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:	3.1
Filename:	iGen_Peripheral_Generator_User_Guide.doc
Last Saved:	Thursday, 11 February 2021
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`	code examples.
`code`	code, commands, filename
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, to print information about a specific command use:
`igen -apropos <command>`.

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 include empty functions that can be filled in by the writer to complete the model.

iGen can also generate a SystemC TLM2.0 interface for the model. See section 8. 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

This chapter describes the TCL commands that create a peripheral model template.

A peripheral model definition must define:

- the *VLNV* of the model
- *bus master* and *bus slave* ports
- *net* and *packetnet* inputs and outputs
- *parameters* to configure the 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

`imodelnewperipheral` begins the construction of the model. Other commands must follow until the model is complete. *iGen* reads the TCL script then writes the model C files when the script ends.

`imodelnewperipheral`	Start a new peripheral model
`-name <name>`	VLNV name of the new peripheral
`-vendor <vendor>`	VLNV vendor of new peripheral
`-library <library>`	VLNV library of new peripheral
`-version <version>`	VLNV version of new peripheral
`-extensionfile <file>`	Optionally specify an intercept library for this model
`-constructor <function>`	A function to be called at construction time
`-destructor <function>`	A function to be called at destruction time
`-saverestore`	Generate stub functions to save and restore the model state
`-nbyteregisters`	All registers and local memories use a new API supporting registers and memories more than 32 bits wide. This is the preferred option for new models
`-formalvalues`	Create a global variable of the same name for each model parameter, with code that sets the variable to the parameter value

If the peripheral model requires an intercept library, use `-extensionfile` to specify its whereabouts relative to the PSE model. The extension library file extension need not be specified.

3.1 Bus Master

A bus *master port* can initiate a bus transaction to read or write data to other bus devices.

imodeladdbusmasterport	Add a bus master port to the model
`-name <name>`	Name of the port
`-addresswidth <integer>`	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 <name>`	Specify the interrupt output net port to be asserted if a bus error occurs.

3.2 Bus Slave

A bus *slave port* receives bus read or write transactions.

imodeladdbusslaveport	Add a bus slave port
`-name <name>`	Name of the port
`-addresswidth <integer>`	Number of address bits that can be decoded
`-size <integer>`	Range of addresses this port can generate (in bytes)
`-mustbeconnected`	If specified, the simulator will fail to run if the port is unconnected
`-remappable`	The port's address will be determined at run-time
`-defaultfunctions`	Write code to report accesses that do not address a register or local memory

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 will be set up during the peripheral model execution.

A slave port can contain one or more address blocks. Address blocks can contain memory mapped registers and/or local memory.

3.2.1 Bus Slave implementation

A bus slave port is connected to a bus by the platform. It maps the specified range of addresses to memory in the PSE's address space. This memory can have memory mapped registers or local memory callbacks mapped onto it using `imodeladdaddressblock`, `imodeladdmmregister` and `imodeladdlocalmemory`

Gaps in the slave port region that have no memory mapped registers or local memory remain mapped to the PSE's memory space. Reads and writes from the connected bus to these regions will silently access this space. If you need to know if accesses are reaching

the gaps then specify the `-defaultfunctions`. This installs callbacks on the whole region which use `bhmMessage()` to report the size of offset of each access.

3.3 Net Port

A net carries a 32 bit value which usually using zero to represent logic 0 and non-zero to represent logic 1.

imodeladdnetport	Add a net port to the model
`-name <name>`	Name of the net port
`-mustbeconnected`	If specified, the simulator will fail to run if the port is unconnected
`-type <type>`	Type must be `input`, `output` or `inout`
`-updatefunction <function>`	A function to be called when the net is written.
`-updatefunctionargument <value>`	A value to pass to the update function

3.4 Packetnet Port

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>`	Name of the packetnet port
`-mustbeconnected`	If specified, the simulator will fail to run if the port is unconnected
`-maxbytes <integer>`	Maximum size of a packet, checked at run-time
`-updatefunction <function>`	Function to be called when the packetnet is written.
`-updatefunctionargument <value>`	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.

