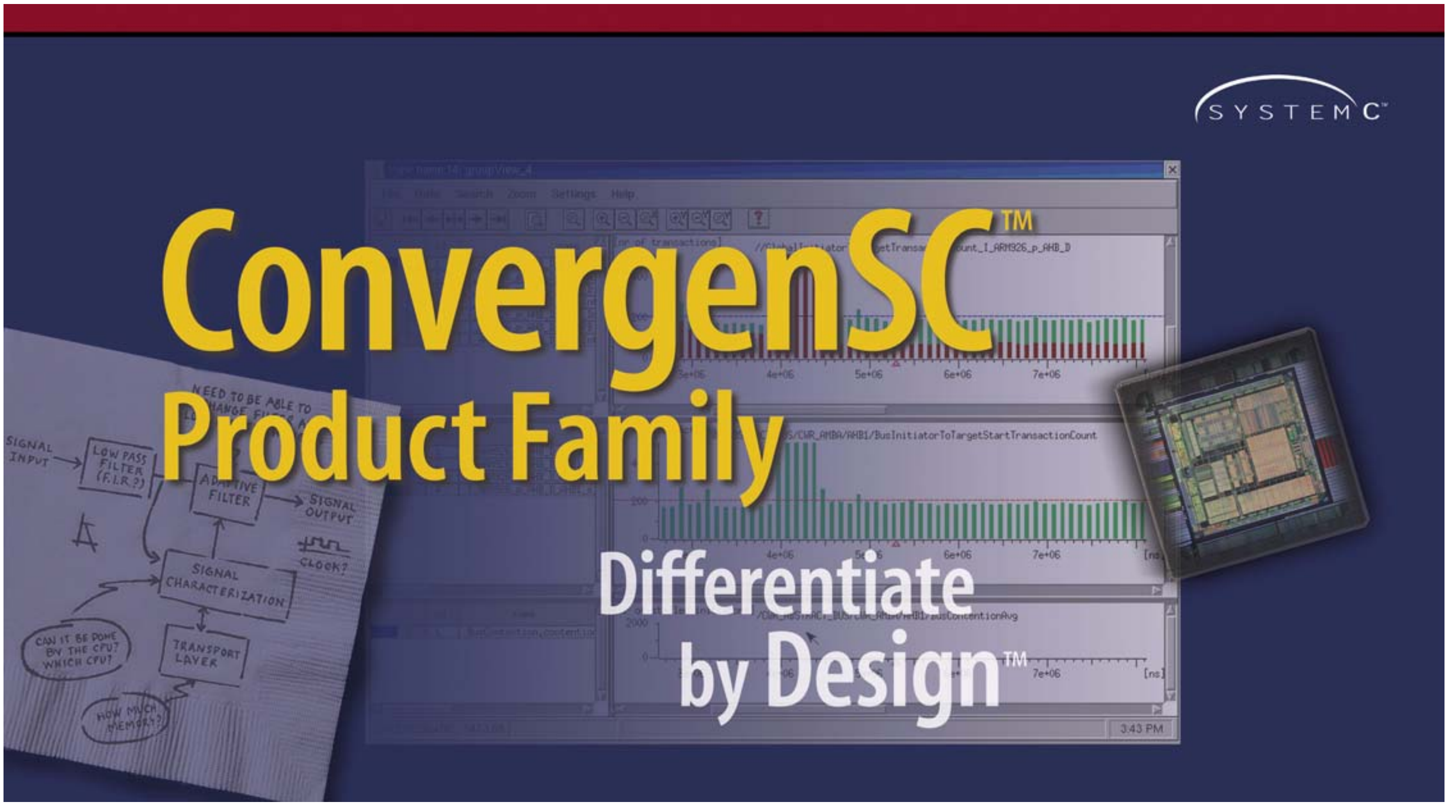


AMBA Bus Library

Version 2004.2.2 ■ January 2005

2121 North First Street
San Jose, CA 95131
USA

Main 408-436-4720
Fax 408-436-4740
www.CoWare.com

Contents

Preface

This manual describes the AMBA Bus Library, version V2004.2.2.

This version of the AMBA Bus Library is compliant with version V2004.2.2 of ConvergenSC running on Solaris 8 or Solaris 9 and Linux Red Hat 8 or Linux Red Hat Enterprise 3.

The AMBA Bus Library consists of the following:

- A bus library handler, which is an executable program used by Platform Creator to access and instantiate bus models
- An information file, containing the AMBA-specific bus information
- A TBS, containing the following bus models:
  - AHB and AHB-Lite Bus Model
  - Input-Stage and Output-Stage Bus Model
  - APB Bus Model
  - ***Lite2AHB*** bridge
  - ***Downsizer*** bridge
  - ***AHB2Lite*** block

This manual is organized as follows:

- Chapter 1, "Release Information," contains release information.
- Chapter 2, "Loading the AMBA Bus Library in Platform Creator," describes how to load the AMBA Bus Library in ConvergenSC Platform Creator.
- Chapter 3, "Protocols," describes the AHB and APB protocols.
- Chapter 4, "AHB and AHB-Lite Bus Model," describes the AHB and AHB-Lite Bus Model.
- Chapter 5, "Input-Stage and Output-Stage Bus Model," describes the Input-Stage and Output-Stage Bus Model.
- Chapter 6, "APB Bus Model," describes the APB Bus Model.
- Chapter 7, "Lite2AHB Bridge," describes the ***Lite2AHB*** bridge.
- Chapter 8, "Downsizer Bridge," describes the ***Downsizer*** bridge.
- Chapter 9, "AHB2Lite Block," describes the ***AHB2Lite*** block.
- Chapter 10, "Generating an RTL Bus," describes how to generate an RTL bus.
- Chapter 11, "User-Defined Arbiter," describes user-defined arbiters
- Chapter 12, "Using the Generated Bus Model in Incisive,"describes how to use the generated bus model in Incisive.
- Appendix A, "AMBA TLM API Quick Reference," describes the AMBA TLM API.

NOTE For information on the TLM API in general, see the ***TLM API Manual***.

The following describes:

- Terminology
- Customer Support

Terminology

- ***ADK*** stands for ARM Development Kit.
- ***AHB*** stands for Advanced High-performance Bus.
- ***APB*** stands for Advanced Peripheral Bus.
- ***API*** stands for Application Programmer's Interface.
- ***CLI*** stands for Command-Line Interface.
- ***GUI*** stands for Graphical User Interface.
- ***HDL*** stands for Hardware Description Language.
- ***PV*** stands for Programmer's View.
- ***RTL*** stands for Register-Transfer Level.
- ***SCV*** stands for SystemC Verification.
- ***TBS*** stands for Transactional Bus Simulator.
- ***TLM*** stands for Transaction-Level Modeling.
- ***VCD*** stands for Value-Change Dump.

Customer Support

If you have any problems with the software or documentation, please contact customer support via e-mail at one of the following addresses:

- ***support@CoWare.com***
- ***support.japan@CoWare.com***

Chapter 1

Release Information

This chapter describes:

- Compatibility Information
- Changes and Enhancements
- Fixed Problems
- Known Problems
- License Issues

Compatibility Information

The following lists compatibility issues that you must take into account in this release of the AMBA Bus Library.

NOTE When using the AMBA Bus Library of a previous release, you need to reinstall the AMBA Bus Library with the new release in order to work. For more information, see “ConvergenSC Model Library” in the ***ConvergenSC Installation Manual***.

- V2004.2.2
- V2004.2.1
- V2004.2.0
- V2004.1.1
- V2004.1.0

V2004.2.2

There are no compatibility issues in this release of the AMBA Bus Library.

V2004.2.1

- Migration Issue

Migration Issue

The AMBA protocol file has been updated to support the PV (Programmer's View) abstraction level. The protocol file now offers support for the untimed, PV, TLM, and RTL abstraction.

To upgrade designs from V2004.1.***x*** to V2004.2.0, you have to take the following steps:

1. Updating a User Block Library
2. Updating the Design

Updating a User Block Library

You have to recreate all user block libraries depending on one or more AMBA protocols and/or blocks. In other words, you must regenerate the XML file of this library using the latest protocols/blocks of the V2004.2 AMBA Bus Library.

There are two possibilities:

- The user block library comes as an XML file.
- The user block library is generated through a Tcl script.

The following describes these possibilities.

- The user block library comes as an XML file.

  Take the following steps:

  a. Open the user block library.
  b. Open the new ***AMBA BL*** library.
  c. Right-click on ***AMBA BL*** and from the pop-up menu select ***Update System Library***.
  d. Save the user block library.

- The user block library is generated through a Tcl script.

  The library is usually created by running a Tcl script in the Platform Creator GUI or ***pcsh***. So in this step you just have to rerun the Tcl script with the latest version of the ***AMBA BL*** library.

Updating the Design

There are two possibilities:

- The design is saved in a Platform Creator XML file.
- The design is created using a Tcl script.

The following describes these possibilities.

- The design is saved in a Platform Creator XML file.

  To update the design, take the following steps:

  a. In the Library Drawer of the Platform Creator window, close all libraries except the system library.

b Open the new ***AMBA BL*** library, the ARM PSP libraries, and the updated user libraries (see "Updating a User Block Library" on page 2).

c For each library (***AMBA BL***, ***ARM926EJS_AHB_PSP***, and so on), execute the ***Library > Update System Library*** command.

The design is updated to use the new AMBA protocols and blocks. The XML file can now be saved for further use.

- The design is created using a Tcl script.

  You do not need to modify the Tcl script. Just rerun it using the latest version of the ***AMBA BL*** library, ARM PSP libraries**,** and user libraries and everything will work fine.

V2004.2.0

- Migration Issue

Migration Issue

To support multiple bus libraries in one design, a namespace has been added to the AMBA protocol file. Up to V2004.1.***x***, you could only have one bus library in a platform, now you can have more.

To upgrade designs from V2004.1.***x*** to V2004.2.0, you have to take the following steps:

1. Updating the User Design Files
2. Updating a User Block Library
3. Updating the Design

Updating the User Design Files

There are three ways to update the user design files.

