iGen Peripheral Generator User Guide

This document describes the use of the
Imperas Model Generator ***iGen*** to generate a peripheral model.

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

Author:	Imperas Software Limited
Version:	2.2
Filename:	iGen_Peripheral_Generator_User_Guide.doc
Last Saved:	Friday, 20 October 2017
Keywords:	iGen Peripheral Model Generator User Guide

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

1 Preface

The Imperas simulators can use models described in C or C++. The models can be exported to be used in simulators and platforms using C, C++, SystemC or SystemC TLM2.0.

This document describes the use of Imperas Model Generator, iGen, to create a C template for a peripheral simulation model.

1.1 Notation

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

1.2 Related Documents

Getting Started

- Imperas Installation and Getting Started Guide

Interface and API

- OVP Peripheral Modeling Guide
- Writing Platforms and Modules in C User Guide
- Simulation Control of Platforms and Modules User Guide

References to general iGen document

- iGen Model Generator Introduction

2 Introduction

This document describes the use of *iGen* for peripheral model template generation. General information on *iGen* can be found in other referenced documents.

iGen is used as a batch program, reading a TCL input file and writing C output files. All *iGen* command line arguments are in the unix style, requiring one hyphen, but accepting two (with no distinction). Some command line arguments must be followed by a value.

As a learning aid, *iGen* can be used interactively; simply invoke *iGen* without arguments and wait for the iGen prompt. The command *ihelp* lists the Imperas TCL extension commands, each of which accepts the –help argument which prints its arguments and usage.
Alternatively, *iGen* will print the help information of a specific command by supplying it on the command line after the–*apropos* argument.

A peripheral model template will

- Construct a model instance
- Construct bus and net ports for connection to the platform
- Construct memory mapped registers and memory regions.
- Construct formal parameters which can be set when the peripheral is instanced in a platform or module and overridden by the simulator to control features of the peripheral model.

The peripheral template will provide empty functions, stubs, that can be filled in by the user to add behavior to the model.

The peripheral template can be compiled and used in simulations to provide the peripheral device programmers view i.e. the register structure and a default behavior.

iGen can also generate a SystemC TLM2.0 interface for the model. Examples of SystemC TLM2.0 interfaces for OVP peripherals have been tested with all major SystemC TLM2.0 simulators.

3 Creating a peripheral model definition

A peripheral model definition must define:

- VLNV of the model
- Bus master and slave ports which interface to simulated memory spaces
- Net and packetnet inputs and outputs
- Configuration parameters specific to this model

These *iGen* commands are used to create a peripheral model:

Command name	Action
imodelnewperipheral	Start a new peripheral model
imodeladdbusmasterport	Add a bus master port
imodeladdbusslaveport	Add a bus slave port
imodeladdaddressblock	Add a region for memory mapped registers to a bus slave port
imodeladdlocalmemory	Add a region of local memory to a bus slave port
imodeladdmemmregister	Add a memory mapped register to an address block
imodeladdfield	Add a bit field to a register
imodeladdreset	Add a reset function to a register
imodeladdnetport	Add an in input or output (wire) port
imodeladdpacketnetport	Add a packetnet port
imodeladdformal	Add a formal parameter
iadddocumentation	Add a description to a model feature

3.1 A basic model

`imodelnewperipheral` begins the construction of the new model. Other commands can be used until the model is completed. The model C files are written when the script ends.

imodelnewperipheral	Start a new peripheral model
-name	VLNV name of the new peripheral
-vendor	VLNV vendor of new peripheral
-library	VLNV library of new peripheral
-version	VLNV version of new peripheral
-extensionfile	Specify a host interface shared object for this model
-constructor	Function to be called at construction time
-destructor	Function to be called at destruction time

imodeladdbusslaveport	Add a bus slave port
-addresswidth	number of address bits that can be decoded
-name	name of new port
-size	range of addresses this port can generate (in bytes)
-mustbeconnected	if specified, the simulator will fail to run if the port is unconnected

-remappable	this port is not at a fixed address

If a slave port has no size specified (or has a size of 0) then it is left to the user defined code to implement the port, this is a dynamic port that is defined during the peripheral model execution.

imodeladdbusmasterport	**Add a bus master port to the model**
-name	name of new port
-addresswidth	number of address bits driven by the bus master
-mustbeconnected	if specified, the simulator will fail to run if the port is left unconnected
-errorinterrupt	specify the interrupt-output to be asserted if a bus error occurs.