3.5 Formal Parameter

imodeladdformal	Add a formal to the model
`-name <name>`	Name of the formal attribute
`-type <type>`	The formal type (see below)
`-min <number>`	Optional minimum value for an integer parameter
`-max <number>`	Optional maximum value for an integer parameter
`-defaultvalue <number>`	Optional default value for the parameter

A *formal* defines a parameter that the model will accept. If the flag `-formalvalues` was supplied to `imodelnewperipheral` then code will be generated to fetch the parameter's

value from the platform. If not then it is expected that the functions `bhmInt32ParamValue(), bhmStringParamValue()`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, flag`	Boolean value	n
`double, float`	C double-precision floating point	y
`endian`	Predefined enumerated type taking values `big` or `little` and set to a member of `bhmEndian`	n
`enum, enumeration`	Select one from a set names	n
`int32, integer`	32 bit signed integer	y
`int64`	64 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 using `imodeladdenumeration`;

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

3.6 Address Block

An *Address block* must be nested inside a slave port. An address block defines a memory region wholly within the address region decided by a slave port. A slave port can contain more than one address block. Address blocks cannot overlap their addresses. An Address block can contain memory mapped registers and/or local memories.

`imodeladdaddressblock`	Add an Address block to a slave port
`-name <name>`	Name of the address block
`-port <name>`	Name of an existing bus slave port
`-offset <integer>`	Lowest address relative to the base of the slave port
`-size <integer>`	Number of bytes in the block's address range
`-width <integer>`	Word width in bits
`-loarray <integer>`	Lower bounds of array of address blocks
`-hiarray <integer>`	Upper bounds of array of address blocks

3.7 Address Block Array

A peripheral device might contain replicated components, each with identical groups of registers. Although TCL loops can be used to replicate registers, address blocks or ports,

this leads to large and cumbersome generated code. Register arrays and address block arrays provide a better alternative.

Address block arrays are constructed using –loarray and –hiarray arguments:

```
imodeladdaddressblock –loarray <integer> –hiarray <integer>.
```

The –loarray and –hiarray arguments set the numerical lower and upper inclusive bounds of the array.

Address blocks are generated with _<integer> in the name, corresponding to the requested numbers. The C structures defined to store the contents of registers in the address block are not replicated, but declared as an array.

Note that C arrays always have indices from 0 to <n-1> so must be accessed this way.

The macros header file contains a definition of the base address of the address block, which takes an index parameter to select an array member.
The index must take values from 0 to <n-1> and not the range specified by –loarray and –hiarray.

If callbacks are specified (using –readfunction, -writefunction and -viewfunction) Then the generated code contains a calculation of the address block index for use in the function.

3.7.1 Array Names

The address block name is derived from the parameters

```
-name     <name>
-loarray  <lo>
-hiarray  <hi>
```

The C name is <name>[<hi> - <lo>]
Characters illegal in C are replaced with '_'.
If <name> is a C keyword, it has '_' prepended.
%u or %d is removed.

The user-visible names are in the range
<name>_<hi> to <name>_<lo>

unless <name> contains %u or %d in which case the %u or %d is replaced by the number.

3.8 Local memory

A local memory is used to model memory inside the peripheral, accessible from outside, through a port. It must be nested inside an address block. The –size (in bytes) and –offset (from the base of the address block) must be specified.

`-readfunction, -writefunction and –changefunction` specify functions to be called when the local memory is accessed. Stub functions matching their prototypes will be created in the user file.