- Solution 1: Adding ***AMBA::*** explicitly to class names and enumerations defined in ***AMBA.h***. For example:

```
/* My platform */
#include "AMBA/AMBA.h"
class MyAPBSlave : public sc_module {
   public:
     // ports
     AMBA::APBTarget_inoutslave_port<12,32> p;
...
   void getTransaction () {
     p.getAddrTrf();
     if (p.AddrTrf->getType() == AMBA::tlmWriteAtAddress) {
     ...
...
```

  This is the preferred method, since it allows multiple bus library headers to be included in one file without name clashes.

  The disadvantage is that it has most impact on the code (multiple modifications per file).

- Solution 2: Adding ***using namespace AMBA;*** on top of the file**.** For example:

```
/* My platform */
#include "AMBA/AMBA.h"
using namespace AMBA;
class MyAPBSlave : public sc_module {
  public:
    // ports
    APBTarget_inoutslave_port<12,32> p;
...
  void getTransaction () {
    p.getAddrTrf();
    if (p.AddrTrf->getType() == tlmWriteAtAddress) {
    ...
```

  You are advised not to add this line in header files that are included in multiple files, because it will cause the namespace to be use in all these files.

  The advantage of this method is that it requires only a minimal modification in the source files of the platform.

  The disadvantage is that you can only use the protocols and enumerations of one bus library in files that have the ***using namespace*** line at the top or at the top of an included file.

- Solution 3: Adding ***-DNO_NAMESPACE_AMBA***. For example:

```
/* My platform */
#include "AMBA/AMBA.h"
class MyAPBSlave : public sc_module {
   public:
     // ports
     APBTarget_inoutslave_port<12,32> p;
...
   void getTransaction () {
     p.getAddrTrf();
     if (p.AddrTrf->getType() == tlmWriteAtAddress) {
    ...
```

  You leave all sources unmodified and add the ***-DNO_NAMESPACE_AMBA*** option for parsing in Platform Creator and for building in ***scsh***.

  In Platform Creator:

```
add_systemc_define NO_NAMESPACE_AMBA 1
```

  NOTE This solution only works for platforms with only one bus library.

Updating a User Block Library

You have to recreate all user block libraries depending on one or more AMBA protocols and/or blocks. In other words, you must regenerate the XML file of this library using the latest protocols/blocks of the V2004.2 AMBA Bus Library.

The library is usually created by running a Tcl script in the Platform Creator GUI or ***pcsh***. So in this step you just have to rerun the Tcl script with the latest version of the ***AMBA BL*** library.

Updating the Design

There are two possibilities:

- The design is saved in a Platform Creator XML file.
- The design is created using a Tcl script.

The following describes these possibilities.

- The design is saved in a Platform Creator XML file.

  To update the design, take the following steps:

  a In the Library Drawer of the Platform Creator window, close all libraries except the system library.

  b Open the new ***AMBA BL*** library, the ARM PSP libraries, and the updated user libraries (see "Updating the User Design Files" on page 3).

  c For each library (***AMBA BL***, ***ARM926EJS_AHB_PSP***, and so on), execute the ***Library > Update System Library*** command.

  The design is updated to use the new AMBA protocols and blocks. The XML file can now be saved for further use.

- The design is created using a Tcl script.

  You do not need to modify the Tcl script. Just rerun it using the latest version of the ***AMBA BL*** library, ARM PSP libraries**,** and user libraries and everything will work fine.

V2004.1.1

- Address as optional pin.

  The address pin of the ***AHBTarget***, ***AHBLiteTarget***, ***APBTarget***, and ***MLTarget*** protocols now have an optional address pin (***HADDR*** for AHB and multilayer protocols, ***PADDR*** for APB protocols). The change was made to enable the connection of peripherals occupying only one memory location. You may need to use the ***Tools > Propagate Port Properties*** menu command or its corresponding ***propagate_port_properties*** CLI command in Platform Creator to update the existing designs. For more information, see the ***Platform Creator User Manual***.

V2004.1.0

This release is not compatible with pre-V2004.1 versions of ConvergenSC, nor with V3.4 and pre-V3.4 versions of CoWare N2C.

Changes and Enhancements

The following lists the main changes and enhancements in this release of the AMBA Bus Library.

If the change or enhancement has been assigned a CoWare ID in the CoWare defect tracking system, this number is also listed.

- V2004.2.2
- V2004.2.1
- V2004.1.0

V2004.2.2

There are no changes and enhancements in this release of the AMBA Bus Library.

V2004.2.1

- Support for the PV (Programmer's View) abstraction level has been added to this release of the AMBA Bus Library.

V2004.1.0

- TLM port types have changed.

  The ***AMBA_*** prefix has been removed from the name of the type. Templates to specify address width and data width have been added. For more information on the new protocols, see the respective chapters.

  The old port types, prefixed ***AMBA_*** are still available in the ***AMBA.h*** header file. Systems dumped with an older version of Platform Creator can still be compiled using the new bus library. However, when creating new systems in Platform Creator, you should update the peripherals and use the new port types to enable Platform Creator to retrieve information on address and data width of the port.

  For example, ports previously using the type:

  ```
  AMBA_AHBLiteInitiator_inmaster_port
  ```

  should now be declared using the type:

  ```
  AHBLiteInitiator_inmaster_port<int address_width, int data_width>
  ```

  For more information, see Chapter 3, "Protocols," on page 15.

- Support for the ***Lite2AHB*** bridge has been added.

  This bridge allows the connection of an AHB-lite initiator to a multimaster AHB node.

  For more information on the usage of the ***Lite2AHB*** bridge, see Chapter 7, "Lite2AHB Bridge," on page 91.

- Support for the ***Downsizer*** bridge has been added.

  This bridge allows 32-bit peripherals or buses to be accessed with 64-bit accesses.

  For more information on the Downsizer bridge, see Chapter 8, "Downsizer Bridge," on page 95.

- Enumerated AMBA TLM transfer attributes are now shown as ASCII values in traces.

  When tracing attributes of an enumeration type (for example, ***AddrTrf.Type***) from ***scsh***, the values are dumped as ASCII strings in the VCD file. As a consequence, you may see some very wide numbers in the VCD trace. Formatting them as ***ascii*** in the VCD viewer will reveal the enumerated value names.

  The following transfer attributes will be traced as ASCII values:

  ```
  ReqTrf.ReqMode
  AddrTrf.Type
  AddrTrf.Kind
  AddrTrf.Group
  AddrTrf.BurstWrap
  AddrTrf.ProtectionType
  EotTrf.Status
  ```

Fixed Problems

The following lists problems that were found in the previous release of the AMBA Bus Library that have now been fixed.

If the problem has been assigned a CoWare ID in the CoWare defect tracking system, this number is also listed; otherwise, ***No ID*** is specified.

- V2004.2.2
- V2004.2.0
- V2004.1.4_IP4
- V2004.1.3
- V2004.1.2
- V2004.1.1
- V2004.1.0

V2004.2.2

- CWRsc04281: At the TLM abstraction level, grant became high two cycles after reset had been put off.

  Description: In case of a single master system, grant should be high one cycle after reset has been put off. At the TLM abstraction level, this actually took two cycles.

  Solution: This problem has been fixed and the relationship between reset and grant is now correct and resembles the one of BCA/HDL.

- CWRsc04930: AMBA RTL bus generation for a multilayer AHB with a user-defined arbiter failed.

  Description: If only one bit is necessary for signal ***AddrInPort***, the signal was ***sc_bool***, whereas it should be ***sc_uint<1>***.

  Solution: If only one bit is necessary for signal ***AddrInPort***, the signal is now ***sc_uint<1>***.

V2004.2.0

- CWRqa02963: Accessing a word-addressed AHB(-lite) target peripheral modeled at the pin-accurate abstraction level for a read with an access width smaller than the data width of a peripheral.

  Description: When accessing a word-addressed AHB(-lite) target peripheral modeled at the pin-accurate abstraction level for a read, with an access width smaller than the data width of a peripheral, the data received by the initiator is not correct.

  Solution: This problem has been fixed.

- No ID: The target of an AHB bus could not be an AHB bus.

  Description: It was not possible to connect an AHB bus to another AHB bus.

  Solution: This problem has been fixed.

V2004.1.4_IP4

- CWRsc04771: Output stage had bad name in case the number of input stages was more than five.

  Description: There was a problem in the output-stage generator that caused the output-stage generator to have a bad name if the number of input stages became more than five.

  Solution: This problem has been fixed. There is no limit to the number of connected input stages anymore.

V2004.1.3

- CWRqa03149: Peripheral with address width greater than memory region not supported unless a multiple of the size.

  Description: It was allowed to have a peripheral located at an address that was not a multiple of its size if the number of address bits of that peripheral reflected its memory size. However, if a peripheral had more address bits than necessary for its memory size (for example, 32 bits of address and memory size ***0x100000***), it was not allowed to put it at an address that was not a multiple of its memory size.

  Solution: There are no limitations anymore on the address configuration when using the AMBA HDL generator.

- CWRsc04226: AMBA generator stopped with a message about ***bitselect***.

  Description: The AMBA generator stopped with a message like this:

  ```
  $ ./amba_generator CwrModule_HARDWARE_CWR_BUS.xml amba.log
  High-value of bitselect should be higher or equal to low-value
  ... ... Offending component is 'bitselect' in block 'iAHB_MuxM2S'
  ...... input: iHADDRM[32]
  ...... output: wire_608[0]
  ```

  Solution: This problem has been fixed. Subtraction of subregions was not yet supported. It is implemented now.

- CWRsc04270: AMBA RTL generator forced IPs to have a fixed address in any system.

  Description: ***"amba_generator"*** did not allow IPs to have address regions, which start from ***0x0*** for relative address mapping. So, for region definition of IPs, system location of IP had to be used. This means the same IP for two address locations could not be used.

  Solution: This problem has been fixed.

V2004.1.2

- CWRqa03099: The arbiter of an AHB node sometimes rearbitrated at TLM abstraction level.

  Description: At the TLM abstraction level, the arbiter of an AHB node sometimes rearbitrated during busy cycles. This should not be the case.

  Solution: This problem has been fixed.

V2004.1.1

- CWRqa02985: Incorrect data terminals.

  Description: The AMBA protocols did not have the correct data terminals in RTL.

