type`	Data type of this parameter (address, bool, double, endian, enum, flag, float, int32, int64, list, pointer, string, stringlist, uns32, uns64)

4.4.3 Commands

Extension library commands can be invoked by the simulator before simulation, or, in an interactive session, whenever the simulator is stopped. A command has a name and can be followed by optional or mandatory arguments.

imodeladdcommand	**Add a command specification.**
`-name`	name of the command
`-help`	When specified, the command is not executed, but help is supplied.
`-class`	Used by the graphical interface to determine how to present the command. Must be query, status or mode.

Their formal arguments are specified as follows:

imodeladdformalargument	**Add a formal argument to a command.**
`-name`	name of the argument
`-type`	Data type of this parameter (address, bool, double, endian, enum, flag, float, int32, int64, list, pointer, string, stringlist, uns32, uns64)
`-mustbespecified`	This is not an optional argument

If the type of a formal argument is specified as enumeration, enumeration names are added as follows:

imodeladdenumeration	**Add an enumeration to a formal**
`-name`	name of the enumeration
`-formal`	name of the formal with this enumerated type

iadddocumentation	**Adds a documentation field to any object**
`-handle`	hierarchical name of the object to be documented
`-name`	name of the documentation category (e.g. Description or Limitations)
`-text`	name of the formal with this enumerated type

iadddocumentation adds a text field to a model. Imperas uses the names *Description*, *Limitations* and *Licensing* although any names are accepted. These text fields can be introspected from the binaries of the models so that tools can probe a model and get information about it. Imperas has tools that build documentation from introspecting models. Documentation fields can also be added to ports and formals. Without -*handle*, the field is added to the root of the model. If specified, the handle should match the string returned by one of the *imodeladdxxx* commands.

```
# createSHL.tcl

imodelnewsemihostlibrary \
    -name           or1kNewlib \
    -vendor         ovpworld.com \
    -library        semihosting \
    -version        1.0 \
    -imagefile      model \
    -attributetable modelAttrs

iadddocumentation -name Description -text "Test model"
iadddocumentation -name Limitations -text "Do not use."
```

4.4.4 Writing the template

TCL commands are added to a TCL file which when supplied to iGen, which supplied to *iGen* puts it into batch mode, writing the requested template output. The template contains C code which can be compiled to create the beginnings of an extension library. The user is then expected to write more code to provide the functionality of the library.

Please refer to Imperas_Binary_Intercept_Technonology_User_Guide.pdf for a description of how to write an extension library.

Argument	**File**
`--batch <TCL input file>`	File of TCL commands to construct the extension library.
`--writec <output file>`	File containing the body of the model, stem of other file names
`--userheader <input file>`	Prepend this text file to each C file (it must be legal C). Can be overridden from the TCL.
`-newargparser`	Use the vmi command parser API (the old method is deprecated)

In this example the name 'model' is used, producing the files listed below:

```
shell> iGen.exe                       \
    --batch         extension.tcl   \
    --writec        model.c         \
    --userheader    companyheader.h \
    --newargparser
```

Output files are specified by –`writec model.c`

File	Contains
model.c	Stub functions to be filled by the user
model.igen.c	The command parser and other code that does not need to be edited.
model.igen.h	Prototypes of generated functions.
model.macros.igen.h	Offsets of each command argument in the parsed array.

If no file extension is provided, igen adds the extension `.igen.stubs`

```
shell> igen.exe                          \
    --batch         extension.tcl   \
    --writec        model           \
    --userheader    companyheader.h \
    --newargparser
```

File	Contains
model.igen.stubs	Stub functions to be filled by the developer
model.igen.c	The command parser.
model.igen.h	Prototypes of generated functions.
model.macros.igen.h	Offsets of each command argument in the parsed array.

Note that since the stubs file can be modified by the user, igen will not overwrite an existing file unless the *–overwrite* flag is supplied.

4.4.5 Adding a standard header

Some organizations require each source file to include a standard header. Header text can be prepended to a generated file using the *--userheader* command line option.

5 Creating SystemC TLM2 interfaces

OVP models can be used in a SystemC TLM2 simulation (see OVPsim_Using_OVP_Models_in_SystemC_TLM2.0_Platforms). To connect an OVP model to SystemC TLM2 an interface is required. iGen can help to generate this interface. In fact most SystemC TLM2 model interfaces shipped with an Imperas release were generated by *iGen*.