If no function is supplied the region will be implemented as memory in the PSE address space than can be read or written by code in the model. Local memory cannot overlap other memories or memory mapped registers.

imodeladdlocalmemory	Add a local memory to an address block
`-access <access>`	`<access>` can be `r,w, rw` (read, write or read/write)
`-name <name>`	Name of the memory
`-addressblock <name>`	Name of an existing address block
`-offset <integer>`	Lowest address relative to the base of the address block
`-size <integer>`	Number of bytes in the memory
`-readfunction <function>`	A function called when a memory is read
`-writefunction <function>`	A function called when a memory is written
`-changefunction <function>`	A function called when a memory is written with a new value
`-userdata <value>`	A value passed to each of the above functions
`-nonvolatile`	Writing the memory does not cause a callback if the value is unchanged.
`-nbyte`	Uses a more efficient interface and supports widths greater than 32 bits

3.9 Memory mapped register

A memory mapped register is used to model the registers typically used in a peripheral device. `imodeladdmmregister` generates code to create the register and, if requested, generates empty callback functions. Memory mapped registers created this way will appear in the model's documentation and be visible to the Imperas Debugger.

Register widths are specified in bits but the implementation rounds the size of storage up to the nearest byte.

A memory mapped register must be the same width or narrower than its containing address block.

A memory mapped register must not overlap other memories or memory mapped registers in the address block.

imodeladdmmregister	Add a memory mapped register
`-name <name>`	Name of the register
`-access <access>`	`<access>` can be `r,w,rw` (read, write or read-write)
`-addressblock <name>`	Name of an existing address block
`-offset <integer>`	Offset relative to the base of the address block
`-width <integer>`	Number of bits in the register (multiple of 8)
`-readfunction <function>`	function called when read

-writefunction <function>	function called when written
-viewfunction <function>	function called when read by a debugger or tool
-userdata <value>	value passed to callbacks
-nbyte	Use the new (preferred) API
-writemask <value>	Write only these bits (see imodeladdbitfield)
-nonvolatile	Writing the register does not cause a callback if the written value is unchanged.
-loarray <integer>	Lower bound of the array of registers
-hiarray <integer>	Upper bound of the array of registers

Setting the –nbyte argument generates code that uses ppmCreateNByteRegister(). instead of the deprecated ppmCreateRegister().

The –access argument controls how the register reacts to reads and writes.
If –nbyte is specified, the –access argument controls the readable and writable arguments to ppmCreateNByteRegister(). Setting –access w calls ppmCreateNByteRegister() with readable=0 which causes a bus error is the register is read. Setting –access r calls ppmCreateNByteRegister() with writable=0 which causes a bus error is the register is written.

Arguments -readfunction, -writefunction and –viewfunction specify functions to be called when the MMR is accessed. Stub functions matching their prototype will be created in the user file. The view function should not change the state of the peripheral.

Defining a memory mapped register will define storage for the register's value in a global structure the model. The value will be a union of the integer value with the bit-fields, if defined.

3.10 Register Array

A peripheral device might contain replicated components, each with identical registers or groups of registers. Although TCL loops can be used to replicate registers, address blocks or ports, this leads to large and cumbersome generated code. Register arrays and address block arrays provide a better alternative.

Arrays are constructed using –loarray and –hiarray arguments which arguments set the numerical lower and upper (inclusive) bounds of the array:

```
imodeladdmmregister  –loarray <integer>  –hiarray <integer>
```

Registers are generated with _<integer> in the name, corresponding to the requested numbers, unless the sequence %u is included somewhere in the name in which case the %u is replaced with the number. The C structures defined to store the contents of the registers are not replicated, but declared as an array.

Note that C arrays always have indices from 0 to <n-1> so must be accessed this way.

The macros header file contains a definition of the address of the register, which takes an index parameter to select an array member.
The index must take values from 0 to <n-1> and not the range specified by –loarray and –hiarray.

3.10.1 Which register is accessed?

If callbacks are specified (using –readfunction, -writefunction and -viewfunction) the generated code contains a calculation of the register index for use in the function.

Alternatively, if the register is created with –userdata _index (assuming there is no other requirement for userdata), then the userdata field is supplied with the index.

3.10.2 Array Names

The register name is derived from the parameters

```
-name     <name>
-loarray  <lo>
-hiarray  <hi>
```

The C name is <name>[<hi> - <lo>]
Characters illegal in a C identifier are replaced with '_'.
If <name> is a C keyword, it has '_' prepended.
%u or %d is removed.

The user-visible names are in the range
<name>_<hi> to <name>_<lo>

unless <name> contains %u or %d in which case it is substituted for the number.