  Solution: The AMBA protocols have been modified to provide the correct data terminals in RTL:

  - The AHB initiator protocols (***AHBInitiator*** and ***AHBLiteInitiator***) now have ***HWRITE*** as optional terminal; ***HWDATA*** is not part of an inmaster, outslave protocol; ***HRDATA*** is not part of an outmaster, inslave protocol.
  - The AHB target protocols (***AHBTarget*** and ***AHBLiteTarget***) do not have ***HWDATA*** and ***HWRITE*** when an inmaster/outslave protocol is selected, and do not have ***HRDATA*** and ***HWRITE*** when an outmaster/inslave protocol is selected.

V2004.1.0

- CWRqa02666: Pin-accurate peripherals had to duplicate byte lanes when sending write or read data to the bus.

  Description: When a pin-accurate peripheral put data on the bus (so on the ***HRDATA*** signal on the target-port side, and on ***HWDATA*** on the initiator-port side), and the access size was smaller than the port width, the peripheral had to duplicate the data on all available byte lanes.

  For example, when writing data ***0xFFFF*** (16 bits), on a 32-bit data bus, you had to put data ***0xFFFFFFFF*** on the bus.

  Solution: This problem has been fixed. It is no longer necessary to duplicate the data.

- CWRqa02798: AHB 2 cycle error response was wrong for pin-accurate/TLM co-simulation in SystemC.

  Description: On the AHB, AHB-lite and output-stage nodes, when a target peripheral using a SystemC pin-accurate interface gave a two-cycle response, the ***HREADYin*** pin of the target was asserted one cycle later than the ***HREADYout*** pin. Also, in that case, the initiator would receive the ***EotTrf*** one cycle too late.

  Solution: This problem has been fixed.

- CWRsc03563: 64-bit wrapping burst did not work on a multilayer architecture.

  Description: On a multilayer architecture, wrapping bursts did not work correctly when ***AccessSize*** was equal to ***64***.

  Solution: This problem has been fixed.

- CWRsc03919/CWRsc03923: Pin-accurate co-simulation ***HMASTER*** pin on target interface was not modeled.

  Description: The ***HMASTER*** pin on the target interface was continuously zero.

  Solution: The ***HMASTER*** pin now shows the correct value, being the masters priority + ***1***.

Known Problems

The following lists known problems in this release of the AMBA Bus Library, and describes workarounds where available.

If the problem has been assigned a CoWare ID in the CoWare defect tracking system, this number is also listed; otherwise, ***No ID*** is specified.

- General
- AHB and AHB-Lite Bus Model
- Output-Stage Bus Model
- Downsize Bridge

General

- CWRqa03011: Linking in ***AMBABusModel*** in a simulation having ***SC_CTHREAD*** watching can give a segmentation fault.

  Description: Linking in ***AMBABusModel*** in a simulation having ***SC_CTHREAD*** watching can sometimes result in a segmentation fault (and hence a core dump). For example:

  ```
  SC_CTHREAD( do_clock, clk.pos());
  watching( !nReset.delayed());
  ```

  Workaround: Increasing the stack size for ***(C)THREAD*** solves the problem. For example:

  ```
  const int SC_DEFAULT_STACK_SIZE = 0x10000;

  SC_CTHREAD( do_clock, clk.pos());
  watching( !nReset.delayed());
  set_stack_size( SC_DEFAULT_STACK_SIZE << 2 );
  ```

- No ID: Possible issues with reset behavior of the AMBA Bus Library when using it in multicore simulations.

  Description: AMBA uses an asynchronous reset. The current reset implementation is, however, done in a clock thread, which is not correct. This could result in unexpected behavior at the time a reset is given when using the AMBA Bus Library in multicore simulations.

  Workaround: There is no workaround available.

- No ID: In Platform Creator, all AMBA nodes have a ***clk*** and a ***reset*** port at the PV abstraction level.

  Description: In Platform Creator it is not possible to have a node which has a ***clk*** and a ***reset*** port at one abstraction level, but does not have these ports at another abstraction level. Since all AMBA nodes can be used at the TLM and the HDL abstraction level, they do need a ***clk*** and a ***reset*** port. This implies that these ports will also be present even if you only use the AMBA nodes at the PV abstraction level. However, this is only the case in the Platform Creator representation. When the system gets exported, the ***clk*** and the ***reset*** port will not be created, and so during simulation at the PV abstraction level, you will not use a clock or a reset.

  Workaround: When you only want to use the AMBA Bus Library at the PV abstraction level, you can just instantiate a ***Clock*** initiator and ***Reset*** initiator in Platform Creator and connect these to the respective ***clk*** and ***reset*** ports of your nodes. To do this, select ***Edit > Add Initiator > Clock*** and select ***Edit > Add Initiator > Reset*** from the menu bar.

AHB and AHB-Lite Bus Model

- No ID: Split or retry responses are not supported for pin-accurate peripherals.

  Description: When connecting a pin-accurate peripheral to an AHB bus, split/retry responses will trigger a core dump.

  Workaround: There is no workaround available.

Output-Stage Bus Model

- No ID: An unnecessary arbitration cycle is possible when using a PSP with a multilayer structure.

  Description: When you use an output stage in your system and you select the fixed arbitration scheme, it normally takes one clock cycle before the output stage allows an input stage to do accesses. In one specific occasion, this arbitration cycle is not necessary and the input stage can immediately start doing accesses on the output stage.

  The following conditions must be met:

  - An input stage does accesses on an output stage (there was one clock cycle, necessary for the arbitration).
  - When the input stage stops doing transactions on this output stage, it does not do any accesses to another output stage, but keeps on sending idle transactions to the same output stage.
  - No other input stage tries to do accesses to this output stage.
  - The original input stage stops doing idle transactions to this output stage, but now again initiates read or write transactions to the same output stage.

  When these conditions are met, the output stage will immediately allow the input stage access and no arbitration cycle is necessary.

  However, having a PSP which keeps on sending idle cycles when there are no “real” transactions to be done is not desirable, since this would decrease the simulation speed at the TLM abstraction level dramatically. So a transaction initiated by a PSP will always have this one clock cycle for getting access to an output stage. This is not the case at the HDL level.

  Workaround: There is no workaround available.

Downsize Bridge

- No ID: If a ***DownSizer*** bridge is used in your design, global analysis will not work.

  Description: Global analysis can not handle the fact that a 64-bit transaction is split into two 32-bit transactions. When you enable global analysis, the analysis tool will give errors on nonexisting transaction IDs .

  Workaround: There is no workaround available.

License Issues

To use the AMBA Bus Library, a ***tlm_sim_amba*** license is necessary.

When you use the AMBA Bus Library in Incisive, an additional license called ***transactional_bus_simulator*** is necessary.

For more information, please contact ***sales@coware.com***.

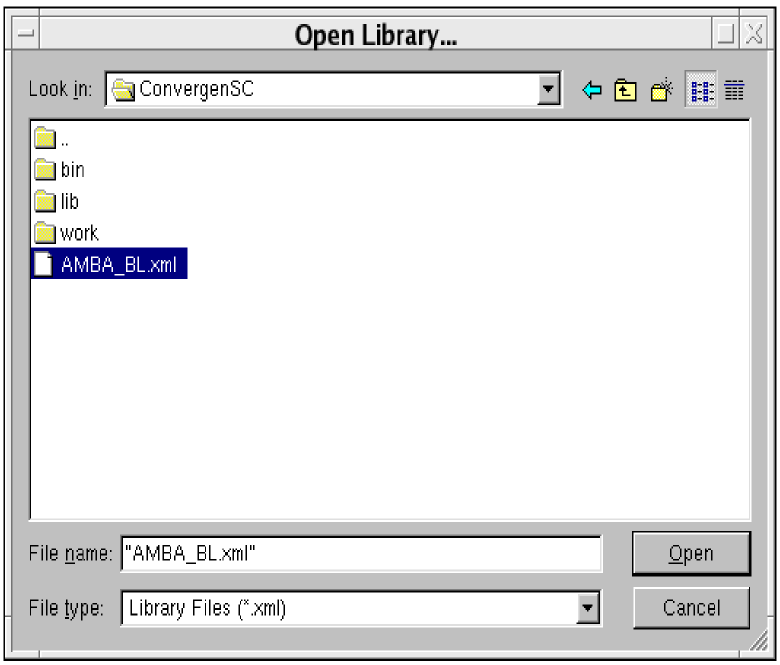
Chapter 2

Loading the AMBA Bus Library in Platform Creator

This chapter describes how to load the AMBA Bus Library in ConvergenSC Platform Creator.

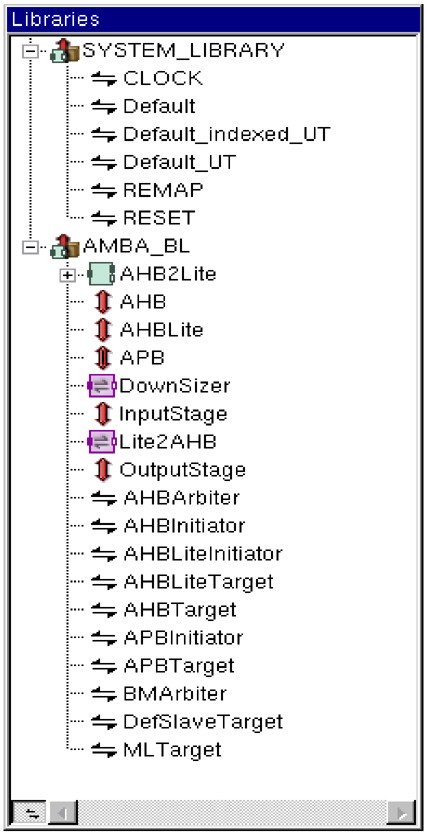
To load the AMBA Bus Library in Platform Creator:

In the Platform Creator GUI, select the XML file called ***AMBA_BL.xml***, as shown in the following figure. This file resides in the ***IPLOCATION/AMBA_BL/ConvergenSC/*** directory.



For more information about loading libraries in Platform Creator, see “Accessing Libraries” in the ***Platform Creator User Manual***.

After loading the library, you should see the following items in the Library Drawer of the Platform Creator GUI.