It is recommended that the model TLM interface files are generated from the model binary file, rather than the TCL description. This method allows the setting of additional parameters during the generation process.

5.1 Choice of names

Component instance names are converted directly to C++ where possible and modified where not. If you want your names to remain unchanged you should therefore avoid:

- C and C++ keywords.
- Names with illegal C identifier characters, e.g. '.' (period).

5.2 Generating a SystemC TLM2 processor interface

iGen can generate the processor model interface directly from the processor model (which must already exist as an executable shared object or DLL)

To produce the processor interface, specify the processor model or sufficient of its VLNV to give a unique result, then specify the output file for the interface TLM2 code:

```
sh> igen.exe -modelname or1k --writetlm or1k.cpp
```

The interface must be produced from the model, rather than from TCL, because a processor model interface can change according to parameters specified during its construction, and the interface must contain each feature of the current configuration. Parameters can be passed to the model to select the correct variant or configuration:

```
sh> igen.exe -modelname microblaze --writetlm blaze.cpp \
      –setparameter variant=V8_00
```

5.3 Generating a SystemC TLM2 peripheral interface

iGen can generate the peripheral model interface directly from the peripheral model (which must already exist as an executable pse.pse)

To produce the peripheral interface, specify the peripheral model or sufficient of its VLNV to give a unique result, then specify the output file for the interface TLM2 code:

```
sh> igen.exe -modelname PLIC --writetlm plic.igen.hpp
```

The interface must be produced from the model, rather than from TCL, because a peripheral model interface can change according to parameters specified during its construction, and the interface must contain each feature of the current configuration. Parameters can be passed to the model to select the correct configuration:

```
sh> igen.exe -modelname PLIC --writetlm plic_127_2.igen.hpp \
       –setparameter num_targets=2 -setparameter num_sources=127
```

5.4 Generating SystemC TLM2 code from TCL

A SystemC TLM2 interface can be produced from the TCL that describes the model or platform / module:

```
shell> igen.exe --batch myMmodel.tcl --writetlm myOutputfile.cpp
```

The output file format depends on the model described in the TCL file. TCL descriptions begin with:

`imodelnewperipheral`	create code for a peripheral interface
`imodelnewmmc`	create code for an mmc interface
`ihwnew`	create module construction code

For successful generation, the TCL description must be correct. A platform must use correctly connected components available in the library (or on the current search path). Errors in the TCL will prevent the file being written. The *iGen* return code is checked by *make*, if you use it.

5.5 Specifics of the SystemC TLM2 Processor Interface

To discuss the specifics of the SystemC TLM2 interface for OVP processor models, we will explore the OR1K model.

Referring to the file
`ImperasLib/source/ovpworld.org/processor/or1k/1.0/tlm2/processor.igen.hpp`

The interface is a specialization of the generic functionality in
`ImperasLib/source/ovpworld.org/modelSupport/tlmProcessor/1.0/tlm2/processor.igen.hpp`

It is implemented in a class with the same name as the processor description, `or1k` in this example. This class uses functionality from the `tlmProcessor` class in the CpuManager API.
It creates an instance of `tlmBusMasterPort` for each processor bus interface and uses an instance of `tlmNetInputPort` for each interrupt input. As in a platform constructed using the OP C interface, the processor instance requires a name (usually the instance name) and a unique ID, typically a small integer.

The bus master ports are mapped to TLM2.0 initiator sockets.

The constructor can pass parameters to the model.

The constructor uses `opVlnvString` to calculate the file path to the processor model, then constructs the processor model, the bus interfaces and the interrupt interfaces. An example of a platform using this model is described later.

5.5.1.1 DMI

The generic `tlmProcessor` class will attempt to use DMI for data and code access, so can achieve simulation speeds close to those of OVPsim and CpuManager.

5.5.1.2 Code Caching

Note that Imperas/OVP simulators use dynamic code translation to achieve their high simulation speed. In a pure OVP simulator environment the simulator is notified if code memory is overwritten, so that code can be re-translated. In a SystemC environment code is usually stored in SystemC memory which can be written by non-OVP models. In this case, OVPsim will not be notified of code changes so will not re-translate. Refer to documentation for the `flush()` method on the OP processor object.