3.11 Combination arrays

Note that an array of address blocks can contain an array of registers. The structures generated to store the register contents require two indices.

3.12 Register bit-field

A memory mapped register can be divided into bit-fields. The fields must be the same width or narrower than the register. Fields can be supplied in any order but must not overlap. Parts of the register without fields will be padded. The specified access cannot be higher that the containing register (e.g. a read-only register cannot contain a writeable bit field).

imodeladdfield	Add a field to a memory mapped register
-name <name>	Name of the field
-access <access>	Access= r,w,rw meaning read, write or read-write

-mmregister <name>	Name of an existing memory mapped register
-bitoffset <integer>	Offset relative to the LSB of the register.
-width <integer>	Number of bits in the field
-reset <value>	Reset value

imodeladdfield generates C bit field entries in the structure that stores the register's value. The reset value will be applied to the field during model initialization or when a designated reset input is activated.

The -access option appears in the documentation for the field and if the register read and write functions are not supplied, will be honored by the simulator. However, if read and write functions are supplied, the simulator cannot prevent them accessing the bit-field.

3.13 Reset

An input net port can be named as a *reset* input and associated with one or more memory mapped registers.

imodeladdreset	Designate a reset capability to a net port
-name <name>	Name of an existing input port
-mmregister	Name of an existing memory mapped register
-mask <value>	Specify which bits of the register will be reset (others are unchanged)
-value <value>	Specify the reset value

This will generate code to set the register contents during initialization and when the reset input is written to a non-zero value. Thus, the reset is edge-triggered and active high.

If an MMR has bit fields it is preferable to specify the reset value by omitting the -mask and -value flags and instead using

```
imodeladdfield –reset <value>
```

which limits each field to 64 bits maximum size rather than

```
imodeladdreset -mask <value> –value <value>
```

which limits the complete register reset mask and value each to 64 bits.

3.14 Documentation

Documentation fields can be added to most objects mentioned in this chapter. Their text is embedded in the model, can be accessed through an API and will appear in documentation produced by other Imperas tools.

iadddocumentation	Add a documentation entry to a model or object

`-name <name>`	name of the entry
`-text <text>`	Content
`-handle <path>`	Optional 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 entries can 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 most recently created object. If specified, the handle should match the full name of the object (which is the string returned by the `imodeladd` commands).

Documentation entries also have handles. Adding documentation to an existing document handle produces a sub-section.

4 Example 1 : DMA

The example `Examples/Models/Peripherals/creatingDMAC/5.nativeBehaviour` shows a peripheral model using the TCL specification `Examples/Models/Peripherals/creatingDMAC/5.nativeBehaviour/peripheral/pse/pse.tcl`

4.1 Introducing the model

`imodelnewperipheral` starts creating the peripheral:

```
imodelnewperipheral \
    -name        dmac \
    -vendor      ovpworld.org  \
    -library     peripheral \
    -version     1.0 \
    -constructor constructor \
    -destructor  destructor  \
    -nbyteregisters \
    -endianparam endian \
    -formalvalues \
    -extensionfile ../model
```

Arguments `-constructor constructor` and `–destructor destructor` create stub constructor and destructor functions (stub functions are written in the file `pse.c.igen.stubs`):

```
PPM_CONSTRUCTOR_CB(constructor) {
    // YOUR CODE HERE (pre constructor)
    periphConstructor();
    // YOUR CODE HERE (post constructor)
}

PPM_DESTRUCTOR_CB(destructor) {
    // YOUR CODE HERE (destructor)
}
```

Argument `-nbyteregisters` makes *iGen* use `ppmCreateNByteRegister()` for all registers.

Argument `-endianparam endian` creates a formal parameter called *endian* which is passed to `ppmCreateNByteRegister()` so that all registers greater than 8 bits will read and write registers with the requested byte order.

Argument `–formalvalues` creates a global variable with the name and type of each formal parameter and generates code to initialize the variable to the parameter value.

Argument `–extensionfile ../model` tells the model to load the model's intercept library from the given directory.