Chapter 3

Protocols

The following protocols are available in the AMBA Bus Library:

- ***DefSlaveTarget*** [1]
- ***APBInitiator***[1]
- ***APBTarget***
- ***AHBInitiator***
- ***AHBTarget***
- ***AHBLiteInitiator***
- ***AHBLiteTarget***
- ***MLTarget*** [1]

This chapter describes:

- Protocols and Abstraction Levels
- AHB Protocols
- APB Protocols

Protocols and Abstraction Levels

- PV, TLM, and Pin-Accurate Protocols
- TLM-HDL Co-Simulation and Its Limitations

PV, TLM, and Pin-Accurate Protocols

Each of the protocols available in the AMBA Bus Library can be used at the PV, TLM, or pin-accurate abstraction level.

- Peripherals using PV-level protocols use PV ports and the PV API.
- Peripherals using TLM-level protocols use TLM ports and the TLM API.
- Pin-accurate protocols are used to connect peripherals to the bus using SystemC signals.

You can mix the TLM and the pin-accurate abstraction levels. For example, to a TLM node you can connect a mix of TLM peripherals and pin-accurate peripherals.

1. These protocols are currently for internal usage only. You should not attempt to create peripherals using these protocols

TLM-HDL Co-Simulation and Its Limitations

When using a pin-accurate protocol, a TLM-HDL co-simulation can be achieved. This allows connection of HDL peripherals to a TLM bus model.

To be able to connect an HDL peripheral to a TLM bus, the HDL peripheral must be wrapped in a proxy module (for more information, see “SystemC-HDL Co-Simulation” in the ***Simulation and Debugging Manual)***. This will enable the translation of HDL signals to SystemC signals.

Take into account that the bus model is a TLM bus model. This means that some details are abstracted away compared to an RTL bus. This has the following consequences:

- The signals on the pin-accurate interface will not be exactly the same as when an RTL bus would be used. The TLM bus model only ensures that the signals are the same as on the RTL bus at times when they are important for the functional correctness of the simulation. Signals will only be valid at times where they matter according to the protocol. For example, on an AHB target interface, the ***HADDR*** signals will only be relevant while ***HSEL*** is high, as a target should not sample ***HADDR*** anyway while ***HSEL*** is low.
- The TLM bus model is cycle accurate, not timing accurate. This means that all changes of signals will happen on the clock edge. An HDL peripheral that has outputs connected to the TLM bus model should also only change these signals on the clock edge. There should be no delays. Inserting delays will result in incorrect simulation results. For example, an HDL AHB target accessed for read should set ***HRDATA*** on the rising clock edge. In the case of Verilog code, an HDL AHB target contains code like:

  ```
  HRDATA &lt;= #2 internal_data;
  ```

  and ***internal_data*** is set on the rising edge, the code must be changed so that ***HRDATA*** is set immediately:

  ```
  HRDATA &lt;= #0 internal_data;
  ```

AHB Protocols

- PV AHB Protocols
- TLM AHB Protocols
- Pin-Accurate AHB Protocols
- AHB Protocol-Common Parameters: Setting Address and Data Width

PV AHB Protocols

At the PV abstraction level, four protocols are available:

- ***AHBInitiator***
- ***AHBTarget***
- ***AHBLiteInitiator***
- ***AHBLiteTarget***

You can select these protocols in Platform Creator when editing the properties of an initiator or a target port.

All these protocols are the same at the PV abstraction level. The four protocols exist to allow a top-down flow to the TLM and the pin-accurate abstraction level.

When you want to connect a peripheral containing a PV protocol to one of the AMBA nodes which only supports AHB protocols, you have to change the PV protocol of that peripheral into one of these PV AHB protocols.

For more information on the PV protocol, see “PV Transport API Syntax” in the ***Platform Creator User Manual***.

TLM AHB Protocols

At the TLM abstraction level, four protocols are available:

- ***AHBInitiator***
- ***AHBTarget***
- ***AHBLiteInitiator***
- ***AHBLiteTarget***

You can select these protocols in Platform Creator when editing the properties of an initiator or a target port.

The following describes:

- TLM AHB Ports
- Transfers Available in the TLM AHBTarget and AHBLiteTarget Protocol
- Transfers Available in the TLM AHBInitiator and AHBLiteInitiator Protocol
- Cross-Referencing Transfers Through the TLM API
- TLM Transfer Timing
- Mapping TLM Transfers and Transfer Attributes to AMBA AHB Signals
- Using the sendDelayed TLM API Call to Send Transfers

TLM AHB Ports

For each of the existing protocols, ports are defined for each combination of master/slave and in/out/inout. These are the classes that need to be instantiated when writing a peripheral that will be used with a bus generated by Platform Creator.

AHBLiteInitiator Ports

- ***AHBLiteInitiator_inmaster_port***
- ***AHBLiteInitiator_inoutmaster_port***
- ***AHBLiteInitiator_outmaster_port***

AHBLiteTarget Ports

- ***AHBLiteTarget_inoutslave_port***
- ***AHBLiteTarget_inslave_port***
- ***AHBLiteTarget_outslave_port***

AHBInitiator Ports

- ***AHBInitiator_inmaster_port***
- ***AHBInitiator_inoutmaster_port***
- ***AHBInitiator_outmaster_port***

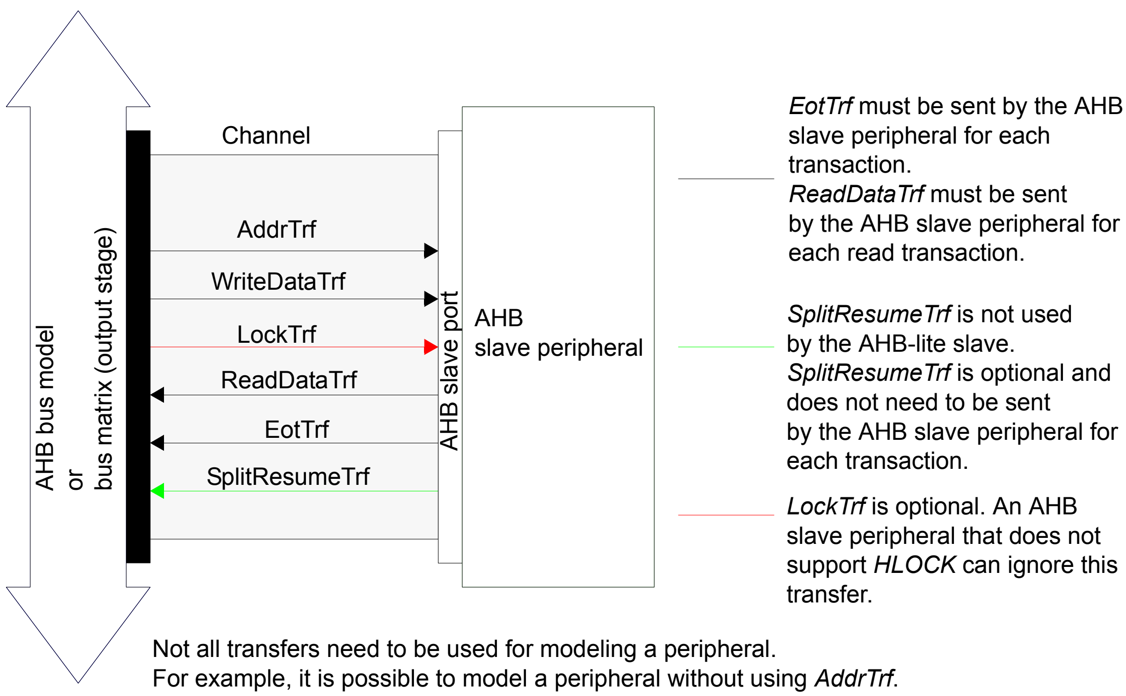
AHBTarget Ports

- ***AHBTarget_inoutslave_port***
- ***AHBTarget_inslave_port***
- ***AHBTarget_outslave_port***

All these port types are defined in ***AMBA.h***, which can be found in the ***IPLOCATION/include/AMBA*** directory: They are all derived from the ***sc_port*** class. Other types are available in this header file, but these should not be used when writing a peripheral. They are intended for use in the generated bus itself.

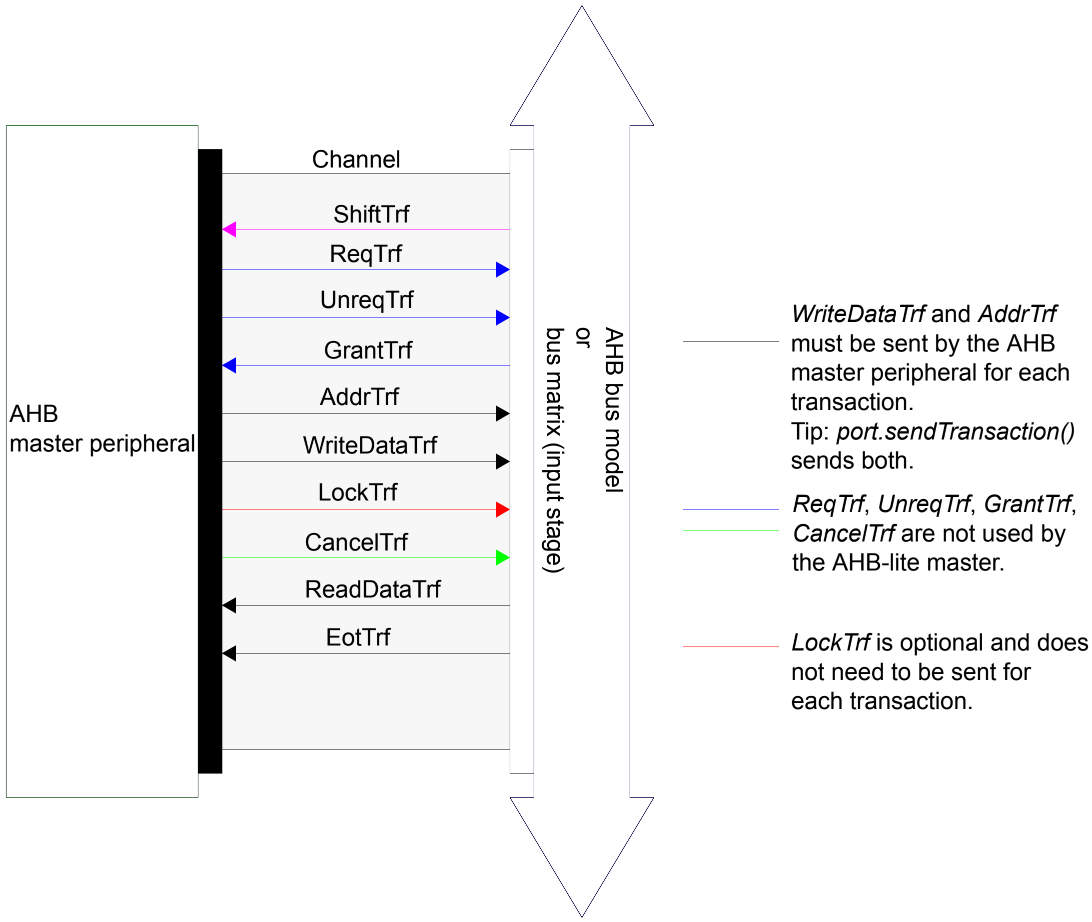
Transfers Available in the TLM AHBTarget and AHBLiteTarget Protocol

The following figure describes all transfers when using the ***AHBTarget*** or the ***AHBLiteTarget*** protocol.