5.6 Specifics of the SystemC TLM2 Peripheral Interface

To discuss the specifics of the SystemC TLM2 interface for OVP peripheral models, we will explore the National 16450 UART model.

Referring to the file

```
ImperasLib/source/national.ovpworld.org/peripheral/16450/1.0/tlm2/pse.igen.hpp
```

The interface is a specialization of the generic functionality in

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

It is implemented in a class with the same name as the peripheral description, *Uart16450* in this example. This class uses functionality from the tlmPeripheral class in the CpuManager API.
It creates an instance of tlmBusSlavePort for each bus slave interface, tlmBusMasterPort for each bus master interface (not used here) and uses an instance of tlmNetOutputPort for each single-bit output. Like in CpuManager, the peripheral instance requires a name (usually the instance name).

The bus master ports are mapped to TLM2.0 initiator sockets and the bus slave ports are mapped to TLM2.0 acceptor sockets.

Note that the constructor can also pass user defined parameters to the model.

The constructor uses opVlnvString to calculate the file path to the model, then constructs the model, the bus interfaces and the pin interfaces. An example of a platform using this model is described later.

5.7 Specifics of the SystemC TLM2 MMC Interface

Consider the file

```
ImperasLib/source/ovpworld.org/mmc/wb_1way_32byteline_2048tags/1.0/tlm2/mmc.igen.hpp
```

The interface is a specialization of the generic functionality in

```
ImpPublic/source/host/tlm/tlmMMC.cpp
```

And included by

```
ImpPublic/include/host/tlm/tlmMMC.hpp
```

It is implemented in a class with the same name as the MMC description, `wb_1way_32byteline_2048tags` in this example. This class uses an MMC model from the VLNV.
The class creates an instance of an MMC and maps the bus master port to a TLM2 initiator socket and the bus slave port to TLM2 acceptor sockets. Thus a transaction from a connected TLM initiator results in a callback in the MMC TLM interface. This creates a callback into the MMC from a bus created for this purpose in the TLM interface. Depending on its behavior, the MMC could consequently initiate a TLM transaction on a bus connected to the model.

Parameters in the platform TCL applied to an MMC instance result in *iGen* creating a function to pass a parameter list to the OVP MMC model.

6 Using iGen to Create XML Interface Description

The interface and configuration parameters of OVP models can be generated as an XML format file which is convenient for parsing to allow the extraction of information by a third-party tool.

The Imperas VLNV library includes an XML description of the default model interface for each component.

The generation of the default XML interface file or a configured model XML interface file is shown in the following sections.

6.1 Default Model Interface

The default interface of a model can be generated by providing the model unique VLNV on the iGen command line as shown below.

Processor model
```
shell> igen.exe –modelvendor riscv.ovpworld.org -modelname riscv \
                -writexml riscv_default.xml
```

Peripheral model
```
shell> igen.exe –modelvendor riscv.ovpworld.org -modelname CLINT \
                -writexml clint_default.xml
```

6.2 Configured Model Interface

When a model is instanced into a virtual platform it is usually configured. For example, for a processor model the variant will be specified and also other configuration options.

The configured interface of a model can be generated by providing the model unique VLNV and further configuration parameter overrides and settings on the iGen command line as shown below.

Processor model; setting the variant and overriding other parameters
```
shell> igen.exe –modelvendor riscv.ovpworld.org -modelname riscv \
                -variant RV32IMAC \
                -override add_Extensions=V \
                -writexml riscv_rv32IMACV.xml
```

Peripheral model; overriding parameters
```
shell> igen.exe –modelvendor riscv.ovpworld.org -modelname CLINT \
                -override num_harts=8 \
                -writexml clint_num_harts8.xml
```

NOTE
Use the iGen command line argument `-showvariants` with a processor model to show the included variants and use `-showoverrides` with any model to show the configuration parameters that can be modified.