4.2 Documentation

The line

```
iadddocumentation -name Description -text "DMAC peripheral model"
```

Creates a documentation node. This will produce an entry in the automatically generated document.
`iadddocumentation` is used throughout the example to add documentation to ports, parameters, registers and fields.

4.3 Bus ports

These lines create bus master ports:

```
imodeladdbusmasterport -name MREAD   -addresswidth 32 -mustbeconnected
imodeladdbusmasterport -name MWRITE  -addresswidth 32 -mustbeconnected
```

`-mustbeconnected` makes the simulator check that the ports are connected and raises an error if not.

This line creates the bus slave port used to read and write the programming registers. The port will occupy 0x140 bytes of space on the connected bus (there are gaps without registers in this space).

```
imodeladdbusslaveport -name DMACSP -size 0x140 –mustbeconnected
```

4.4 Nets

The model has one input port for resetting the device and one output port for interrupting a processor when a DMA operation is complete.

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

4.5 Address Blocks and registers

The first address block contains 8-bit registers addressed on 32-bit boundaries. Its base is at the base of the region decoded by the slave port.

```
imodeladdaddressblock -port DMACSP -name ab8 -width 8 -offset 0 -size 0x40

imodeladdmmregister -addressblock DMACSP/ab8 -name intStatus    -offset 0x00 -access r
imodeladdmmregister -addressblock DMACSP/ab8 -name intTCstatus  -offset 0x04 -access rw \
                    -writefunction TCclearWr
imodeladdmmregister -addressblock DMACSP/ab8 -name rawTCstatus  -offset 0x14 -access r
imodeladdmmregister -addressblock DMACSP/ab8 -name enbldChns    -offset 0x1C -access r
```

This register contains a bit field:

```
imodeladdmmregister -addressblock DMACSP/ab8 -name config       -offset 0x30 \
                    -access rw -writefunction configWr

imodeladdfield      -mmregister DMACSP/ab8/config -name burstSize \
                         -bitoffset 0 -width 2  -reset 0
```

An array of two address blocks 32 bits wide has offsets from the base of the slave port of 0x100 and 0x120.

The `config` register callback `configChWr` needs to know the channel (address block) that is being written; the –`userdata _index` argument to `imodeladdregister` causes the address block index (value 0 or 1) to be passed to its `userData` parameter.

```
# Array of 2 sets of 32 bit registers, one per DMA channel

set adrBlk32 "ab32"

imodeladdaddressblock -name ${adrBlk32} \
                  -width 32 \
                  -offset 0x100 \
                  -size 0x20 \
                  -loarray 0 \
                  -hiarray 1

imodeladdmmregister -addressblock ${slvPrt}/${adrBlk32} -name srcAddr -offset 0x0 -access rw
imodeladdmmregister -addressblock ${slvPrt}/${adrBlk32} -name dstAddr -offset 0x4 -access rw

imodeladdmmregister -addressblock ${slvPrt}/${adrBlk32} -name control -offset 0xC -access rw
imodeladdfield      -mmregister ${slvPrt}/${adrBlk32}/control \
    -name      transferSize \
    -bitoffset 0 \
    -width     12

imodeladdmmregister -addressblock ${slvPrt}/${adrBlk32} \
    -name config \
    -offset 0x10 \
    -access rw \
    -writefunction configChWr \
    -userdata _index

imodeladdfield      -mmregister ${slvPrt}/${adrBlk32}/config -name enable -bitoffset 0  -width 1
imodeladdfield      -mmregister ${slvPrt}/${adrBlk32}/config -name inten  -bitoffset 15 -width 1
imodeladdfield      -mmregister ${slvPrt}/${adrBlk32}/config -name halt   -bitoffset 18 -width 1

iadddocumentation -name Description -text "source address" -handle ${slvPrt}/${adrBlk32}/srcAddr
iadddocumentation -name Description -text "dest address"   -handle ${slvPrt}/${adrBlk32}/dstAddr
iadddocumentation -name Description -text "control"        -handle ${slvPrt}/${adrBlk32}/control
iadddocumentation -name Description -text "configuration"  -handle ${slvPrt}/${adrBlk32}/config
```

Note that `imodeladdaddressblock` need not specify a bus slave port; the most recently created port is used.