Transfers Available in the TLM AHBInitiator and AHBLiteInitiator Protocol

The following figure describes all transfers when using the ***AHBInitiator*** or ***AHBLiteInitiator*** protocol.



Cross-Referencing Transfers Through the TLM API

When writing TLM peripherals, the TLM API allows you to access attributes of other transfers via the currently available transfer.

For example, when receiving an ***EotTrf***, it is possible to retrieve the address associated with the same transaction for which the ***EotTrf*** was received. This mechanism is referred to as ***cross-referencing transfers***. For more information, see the ***TLM API Manual***.

The syntax in this example would be:

```
address = p.EotTrf->getAddrTrf()->getAddress();
```

From a given transfer, it is not possible to access any other transfer. Only transfers for which a cross-reference exists are accessible. There is for example no cross-reference between ***EotTrf*** and ***WriteDataTrf*** of a target-port protocol, so you cannot write:

```
writeData = p.EotTrf->getWriteDataTrf()->getWriteData();
```

The following describes:

- AHBLiteInitiator Protocol Transfer Cross-References
- AHBLiteTarget Protocol Transfer Cross-References
- AHBinitiator Protocol Transfer Cross-References
- AHBTarget Protocol Transfer Cross-References

AHBLiteInitiator Protocol Transfer Cross-References

The following table shows the ***AHBLiteInitiator*** protocol transfer cross-references.

Can refer to	AddrTrf	WriteDataTrf	ReadDataTrf	EotTrf	LockTrf	ShiftTrf
AddrTrf						
WriteDataTrf	÷				÷	
ReadDataTrf	÷			÷	÷	
EotTrf	÷	÷	÷		÷	
LockTrf						
ShiftTrf						

AHBLiteTarget Protocol Transfer Cross-References

The following table shows the ***AHBLiteTarget*** protocol transfer cross-references.

Can refer to	AddrTrf	WriteDataTrf	ReadDataTrf	EotTrf	LockTrf
AddrTrf					÷
WriteDataTrf	÷				÷
ReadDataTrf	÷				÷
EotTrf	÷				÷
LockTrf					

AHBinitiator Protocol Transfer Cross-References

ShiftTrf, ***ReqTrf***, ***UnreqTrf***, ***GrantTrf***, and ***CancelTrf*** cannot refer to other transfers or cannot be referred to.

For all other transfers, the cross-references are the same as for the ***AHBLiteInitiator*** protocol.

AHBTarget Protocol Transfer Cross-References

SplitResumeTrf cannot refer to other transfers or cannot be referred to.

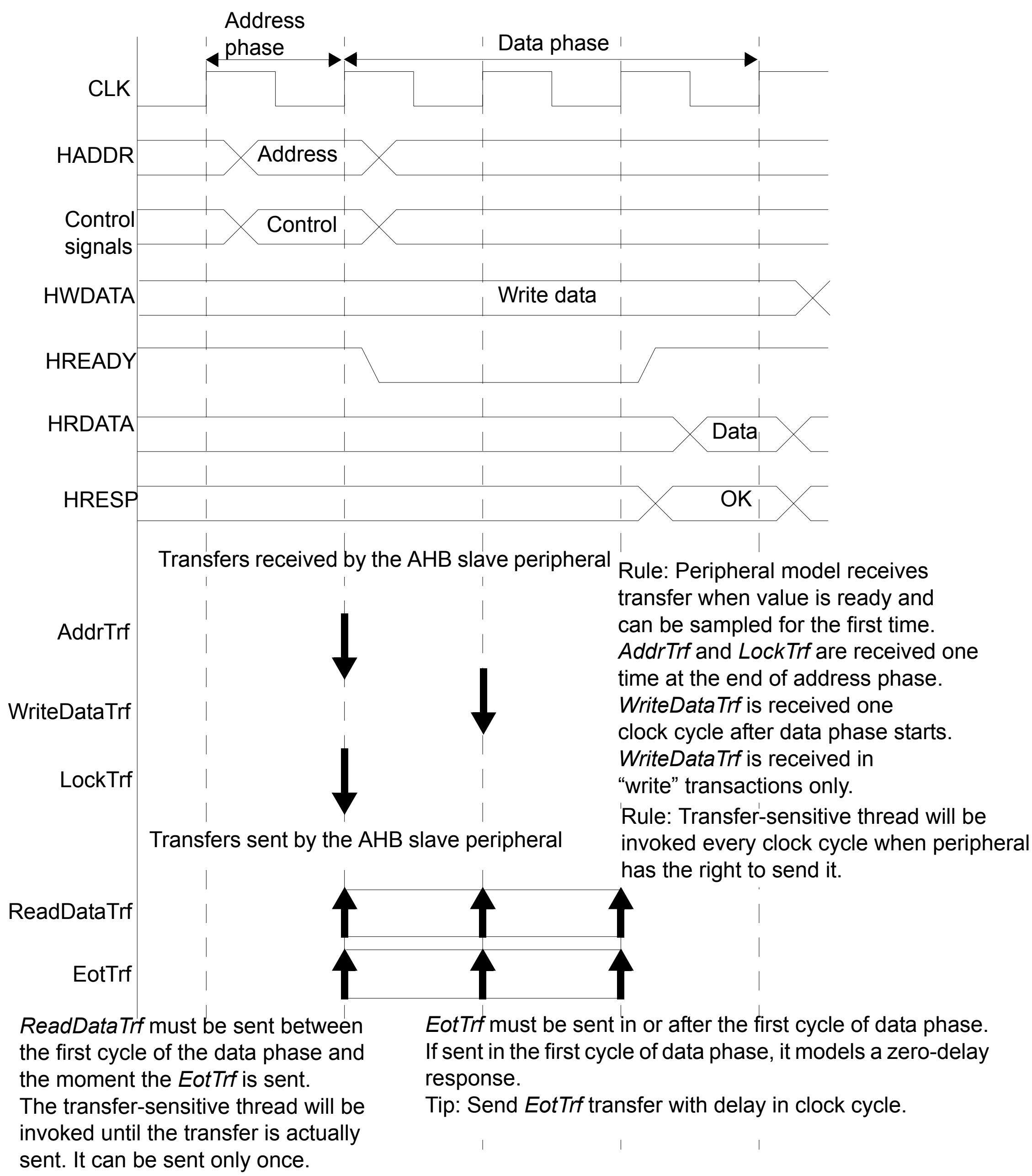
For all other transfers, the cross-references are the same as for the ***AHBLiteTarget*** protocol.

TLM Transfer Timing

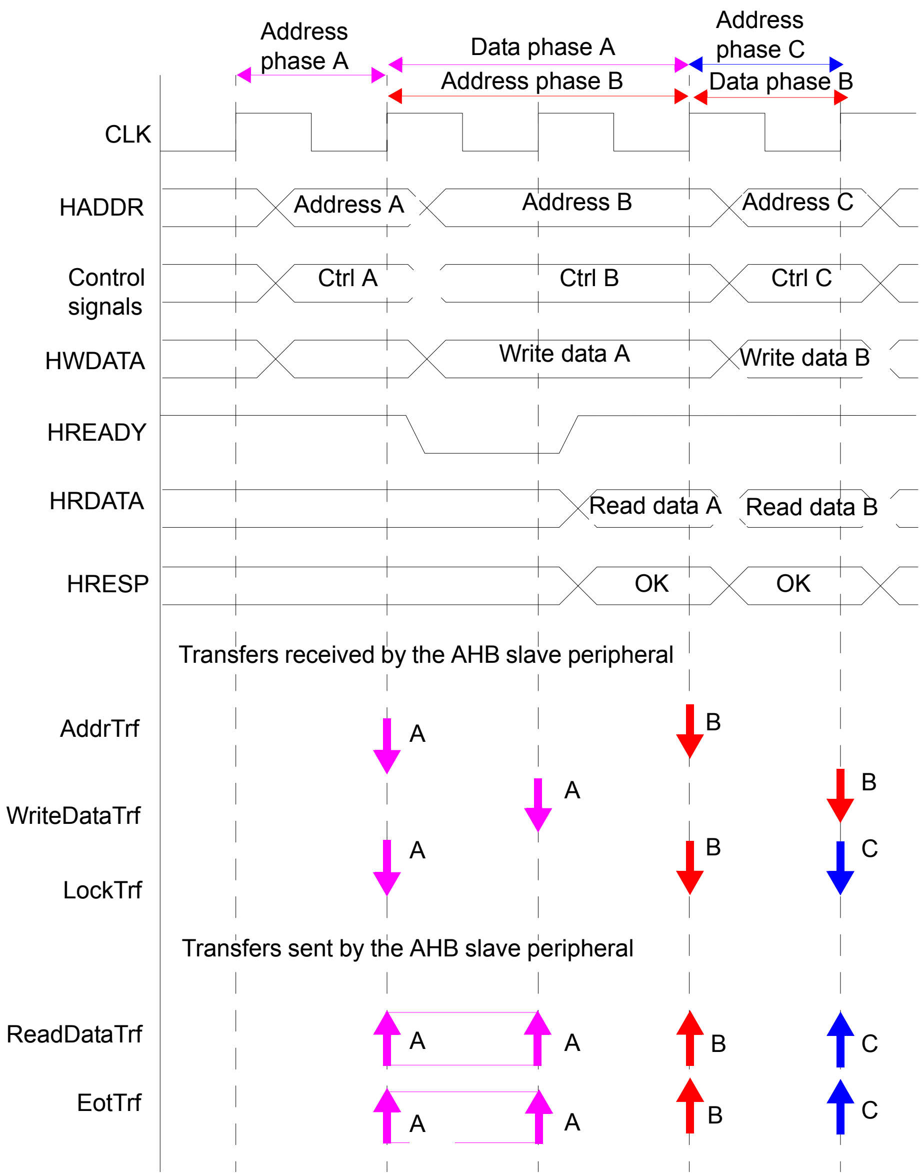
The following describes the relation, in time, between the sending and receiving of TLM transfers, and the signals described in the ***ARM AMBA AHB Specification***.