imodeladdnetport	**Add a net port to the model**
-name	name of the new net port
-mustbeconnected	if specified, the simulator will fail to run if the port is unconnected
-type	must be **input** , **output** or **inout**
-updatefunction	Function to be called when the net is written.
-updatefunctionargument	specify a value to pass to the update function

A packetnet is used to model packet based communication such as Ethernet, CAN bus or GSM. A packetnet is created in a platform, then connected to packetnet ports on model instances. A packetnet can have many connections, each able to send or receive packets. *iGen* creates a handle which is used when writing (sending) a packet to the packetnet. It also creates the stub of a function that is called each time the packetnet is written.

imodeladdpacketnetport	**Add a packetnet port to the model**
-name	name of the new packetnet port
-mustbeconnected	if specified, the simulator will fail to run if the port is unconnected
-maxbytes	Maximum size of a packet, checked at run-time
-updatefunction	Function to be called when the packetnet is written.
-updatefunctionargument	specify a value to pass to the update function

If the *--updatefunction* argument is specified, a stub function will be created in the 'user' file, and that function will be called when the net is written by another model.

imodeladdformal	**Add a formal to the model**
-name	name of the formal attribute
-type	a legal formal type (see below)
-min	optional minimum value for an integer parameter
-max	optional maximum value for an integer parameter
-defaultvalue	optional default value for the parameter

A *formal* defines a parameter (sometimes and previously referred to as an attribute) which will be accepted by the model. It is expected that bhmIntegerAttribute(), bhmStringAttribute() etc. will be used in the model to fetch the value. If the formal has a numeric type, default and legal minimum and maximum values can be specified.

Formal type	Description	min/max?
Address	An address	y
bool, Boolean or flag	Boolean value	n
double or float	C double-precision floating point	y
enum or enumeration	Select one from a set names	n
int32 or integer	32 bit signed integer	y
string	Null-terminated C string (null if not specified)	n
uns32	32 bit unsigned integer	y
uns64	64 bit unsigned integer	y

If a formal type is *enumeration* then legal values are added with imodeladdenumeration:

imodeladdenumeration	Add an enumeration to a formal
-name	Name of the enumeration
-formal	Name of the formal. Defaults to the last created formal.
-value	Optional integer value. Defaults to the previous value plus one.

iadddocumentation	Add a documentation entry to a model or object
-name	name of the entry
-text	content
-handle	object to be documented. Defaults to the most recently created object.

Imperas uses the names `Description`, `Limitations` and `Licensing` although any names are accepted. Documentation fields can also be added to

- The model
- Net ports
- Formal arguments
- Bus ports
- Address blocks
- Memory mapped registers
- Register bit fields.

Without `-handle`, the field is added to the last created object. If specified, the handle should match the string returned by one of the `imodeladd` commands.

Documentation entries also have handles. Adding documentation to an existing document handle results in indented text.

3.1.1 Example

The following example *iGen* script creates a similar model template to that in the model *ovpworld.com/peripheral/SimpleDma/1.0* which is included in an OVP release.

Bus master ports *MREAD* and *MWRITE* let this model initiate bus cycles. The separate ports let it transfer data between two busses if required. Its address logic can generate 32-bit addresses.

A bus slave port *DMACSP* is used to read and write its programming registers. The port will occupy 0x140 bytes of space on the connected bus.

The model has one reset input port and two interrupt output ports.

The model would not function if any bus port was unconnected, however the interrupt and reset connections are optional. Thus only the bus ports specify the `-mustbeconnected` flag.

```
# createPeripheral.tcl

imodelnewperipheral \
    -name      SimpleDma \
    -vendor    ovpworld.com \
    -library   peripheral \
    -version   1.0

iadddocumentation -name Description -text "Test peripheral"
iadddocumentation -name Limitations -text "Under development."

imodeladdbusmasterport -name MREAD  -addresswidth 32 -mustbeconnected
imodeladdbusmasterport -name MWRITE -addresswidth 32 -mustbeconnected
imodeladdbusslaveport  -name DMACSP -size 0x140      -mustbeconnected

imodeladdnetport -name RESET -type input
imodeladdnetport -name INTR  -type output
imodeladdnetport -name INTRC -type output
```

3.2 Adding memory mapped registers

The example model has an external interface sufficient to connect it to its platform, but has no internal structure. Defining memory mapped registers at this stage gives several benefits;

- The registers are documented.
- The peripheral template generator can generate C code to construct the registers.
- The constructed registers will appear in the debugger's user-view.