7 Using iGen to Create TLM2 Platforms

A SystemC platform generated by *iGen* uses standard files that interface between SystemC TLM2 functions and the Imperas OP API. The standard interface files are in:

```
   ImpPublic/source/host/tlm
    ImpPublic/include/host/tlm
```

Code that initializes the virtual platform is in

```
    tlmModule.cpp
```

Components interfaces are in

```
    tlmMMC.cpp
    tlmPeripheral.cpp
    tlmProcessor.cpp
```

Port interfaces are in

```
    tlmBusDynamicSlavePort.cpp
    tlmBusSlavePort.cpp
    tlmBusMasterPort.cpp
    tlmDMIsLave.cpp
    tlmNetInputPort.cpp
```

The Imperas bus model (a simple address decoder) is in

```
    tlmDecoder.cpp
```

Two types of memory are in

```
    tlmMemory.cpp
    tlmDenseMemory.cpp
```

7.1 *Example platform*

Consider the example platform (generated by *iGen*) in

```
Examples/PlatformConstruction/SystemC_TLM2.0/platform_cpp
```

The input file, read by iGen is

```
platform.tlm.tcl
```

Output files, created by iGen are in

```
platform.*.igen.h
```

The platform is a class, inheriting `sc_core::sc_module`. It constructs SystemC instances of each component in the platform and also an instance of the `tlmPlatform` class, used to initialize the simulator. Buses are modeled using a bus decoder `tlmDecoder`. Memories use `tlmRam`. Note that memories are sparse by default so allocate native storage only as required.

Each component from the Imperas VLNV is implemented as a class. Its definition is in the VLNV source tree (not binary) next to the model. The SystemC platform creates an instance of a model class to create an instance of the OVP model. A function is created for each instance to set OVP parameters when required.

The `tlmDecoder` decoder template class is parameterized according to the number of bus master and slave connections.

The `tlmPlatform` constructor puts the simulator is a mode which is usually suitable for SystemC simulation (the user might wish to change it) then calls each model constructor. Standard SystemC code binds initiator to acceptor sockets. *iGen* uses the SystemC TLM2 tlm_`analysis`_port to connect single-bit ports that model nets.

8 iGen Command Line Arguments

Group	Flag		Argument	Description
Diagnostics				
	`--apropos`		command	Show igen commands similar to the given argument
	`--help`	-h		Print list of flags
	`--showcommands`			Show all igen commands
	`--showdomains`		[root]	List the (initial) state of each memory domain
	`--showenvvars`			List all environment variables read by Imperas products
	`--showmodeloverrides`		[root]	List overrides requested by models in the platform
	`--showoverrides`		[root]	List all possible platform overrides
	`--showsystemoverrides`		[root]	List overrides in the platform provided by the simulator
Input				
	`--batch`	-b	filename	Execute this tcl file
	`--batchargv`		argument	Argument to --batch file
	`--checkmodels`			Load and check models when writing a platform
	`--exec`	-e	string	Execute this tcl string
	`--modellibrary`		string	Processor VLNV library
	`--modelname`		string	Processor VLNV name
	`--modelvendor`		string	Processor VLNV vendor
	`--modelversion`		string	Processor VLNV version
Library				
	`--showlibraryextensions`			List semihost and intercept libraries in the paths set by $IMPERAS_VLNV
	`--showlibrary`			List models and platforms in the paths set by $IMPERAS_VLNV
	`--showlibrarymmcs`			List MMCs in the paths set by $IMPERAS_VLNV
	`--showlibrarymodules`			List modules in the paths set

Group	**Flag**		**Argument**	**Description**
				by $IMPERAS_VLNV
	`--showlibraryperipherals`			List peripherals in the paths set by $IMPERAS_VLNV
	`--showlibraryplatforms`			List platform executables in the paths set by $IMPERAS_VLNV
	`--showlibraryprocessors`			List processors in the paths set by $IMPERAS_VLNV
	`--vlnvroot`		path	Add to the search path for models
Log				
	`--excludem`	-x	string	Exclude message category
	`--logfile`		Filename	Output log file
	`--nobanner`			Suppress product banner
	`--nowarnings`	-w		Suppress warnings
	`--output`	-o	Filename	Output log file
	`--quiet`	-q		Suppress information messages
	`--verbose`	-v		Produce verbose output
	`--version`			Print version information
	`--werror`	-W		Treat warnings as errors
Output				
	`--html`			Command help written in HTML
	`--icm`			Write C using deprecated ICM API, rather than OP
	`--newargparser`			Intercept library using the new argument parser
	`--op`			Write C using OP API (this is the default)
	`--overwrite`			Overwrite any existing file
	`--single`			Write a single file
	`--userheader`		filename	Put this file at the top of generated C files
	`--writec`		filename	Write C model or platform
	`--writetlm`		filename	Write TLM interface
	`--writexml`		filename	Write XML from tcl
Parameters				
	`--setparameter`		string	Set a processor parameter