Timing for the TLM AHB and AHBLiteTarget Protocols

The following figure shows the TLM timing of an ***AHBTarget*** or ***AHBLiteTarget*** peripheral for a basic transfer with wait states.

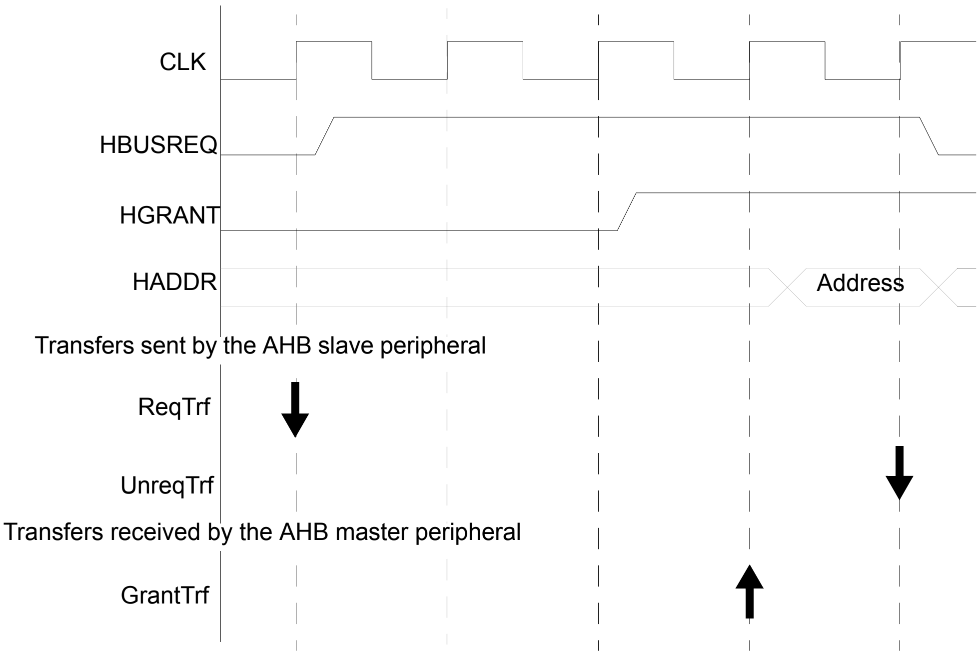


The following figure shows the AHB slave peripheral TLM timing consecutive access. Note that the threads sensitive to ***ReadDataTrf*** and ***EotTrf*** are triggered until these transfers are actually sent.

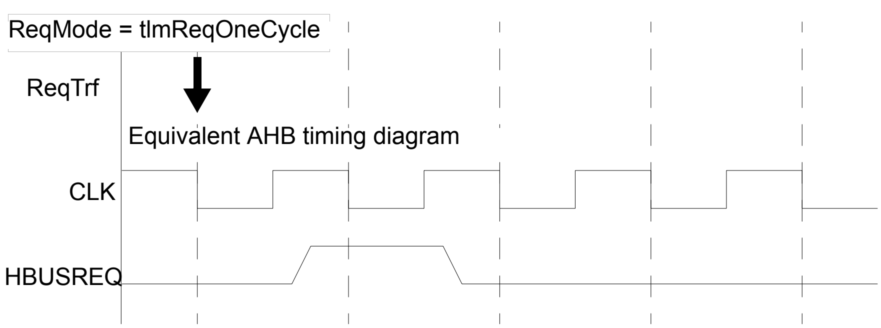


Timing for the TLM AHBInitiator Arbitration and the Different Arbitration Modes

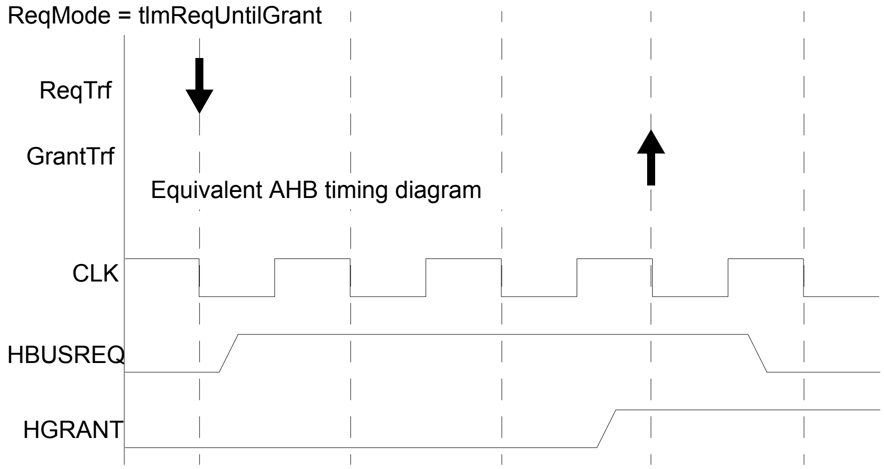
The following figure shows the AHB master arbitration TLM timing (***ReqUntilUnreq*** mode). Note that ***ReqTRf*** has three different modes (see also "ReqTrf" on page 28).



The following figure shows the timing diagram for the ***ReqOneCycle*** request mode. In this mode, a request is issued for one cycle each time the request transfer is sent. The ***Unreq*** transfer is not used in this mode.

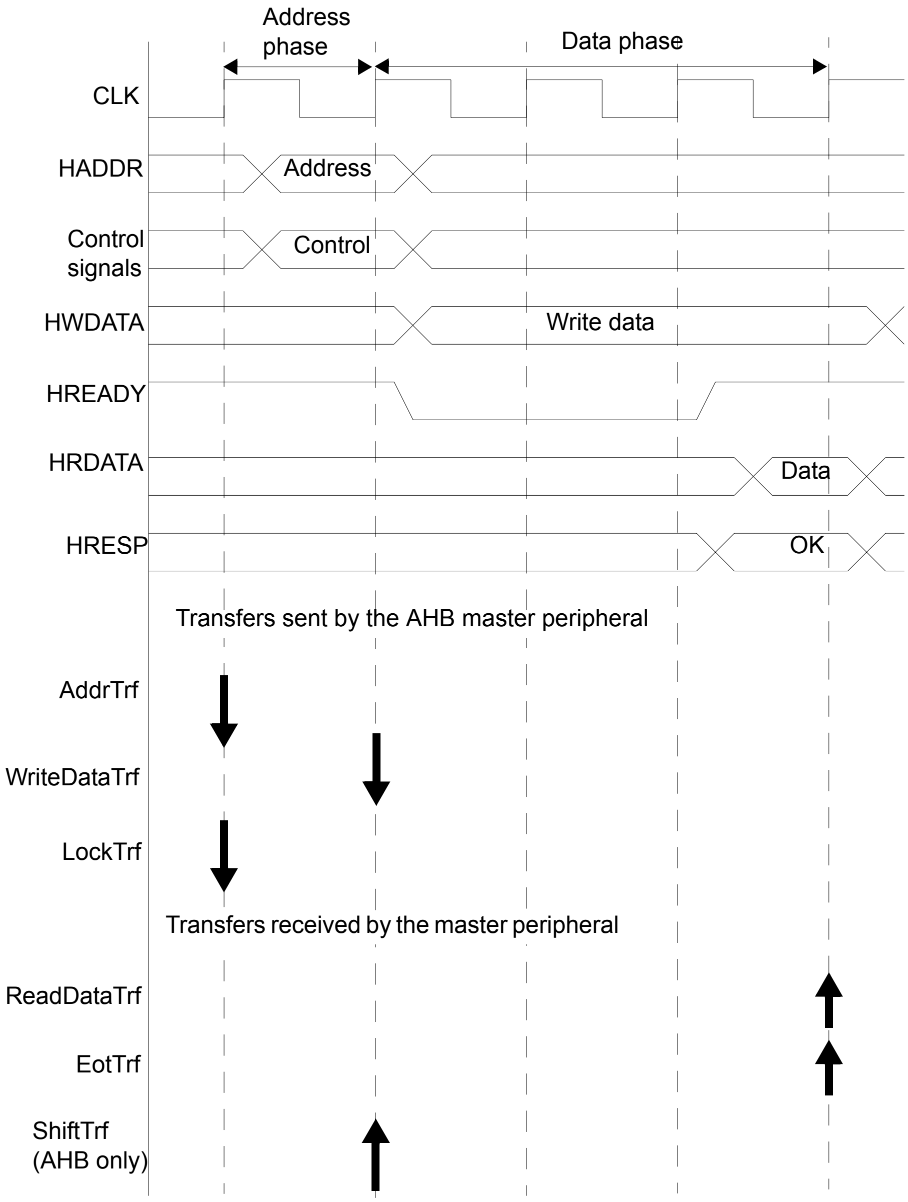


The following figure shows the timing diagram for the ***ReqUntilGrant*** request mode. In this mode, a request issued is after sending the request transfer. The request remains valid until the initiator port is granted. The ***Unreq*** transfer is not used in this mode.



Timing for the TLM AHBInitiator Protocol

The following figure illustrates AHB or AHB-lite basic transfer with wait states seen from the initiator port.