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

4.6 Reset

These lines designate the input net port RESET as a reset input and bind it to the given registers:

```
imodeladdnetport -name RESET -type input

imodeladdreset -name RESET -mmregister ${slvPrt}/${adrBlk8}/intStatus     -value 0
imodeladdreset -name RESET -mmregister ${slvPrt}/${adrBlk8}/intTCstatus   -value 0
imodeladdreset -name RESET -mmregister ${slvPrt}/${adrBlk8}/rawTCstatus   -value 0
imodeladdreset -name RESET -mmregister ${slvPrt}/${adrBlk8}/enbldChns     -value 0

imodeladdreset -name RESET -mmregister ${slvPrt}/${adrBlk8}/config
```

Note that the first 4 registers have no bit fields; the flag `–value` specifies their reset value. Register `config` has a bit field which specifies its own reset value for each bit field:

```
imodeladdfield -mmregister DMACSP/ab8/config -name burstSize \
               -bitoffset 0 -width 2 -reset 0
```

When defining the reset value of a field in the register the register itself must still be bound to the reset signal i.e. you cannot specify a reset value using `imodeladdfield -reset` without also having included `imodeladdreset` for the register

5 Example 2: Interrupt Controller

The example `Examples/Models/Peripherals/registerArrays` shows a peripheral model using the TCL specification `Examples/Models/Peripherals/registerArrays/intc/pse/pse.tcl`

The model is for illustration of programming techniques and does not represent a real component.

5.1 Introducing the model

The model `intc` is a dual channel, 32 input, single output interrupt controller. Each channel can be enabled separately as can each input. Each input can be assigned a priority from 1 to 15, though the example allows only one interrupt to be active at a time.

For convenience and compactness, the model uses arrays of address blocks and output nets to model the dual channels and arrays of registers and input nets to model the functions common to each input.

5.2 TCL description

The TCL is in `Examples/Models/Peripherals/registerArrays/intc/pse/pse.tcl`

The command `imodelnewperipheral` starts construction of the model, sets the VLNV, names constructor and destructor functions and requests use of the (newer) n-byte register interface.

The next few commands `iadddocumentation` add to the documentation that can be generated by the model.

The commands `imodeladdnetport` create the `systemReset` input, the interrupt inputs `intin0` to `intin63` and the interrupt outputs `intout1` and `intout2`. Note that the arguments `-loarray` and `-hiarray` set the (inclusive) bounds of the names of members of arrays of net ports. Handles to the net ports are declared as C arrays which allows computed access to the nets.

The command `imodeladdbusslaveport` creates a bus port which is used to read and write the memory mapped registers.

The command `imodeladdaddressblock -name ch%u -loarray 0 -hiarray 1` creates an array of address blocks corresponding to the identical channels. The names of the address blocks (as visible to the user) has the `%u` substituted with the array indices; 0 and 1 in this case. the `-size` argument sets the size of the address block in bytes and the address interval between members. If required the size could be increased to create unoccupied address space between members.

The command

```
imodeladdmmregister -addressblock sp1/ch%u -name pinControl%u -loarray 0 -hiarray 31
```

creates an array of registers in the given address block. Note that `ch%u` must match the address block name. The `-offset` argument set the offset of the first member of the array. Following members have addresses that increment by the size of the register rounded to the nearest byte.

The `-writefunction <function name>` argument causes a function of that name to be generated in the file pse.c.igen.stubs.

The command `imodeladdfield -mmregister sp1/ch%u/pinControl%u -name enable` creates a bit field in the register. Note that `ch%u` and `pinControl%u` must match the names of the address block and register parents.