Group	**Flag**		**Argumen t**	**Description**
				(param=value)
	`--Showvariants`			Show list of variants

9 iGen TCL Commands

iGen Command	Description
`iadddocumentation`	Add a documentation entry to a model, design or item.
`ihwaddbridge`	Add a bridge instance to a hardware design.
`ihwaddbus`	Add a bus instance to a hardware design.
`ihwaddbusport`	Add an external bus port to a hardware design.
`ihwaddclp`	Add a command line parser to a platform executable.
`ihwaddclparg`	Add an argument to the command line parser.
`ihwaddenumeration`	Add an enumeration value to a formal parameter
`ihwaddextensionlibrary`	Add an extension library to a peripheral or processor instance
`ihwaddfifo`	Add a FIFO instance to a hardware design.
`ihwaddfifoport`	Add an external FIFO port to a hardware design.
`ihwaddformalmacro`	Include a file (defined in a header file).
`ihwaddformalparameter`	Create a formal for a parameter to the current module.
`ihwaddimagefile`	Add a new image file to be loaded onto a bus or processor instance.
`ihwaddmemory`	Add a memory instance to a hardware design.
`ihwaddmmc`	Add a Memory Model Component instance to a hardware design.
`ihwaddmodule`	Add an instance of a module to this hardware design.
`ihwaddnet`	Add a net to a hardware design.
`ihwaddnetport`	Add an external net port to a hardware design.
`ihwaddpacketnet`	Add a packetnet instance to a hardware design.
`ihwaddpacketnetport`	Add an external packetnet port to a hardware design.
`ihwaddperipheral`	Add a peripheral instance to a hardware design.
`ihwaddprocessor`	Add a processor instance to a hardware design.
`ihwaddprotection`	Add a protected region to a bus.
`ihwaddsymbolfile`	Load the symbols from this image file onto a bus or

	processor instance.
`ihwconnect`	Create connections between component ports.
`ihwnew`	Create a new hardware design.
`ihwsetparameter`	Set a parameter on a model instance
`imodeladdaddressblock`	Add an address block to a slave port.
`imodeladdbusmasterport`	Add a new bus master port to a model.
`imodeladdbusslaveport`	Add a new bus slave port to a model.
`imodeladdcommand`	Add a command to an extension library
`imodeladdelf`	Add an extra elf code supported by this processor
`imodeladdenumeration`	Add an enumeration value to a formal attribute or argument
`imodeladdextensionlibrary`	Add an extension library to a peripheral or processor model
`imodeladdfield`	Add a bit field definition to a memory mapped register.
`imodeladdfifoport`	Add a new FIFO port to a model.
`imodeladdformal`	Add a new formal parameter to a model.
`imodeladdformalargument`	Add a formal argument to a command.
`imodeladdformalmacro`	Include a macro (defined in the PPM header file).
`imodeladdlocalmemory`	Add a local memory to an address block.
`imodeladdmmregister`	Add a memory mapped register to an address block.
`imodeladdnetport`	Add a new net port to a model.
`imodeladdpackage`	Put the model in the specified package.
`imodeladdpacketnetport`	Create a packetnet port.
`imodeladdregister`	Add a register to a model.
`imodeladdreset`	Add a reset value to a memory mapped register.
`imodeladdsupportedprocessor`	Add a supported processor to the list on an intercept library (for documentation only).
`imodelnewmemory`	Create a new memory model.
`imodelnewmmc`	Create a new memory mapped component model.
`imodelnewperipheral`	Create a new peripheral model.

`imodelnewprocessor`	Create a new processor model.
`imodelnewsemihostlibrary`	Create a new semihosting library.
`imodulenew`	Create a new hardware module.
`isetattribute`	deprecated function - use `ihwsetparameter`
`quit`	Finish the session

##