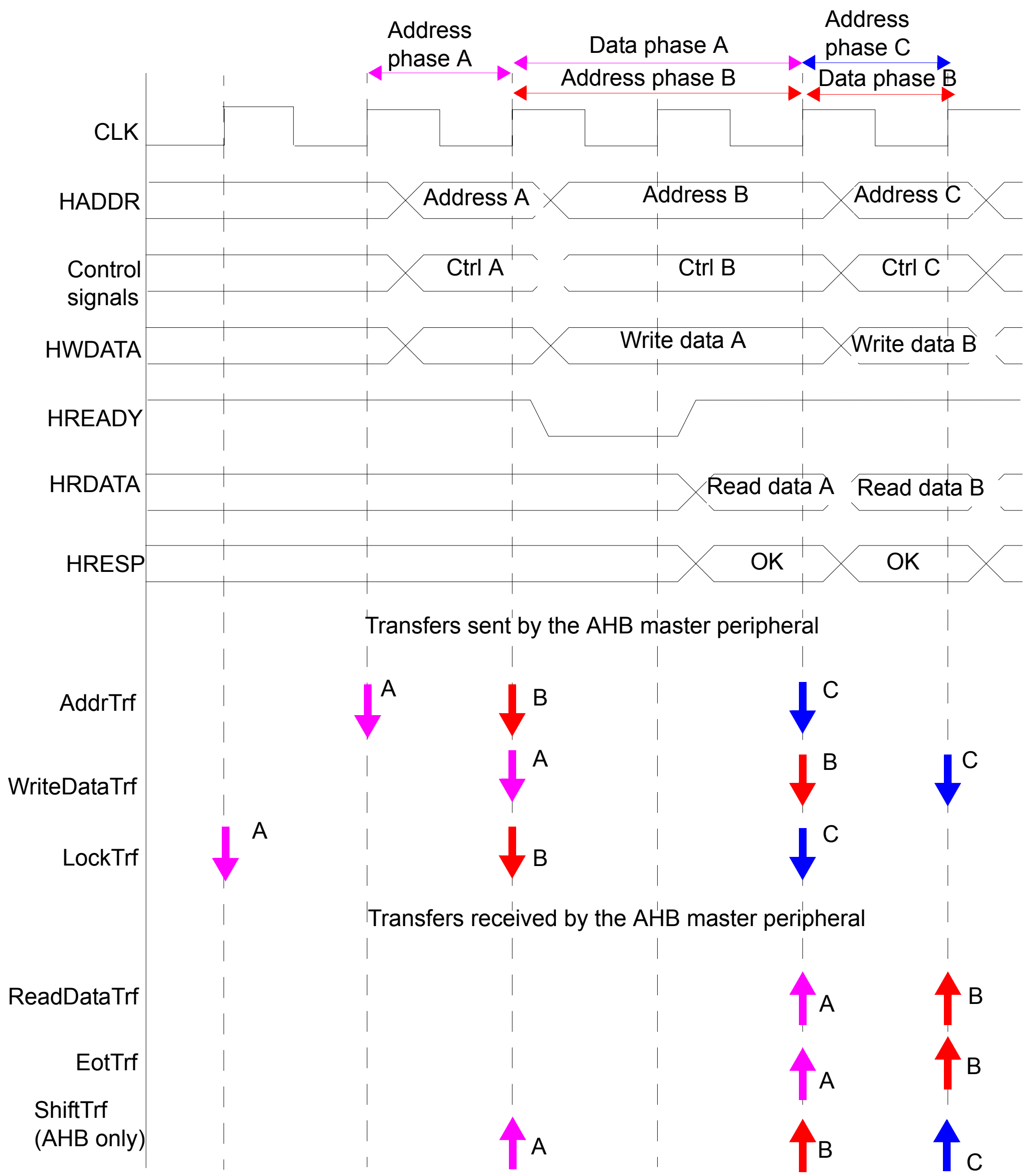


The ***AHBInitiator*** protocol must send an ***AddrTrf*** at the beginning of the address phase, and it must send ***WriteDataTrf*** at the beginning of the data phase of a write access.
The ***LockTrf*** is optional; it does not need to be sent on every transfer. It should only be used when ***HLOCK*** is set.

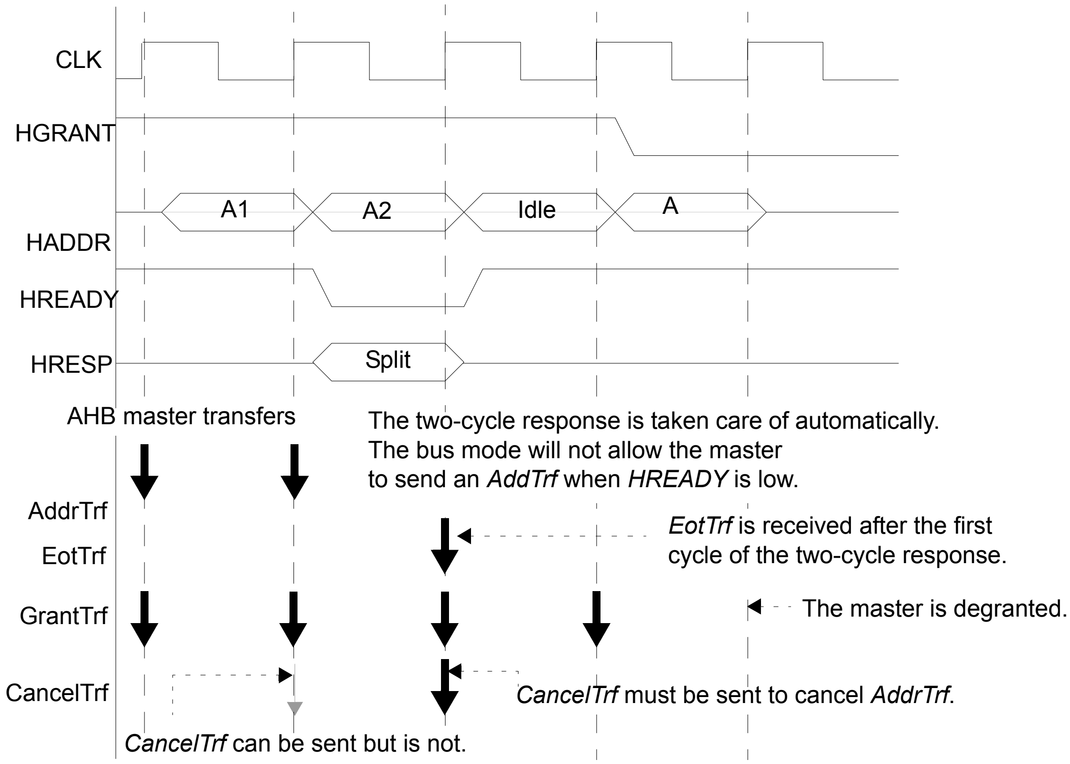
The AHB initiator receives a ***ReadDataTrf*** when read data is ready to be sampled. In the case of successful transaction completion, the AHB master receives an ***EotTrf*** with status set to ***ok***, one cycle after the target port has issued the ***EotTrf***.

ShiftTrf can be received by the initiator every time there is a transition from address to data phase. Only the initiator that has sent the transaction which moves from address phase to data phase will receive this transfer. You normally do not need this transfer when modeling your peripheral, unless it is absolutely necessary that you know when the pipeline is shifted.

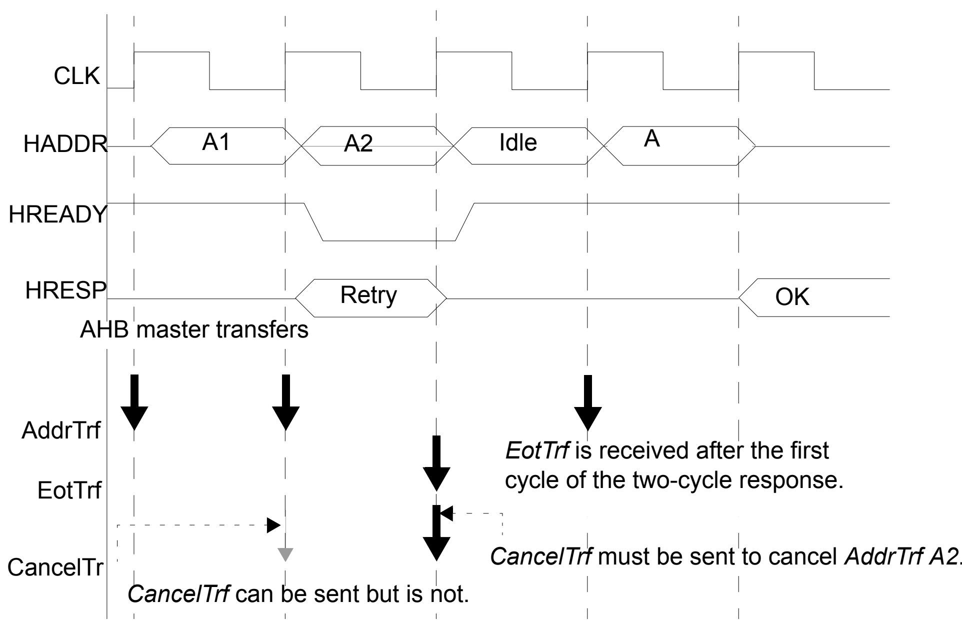
The following figure illustrates consecutive AHB or AHB-lite transfers seen from the initiator port.



The following figure illustrates AHB master split response.



The following figure illustrates AHB master retry response.



The two-cycle response is taken care of automatically. The bus model will not allow the master to send an ***AddTrf*** when ***HREADY*** is low.

Mapping TLM Transfers and Transfer Attributes to AMBA AHB Signals

The following describes the relation between TLM transfers and the AMBA AHB signals, and more specific, the relation between the values of TLM transfer attributes and those signals.

There is not always a one-to-one mapping between a transfer or an attribute and a signal. Sometimes, a single transfer contains information about multiple signals and sometimes, multiple transfer attributes give information about a single signal.

- ***ReqTrf***
- ***UnreqTrf***
- ***GrantTrf***
- ***AddrTrf***
- ***WriteDataTrf***
- ***ReadDataTrf***
- ***LockTrf***
- ***EotTrf***
- ***CancelTrf***
- ***SplitResumeTrf***
- ***ShiftTrf***

ReqTrf

The ***ReqTrf*** transfer can only be sent by an initiator port in the system

This transfer is not available in the AHB-lite protocol. The ***ReqTrf*** transfer corresponds with setting the ***HBUSREQ*** signal at the pin-accurate abstraction level and is only needed for arbitration.