5.3 Generated code

Having run the example, refer to `pse.igen.h` and look for `Register data declaration`. The structure `sp1_ch_dataT` declares storage for registers in the `sp1` slave port region. Note that structure `sp1_ch_data[]` is an array corresponding to the address block array and that `pinControl[]` is an array corresponding to the array of registers. Now look for `Port handles`. The structure definition `handlesT` contain the net port handles `systemReset` and the net port handle arrays `intin[]` and `intout[]`.

The file `pse.c.igen.stubs` contains the write callback functions `writeChannelControl` and `writePinControl`. The functions contain code to calculate the address block and register indices, and to use them to update the register values. As in the other examples, this file can be copied to `user.c` to form the basis of the behavioural code of the model.

Now refer to user.c. The function `raiseInterrupt()` uses the address block array index `ch` and the register index `input` to compute the register to update, and uses `ch` to write to the output net port using the array of handles `handles.intout[]`.

6 Running iGen

iGen has many command line options. These used to write a peripheral model template:

iGen Argument	File
`-batch  <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 <file>`	Prepend this text input file to each generated file (it must be legal C or C comments).
`-overwrite`	Overwrite the output stubs file if it already exists (this is otherwise an error)

The output file names are derived from the argument 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 intended to be modified by the developer, so this file will not be overwritten in subsequent runs 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
`dmac.igen.h`	Function prototypes, storage for registers
`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()`, construction.
`dmac.igen.h`	Function prototypes, storage for registers
`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.

6.1 The user stubs file

The user stubs file `dmac.igen.stubs` contains stub functions. *iGen* creates this file to be completed by the user. Stub functions are created for each callback specified when creating registers, local memories and input ports and also for the model constructor and destructor functions.

6.2 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.

6.3 The include file

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

6.4 The attributes file

The output file `dmac.attrs.igen.c` contains the `modelattrs` structure that will be interrogated by the simulator when the model is loaded. This file does not normally require editing, so will be overwritten each time *iGen* runs.

6.5 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.

6.6 Adding a standard header

Some organizations require each source file to begin with a Copyright message. Text can be prepended to all generated files using `--userheader`.

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

6.7 Building with standard Makefile.pse

The file `Makefile.pse` is provided as part of the Imperas environment.

This will:

1) Generate the peripheral template from an `iGen` TCL file called `pse.tcl`
2) Compile and link the peripheral model, creating the executable `pse.pse`

7 Examples

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

A set of examples that use iGen are available in
`$IMPERAS_HOME/Examples/Models/Peripherals/creatingDMAC`
and
`$IMPERAS_HOME/Examples/Models/Peripherals/registerArrays`

The directory `4.interrupt` contains the complete model. Files names containing `.igen.` are created by *iGen.*

7.1 pse.tcl

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

7.2 pse.c.igen.stubs

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

7.3 dmac.user.c

This file contains the behavior of the peripheral model. The programmer started with pse.c.igen.stubs and modified it to produce this file, adding actions to the existing empty functions and adding new functions where required.

7.4 pse.igen.c

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

7.5 pse.igen.h

This file, generated by *iGen*, is used in the other C files. It includes the API definition, defines structures to store registers values and port handles. It defines prototypes for exported functions.

7.6 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.

7.7 pse.macros.igen.h

This 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 be useful when writing applications that use this device.

8 The SystemC (TLM) Interface

OVP models can be used in a SystemC TLM environment (see OVPsim_Using_OVP_Models_in_SystemC_TLM2.0_Platforms).

An interface is required to connect each peripheral model to SystemC TLM2.0.

iGen can generate this interface; TLM interfaces shipped with the Imperas products are generated by *iGen*.

As an example, refer to the file:
`ImperasLib/source/national.ovpworld.org/peripheral/16450/1.0/tlm/pse.igen.hpp`

The interface is a specialization of the classes defined in:
`ImpPublic/include/host/tlm/tlmPeripheral.hpp`

It is implemented in a class with the same name as the peripheral model, `Uart16450` in this example. This class inherits `tlmPeripheral` which uses the OVP OP API to create and connect the peripheral model instance.