A group of similar width registers are contained in an addressblock. A busslaveport can contain one or more addressblocks which must not overlap, though they can have gaps between them.

A register can be specified as read-only, read-write or write-only. A register which is writable can have a *writemask* which specifies those bits that are written. Bits that are unset in the *writemask* retain their previous values.

A register can be divided into bit fields of different widths. The fields need not occupy the whole register.

A register can be associated with a reset input. When the reset is asserted, those bits specified in the reset *mask* value take the reset *value*.

The following tables list the memory-mapped register commands:

imodeladdaddressblock	**Add a region to contain memory mapped registers**
-port	name of port which addresses this block. Defaults to the last created port.
-name	name of the new address block
-width	width (in bits) of the registers in this block
-size	number of bytes in this block
-offset	offset in bytes from the base of the slave port. Defaults to zero.

imodeladdmmregister	**Add a memory mapped register to an address block**
-access	legal access mode(s). Must be r, w or rw
-addressblock	block containing this register. Defaults to the last created block.
-width	width in bits of this register (default to the width of the address block)
-offset	offset in byte from base of this block
-name	name of new register
-readfunction	function to implement a read access
-viewfunction	function to implement a debugger view access (with no side effects)
-writefunction	function to implement a write access
-writemask	a write access will update only these bits in the register
-userdata	pass this value to the userdata parameter of the read & write functions
-nonvolatile	the write function is called only if the written value changes

imodeladdfield	**Add a bit-field to a register**
-access	legal access mode(s). Must be r, w or rw
-name	name of the new bit field
-bitoffset	offset in bits from the LSB (zero is the LSB). Defaults to zero.
-width	width in bits of the new field
-mmregister	register containing the new field. Defaults to last created register

imodeladdreset	**Add a reset function to a register**
-name	name of reset input which activates this action
-mmregister	name of the register to receive the action. Defaults to last created register.
-value	new value to apply (bitwise 'and' with the mask). Defaults to zero.
-mask	mask that specifies which bits to affect. Defaults to all bits.

Register widths are specified in bits; the implementation rounds each width upwards to the nearest byte alignment:

specified width	implemented
1 - 8	8
9 - 16	16
17 - 32	32
>32	(not implemented)

Sizes and offsets are specified in bytes.

3.2.1 Register fields

At present, the field access control is for documentation and as a place keeper for possible future features.

Memory mapped registers, their fields and their access permissions are described in TCL and represented in igen when it is creating a model. The field information is used to create C bit-fields in the structure that describes the storage which can be used within the peripheral model. However, C does not support 'const' in its bit-field syntax so there is no way to declare a bit field with limited permissions.

Although igen can generate a register's structure definition and read and write callbacks, the majority of a peripheral is arbitrary C code, supplied by the user, where there is no way for the tools to protect against illegal access.

At present the simulator run-time has no representation of register bit fields (there's no way to query the peripheral), so there is no visibility in the debugger or VAP, but their existence allows Imperas to add this feature in the future.

3.2.2 Example

This iGen tcl adds registers to the slave port of the previous example.

```
imodeladdaddressblock  -port   DMACSP -name AB1  -width 8  -size 0x40 -offset 0

imodeladdmmregister    -name intStatus -access r -offset 0x0

imodeladdmmregister    -name intTCstatus -access r -offset 0x4

imodeladdmmregister    -name TCclear -access w -offset 0x8 \
                       -writefunction TCclearWr

imodeladdmmregister    -name config -access rw -offset 0x30 \
                       -readfunction  regRd8 \
                       -writefunction configWr
                       -writemask     0x7F
```

The first block of registers is each 8 bits wide, addressed on 32-bit word boundaries.