Here are the main classes used in the TLM interface:

OVP Object	OVP Class	TLM class
peripheral	`tlmPeripheral`	`sc_module`
bus slave port	`tlmBusSlavePort`	`simple_acceptor_socket`
bus master port	`tlmBusMasterPort`	`simple_initiator_socket`
input net port	`tlmNetInputPort`	`tlm_analysis_port`
output net port	`tlmNetOutputPort`	`tlm_analysis_port`

The `tlm_analysis_port` is used is preference to sc_signal because changes are propagated immediately rather than by the SystemC scheduler. To connect to `sc_signal` an interface class (not supplied) will be required.

8.1 Creating the peripheral model SystemC interface

There are two methods of creating an interface; from the compiled model executable or from a TCL specification. The former is preferred for two reasons;

- The C code of the model might not conform to the original TCL specification – the code can be edited.
- The C code of the model can be written to change the interface according to model parameters set by the platform in the model instance. For example, a parameter `inputs` could be used to control the number of input nets in the instance. The TLM interface must match the interface presented by the specific instance of the model.

If the user's design flow does not have these issues then either method can be used.

8.1.1 Creating the interface from the model

This example uses a pre-compiled peripheral model. iGen loads the model, interrogates its interface then writes the C++ file (In this example it first prepends the company copyright notice).
This method must be used if the model is to be configured using parameters that can change the physical connections that appear on the model, for example change the number of interrupt ports.

If the model is stored in the VLNV library, it can be specified by VLNV. In this example the model parameter `uart16550` is set true. In this case the parameter changes the model behavior but does not change the interface. Note that the parameter value is included in the generated interface to ensure the model is used in the same way as when the wrapper was generated

```
shell> igen.exe                          \
  --modelname    16450                   \
  --modelvendor  national.ovpworld.org   \
  --modellibrary peripheral              \
  --modelversion 1.0                     \
  --setparameter uart16550=1             \
  --writetlm     pse.hpp
```

Alternatively, if the model is not in a VLNV library or you just want to use a specific pse.pse file directly

```
shell> igen.exe                          \
  --modelfile  /home/user/bin/library/peripheral/pse.pse  \
  --writetlm   pse.hpp                   \
  -–userheader company.header
```

Here is some of the output:

The model requires the peripheral base class and because it uses both net and bus interfaces, the `tlmNetPort` and `tlmBusPort` headers are required:

```
#include "tlm/tlmPeripheral.hpp"
#include "tlm/tlmBusPort.hpp"
#include "tlm/tlmNetPort.hpp"
```

The body of the output is the specific peripheral class with the same name class of the model, limited by C++ rules. The class includes an instance of the wrapped model, bus and net port instances matching the ports in the model. The constructor set the name and VLNV of the model instance then initializes the ports. The instance name will be passed from the platform that instances this wrapped model.

```
class _16450 : public tlmPeripheral
{
  private:
    params paramsForPeripheral(params p){
        p.set("uart16550", (Bool)1);
```

```
        return p;
    }

     params paramsForPeripheral(){
        params p;
        p.set("uart16550", (Bool)1);
        return p;
    }
    const char *getModel() {
        return opVLNVString (NULL, "national.ovpworld.org", "peripheral",
"16450", "1.0", OP_PERIPHERAL, 1);
    }

  public:
    tlmBusSlavePort bport1;
    tlmNetOutputPort intOut;

    _16450(tlmModule &parent, sc_module_name name)
        : tlmPeripheral(parent, getModel(), name, paramsForPeripheral())
        , bport1(parent, this, "bport1", 0x8) // static
        , intOut(parent, this, "intOut")
    {
    }

    _16450(tlmModule &parent, sc_module_name name, params p)
        : tlmPeripheral(parent, getModel(), name, paramsForPeripheral(p))
        , bport1(parent, this, "bport1", 0x8) // static
        , intOut(parent, this, "intOut")
    {
    }

}; /* class _16450 */
```

8.1.2 Creating the interface from TCL

This example uses TCL to create the interface. Obviously the same TCL must be used to create the model template, then the interface in the C code must not be changed.

```
shell> igen.exe              \
  --batch   pse.tcl          \
  --writetlm pse.hpp         \

##
```