The `-access` argument controls whether the register has read or write access. Note that read and write function names need not be supplied. The generator creates a structure member for each register (with bit-field unions if specified) and uses a default read and write function to access the structure member. The structure is declared in the header file so can be accessed by the user's code. If an action is required when a register is read or written, `-readfunction` or `-writefunction` is specified, causing code to be triggered by the action. The writemask option generates code to update only selected bits of the register. Those bits with a one in the mask are updated; the others keep their previous value.

```
imodeladdaddressblock  -port DMACSP -name AB2 -width 32 -size 0x34 -offset 0x100

imodeladdmmregister    -name srcAddr0 -access rw -offset 0x0
```

```
imodeladdmmregister    -name dstAddr0 -access rw -offset 0x4

imodeladdmmregister    -name srcAddr1 -access rw -offset 0x20

imodeladdmmregister    -name dstAddr1 -access rw -offset 0x24
```

The second block of registers is 32 bits wide, on 32-bit boundaries. The registers use the default read and write functions (because reading and writing does not trigger any action in the model).

In most situations the `-nonvolatile` option can be omitted. If a register is written many times by the application (in a tight loop, for instance) the execution of the callback could dominate the simulation time. The `-nonvolatile` option allows the simulator to improve this situation be calling the write function only when the written value changes.

3.3 Adding memory to a peripheral

`imodeladdmmregister` is used to model individual word-orientated registers. To model a large bank of similar registers, or a region of memory with similar behavior behind each word, use `imodeladdlocalmemory.` These are the arguments:

imodeladdlocalmemory	**Add a local memory**
-access	type of access. One of **r**, **w** or **rw** (defaults to rw)
-addressblock	block to contain this memory. Defaults to last created block.
-offset	offset in bytes relative to base of the address block (default=0)
-name	name of the new block
-size	size in bytes. Must fit the address block.
-changefunction	function called when region is written with a new value
-readfunction	function called when region is read
-writefunction	function called when region is written
-userdata	value passed to read, write and change functions

If the readfunction, writefunction or changefunction arguments are omitted, the model template will implement a RAM. If the readfunction, writefunction or changefunction arguments are specified, the model template will contain stub functions which will be called on read, write and change events. These must be completed by the model writer. Note that the write function will be called on any write to the memory; the change function will be called only if a new value is written. Using the change function can significantly improve simulation performance where the application frequently re-writes the same value and no action is required by the peripheral model.

4 Generating the peripheral model template

iGen Argument	File
-batch <tcl input file>	File of tcl commands to construct the peripheral.
-writec <output name>	The name of the output stubs file, and the stem used to generate the other output file names.
-userheader <input file>	Prepend this text file to each generated file (it must be legal C).
-overwrite	Overwrite the output stubs file.

All of the output file names are derived from the arguments passed to –writec. If the argument includes a directory (delimited by '/'), then all files are written to this directory. If the argument has an extension, then the stubs file takes this name. *iGen* removes the extension and adds different extensions for the other output files. If the argument has no extension, then one is added. Note that the stubs file is modified by the user (to add behavior to the model), so this file will not be overwritten if it exists. Use the –overwrite argument to change this behavior. Other files are always overwritten.

Example without extension:

```
shell> igen.exe    --batch dmac.tcl    --writec dmac
```

File	Contents
dmac.igen.c	main(), constructors etc.
dmac.igen.h	Function prototypes, storage for registers etc.
dmac.igen.stubs	Stub functions to be expanded by the user
dmac.macros.igen.h	Register offsets for application programs
dmac.attrs.igen.c	modelAttrs structure and associated data.

Example with extension:

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

File	Contents
dmac.igen.c	main(), constructors etc.
dmac.igen.h	Function prototypes, storage for registers etc.
dmac.c	Stub functions to be expanded by the user
dmac.macros.igen.h	Register offsets for application programs
dmac.attrs.igen.c	modelAttrs structure and associated data.

4.1 The user stubs file

All output files start with the name stub specified by `--writec`. The user stubs file, ***dmac.igen.stubs***, contains stub functions. *iGen* creates this file, then the user can copy to a C file and add functionality to each stub. Stub functions are created for each register

and local memory that had the `-readfunction` or `-writefunction` argument specified when it was created, for each input net port that had the `-updatefunction` argument specified, and model constructor and destructor functions if the `-constructor` or `-destructor` arguments were provided to the `imodelnewperipheral` command. If it already exists, this file will not be overwritten. Use `–overwrite` if the existing file is not required.

4.2 The main file

The output file ***dmac.igen.c*** contains the model constructor which connects to bus and net ports, creates registers and initializes register values. Documentation fields created using the `iadddocumentation` command are added to the beginning of the file. This file does not normally require editing, so will be overwritten each time *iGen* runs.

4.3 The include file

The output file ***dmac.igen.h*** contains code required by the other C files. This file does not normally require editing, so will be overwritten each time *iGen* runs.

4.4 The attributes file

The output file ***dmac.attrs.igen.c*** contains code to product objects that can be interrogated by the simulator. This file does not normally require editing, so will be overwritten each time *iGen* runs.

4.5 The macros file

The output file ***dmac.igen.macros.h*** contains C macros defining register offsets and bit fields. This is not required by the model but can be included in the user's peripheral driver code. This file does not normally require editing, so will be overwritten each time *iGen* runs.

4.6 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.

```
shell> igen.exe                         \
    --batch          dmac.tcl           \
    --writec         dmac               \
    --userheader     company.header.h
```

4.7 Building with standard Makefile.pse

The user build environment explained above is all included within the Makefile, Makefile.pse, that is included in Makefiles for peripheral generation and compilation.

This will perform:

1) Generation of peripheral C template from an iGen file, the expected file name is pse.tcl

2) The compilation of the peripheral template

The action performed to generate the peripheral template is the equivalent of

```
shell> igen.exe --batch pse.tcl \
                --writec pse    \
                --userheader IMPERAS_HOME/ImperasLib/fileheaders/refApache.txt \
                --overwrite
```

which will generates the standard set of C peripheral template files below

File	Contents
pse.igen.c	main(), constructors etc.
pse.igen.h	Function prototypes, storage for registers etc.
pse.igen.stubs	Stub functions to be expanded by the user
pse.macros.igen.h	Register offsets for application programs
pse.attrs.igen.c	modelAttrs structure and associated data.

And compile these files to a pse.pse file that can be loaded and executed by the PSE model by a simulator.

5 Examples

Writing behavioral C code for a peripheral model is discussed at length in *The Imperas Peripheral Modeling Guide.*

A set of examples that use iGen are available in *$IMPERAS_HOME/Examples/Models/Peripherals/creatingDMAC.*

The directory *4.interrupt* which contains the complete model:

5.1 pse.tcl

The TCL script that generates the model. The model name *dmac* is used to create other file names.

5.2 pse.igen.stubs

The user (stubs) file, generated by *iGen* in the example, but replaced by the completed file *dmac.user.c*

5.3 dmac.user.c

The user-modified file containing the filled-out stubs functions.

5.4 pse.igen.c

The main file, generated by *iGen*, which constructs the model.

5.5 pse.igen.h

The include file, generated by *iGen*, used by the other C files.

5.6 pse.attrs.igen.c

The interface specification file, generated by *iGen*, which contains the attributes table to be interrogated by the simulator when it loads the model.

5.7 pse.macros.igen.h

The macros file, generated by *iGen*, contains macros which define the relative addresses of registers and bit positions in the registers. It is not used by the model but can useful when writing applications that use this device.

6 The Peripheral Native SystemC TLM2.0 Interface

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

Referring to the file
`ImperasLib/source/national.ovpworld.org/peripheral/16450/1.0/tlm2.0/pse.igen.hpp`

The interface is a specialization of the generic functionality in
`ImperasLib/source/ovpworld.org/modelSupport/tlmPeripheral/1.0/tlm2.0/tlmPeripheral.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 icmPeripheral class in the CpuManager API. It creates an instance of icmSlavePort for each bus slave interface, icmMasterPort for each bus master interface (not used here) and uses an instance of icmOutputNetPort for each wire output. Like in CpuManager, the peripheral instance requires an instance name.

ICM bus master ports are mapped to TLM2.0 initiator sockets. ICM bus slave ports are mapped to TLM2.0 acceptor sockets.

The constructor can also pass user defined attributes to the model.

The constructor uses icmGetVlnvString to calculate the file path to the model then constructs the model, the bus interfaces and the wire interfaces.

6.1 Creating the peripheral model SystemC interface

Example:

```
shell> igen.exe                 \
  --batch pse.tcl              \
  --writetlm pse.hpp           \
  -–userheader company.header

…
… Writing peri
```

The –writetlm argument to *iGen* puts it into batch mode and specifies the TLM2.0 output file. No extension is added. The file will not be overwritten if it already exists.

##