It has the following attribute:

- ***ReqMode*** is an enumerated type. Possible values are:
  - ***tlmReqOneCycle*** specifies that the request is only valid for one cycle
  - ***tlmReqUntilGrant*** specifies that the request is valid until the initiator port is granted
  - ***tlmReqUntilUnreq*** specifies that the request is valid until an ***UnreqTrf*** transfer is sent

UnreqTrf

The ***UnreqTrf*** transfer can only be sent by an initiator port of the system and is used to clear the request signal if a request was sent with an attribute different from ***tlmReqOneCycle***.

This transfer is not available in the AHB-lite protocol. The ***UnreqTrf*** transfer corresponds with clearing the ***HBUSREQ*** signal at the pin-accurate abstraction level and is only needed for arbitration.

This transfer has no attributes.

GrantTrf

The ***GrantTrf*** transfer can only be received by an initiator port of the system and indicates that an initiator port is granted.

This transfer does not occur in the AHB-lite protocol. The ***GrantTrf*** transfer corresponds with the ***HGRANT*** signal at the pin-accurate abstraction level and is only needed for arbitration.

This transfer has no attributes.

AddrTrf

The ***AddrTrf*** transfer can be sent by an initiator port in the system and received by a target port in the system.

It has the following attributes:

- ***Address*** is an unsigned integer. Its width is set by the ***address_width*** template. Its value corresponds with the ***HADDR*** signal at the pin-accurate abstraction level.
- ***Type*** is an enumerated type. It indicates the type of access. Possible values are
  - ***tlmIdle*** specifies that the transfer is an idle or a busy transfer. It corresponds with the ***HTRANS*** signal being equal to ***00*** or ***01*** at the pin-accurate abstraction level.
  - ***tlmReadAtAddress*** specifies that the transfer is a read transfer. It corresponds with the ***HWRITE*** signal being equal to ***0*** and the ***HTRANS*** signal being equal to ***NONSEQ*** (***10***) or ***SEQ*** (***11***) at the pin-accurate abstraction level.
  - ***tlmWriteAtAddress*** specifies that the transfer is a write transfer. It corresponds with the ***HWRITE*** signal being equal to ***1*** and the ***HTRANS*** signal being equal to ***NONSEQ*** (***10***) or ***SEQ*** (***11***) at the pin-accurate abstraction level.
- ***AccessSize*** is the size of the accessing bits. Possible values are: ***64***, ***32***, ***16***, ***8***. Its meaning corresponds with the ***HSIZE*** signal at the pin-accurate abstraction level. Note that the value is not the same as at the pin-accurate abstraction level, where the value equals the number of bytes.
- ***Kind*** is an enumerated type. This attribute is mostly unused, and it corresponds with bit 0 of the ***HPROT*** signal at the pin-accurate abstraction level. Possible values are:
  - ***tlmOpcode*** specifies that the transfer is an opcode fetch.
  - ***tlmData*** specifies that the transfer is a data access.
- ***Group*** is an enumerated type and gives information about a burst access. Possible values are:
  - ***tlmSingle*** specifies that the current transfer is not part of a burst. This value corresponds with the ***HTRANS*** signal being equal to ***NONSEQ*** and the ***HBURST*** signal being equal to ***SINGLE*** at the pin-accurate abstraction level.
  - ***tlmBurstStart*** specifies the start of a burst transfer. This is the first transfer of a burst. This attribute corresponds with the ***HTRANS*** signal being equal to ***NONSEQ*** and the ***HBURST*** signal not being equal to ***SINGLE*** at the pin-accurate abstraction level.
  - ***tlmBurstCont*** specifies a sequential transaction. This attribute corresponds with the ***HTRANS*** signal being equal to ***SEQ*** and the ***HBURST*** signal not being equal to ***SINGLE*** at the pin-accurate abstraction level.
  - ***tlmBurstIdle*** specifies a busy cycle. This attribute corresponds with the ***HTRANS*** signal being equal to ***BUSY*** at the pin-accurate abstraction level. During a busy cycle, the type of ***AddrTrf*** should be ***tlmIdle***.

- ***BurstWrap*** tells whether a burst is incremental or wrapping. It corresponds with the ***HBURST*** signal at the pin-accurate abstraction level. Possible values are:
  - ***tlmIncremental*** specifies that the transfer is part of an incremental burst.
  - ***tlmWrapBurstSize*** specifies that the transfer is part of a wrapping burst.
- ***BurstLength*** specifies the length of a burst. This attribute corresponds with the ***HBURST*** signal. Possible values are: ***4***, ***8***, ***16***. If the burst is of unspecified length, ***0x3ff*** should be specified, which corresponds with the ***HBURST*** signal being equal to ***INCR*** at the pin-accurate abstraction level.
- ***Cacheable*** indicates whether the transfer is cacheable or not. Possible values are: ***0***, ***1***. The attribute corresponds with bit 3 of the ***HPROT*** signal at the pin-accurate abstraction level.
- ***Bufferable*** indicates whether the transfer can be buffered or not. Possible values are: ***0***, ***1***. The attribute corresponds with bit 2 of the ***HPROT*** signal at the pin-accurate abstraction level.
- ***ProtectionType*** indicates whether the transfer is a privileged mode access or a user mode access. It corresponds with bit 1 of the ***HPROT*** signal at the pin-accurate abstraction level. Possible values are:
  - ***tlmUser*** specifies that the transfer is a user-mode access.
  - ***tlmPrivileged*** specifies that the transfer is a privileged-mode access.
- ***MasterId*** can only be accessed by the target ports. It gives the ID of the master in the arbiter and can be used for sending a ***SplitResumeTrf***. The attribute corresponds with the ***HMASTER*** signal at the pin-accurate abstraction level. This attribute has no meaning in the AHB-lite protocol.

WriteDataTrf

The ***WriteDataTrf*** transfer can be sent by an initiator port and received by a target port. It has the following attribute:

- ***WriteData*** contains the write data. The type is ***unsigned int***.

This transfer corresponds with the ***HWDATA*** signal at the pin-accurate abstraction level.

ReadDataTrf

The ***ReadDataTrf*** transfer can be sent by a target port and received by an initiator port. It has the following attribute:

- ***ReadData*** contains the read data. The type is ***unsigned int***.

This transfer corresponds with the ***HRDATA*** signal at the pin-accurate abstraction level.

LockTrf

The ***LockTrf*** transfer can be sent by an initiator port and received by a target port. It has the following attribute:

- ***Lock*** indicates whether or not the transfer is locked. Possible values are ***1*** (meaning locked), ***0*** (means not locked).

The transfer corresponds with the ***HLOCK*** signal at the pin-accurate abstraction level for ***AHBInitiator*** protocols. For ***AHBLiteInitiator*** protocols, ***AHBTarget***, and ***AHBLiteTarget*** protocols, it corresponds with the ***HMASTLOCK*** signal.

EotTrf

The ***EotTrf*** transfer can be sent by a target port and received by an initiator port. It indicates the end of a transfer.

The transfer corresponds with the ***HREADY*** and ***HRESP*** signal at the pin-accurate abstraction level. It has the following attribute:

- ***Status*** is an enumerated type that indicates the kind of response a slave is doing. Possible values are:
  - ***tlmOk*** specifies an okay response.
  - ***tlmError*** specifies an error response.
  - ***tlmRetry*** specifies a retry response. It does not occur in the AHB-lite protocol.
  - ***tlmSplit*** specifies a split response. It does not occur in the AHB-lite protocol.

CancelTrf

The ***CancelTrf*** transfer can be sent by an initiator port, and is used to cancel a transaction or ***AddrTrf*** that was sent a cycle earlier. This can be necessary when the initiator port needs to redo a previous transfer after receiving a split or retry response from the target port.

The transfer is only present in the ***AHBInitiator*** protocol and cannot be received by the target port. It is not used in ***AHBLiteInitiator*** protocol.

This transfer has no attributes.

There are no signals at the pin-accurate abstraction level that correspond with this transfer.

It can best be compared with resetting the ***HTRANS*** signal to ***0*** after it has already been set to one, while the ***HREADY*** signal has not been high yet during the time the ***HTRANS*** signal was not equal to ***0***.

SplitResumeTrf

The ***SplitResumeTrf*** transfer is only found in the ***AHBTarget*** protocol, hence, it can only be sent by the target port. The initiator port cannot access this transfer.

This transfer indicates which master can be granted again after the target port has issued a split response for that master.

The transfer corresponds with the ***HSPLIT*** signal at the pin-accurate abstraction level.

It has the following attribute:

- ***splitResume*** behaves in the same way as the ***HSPLIT*** signal. It is 16 bits wide. The slave must set the bit corresponding to the master ID of the master for which a split response has previously been issued. See also "AddrTrf" on page 29.

This transfer does not occur in the ***AHBLiteTarget*** protocol.

ShiftTrf

The ***ShiftTrf*** transfer can only be received by the initiator.

When the initiator can receive ***ShiftTrf,*** this corresponds with sampling ***HREADY*** high while the initiator has a transaction in its address phase on the bus.

This transfer has no attributes.

Using the sendDelayed TLM API Call to Send Transfers

The ***sendDelaye***d TLM API call described in the ***TLM API Manual*** is only applicable to ***ReadDataTrf*** and ***EotTrf*** of the ***AHBLiteTarget*** and ***AHBTarget*** protocol.

So the available calls are:

```
port.sendDelayedEotTrf(delay)
port.sendDelayedReadDataTrf(delay)
```

These calls are typically used when a target peripheral wishes to insert wait states.

Pin-Accurate AHB Protocols

The following pin-accurate protocols can be selected for the initiator and target ports:

- ***AHBInitiator***. The ***AHBInitiator*** terminals are described in "Pin-Accurate AHBInitiator Terminals" on page 33.
- ***AHBTarget***. The ***AHBTarget*** terminals are described in "Pin-Accurate AHBTarget Terminals" on page 34.
- ***AHBLiteInitiator***. The ***AHBLiteInitiator*** terminals are described in "Pin-Accurate AHBLiteInitiator Terminals" on page 35.
- ***AHBLiteTarget***. The ***AHBLiteTarget*** terminals are described in "Pin-Accurate AHBLiteTarget Terminals" on page 36.

NOTE All signals fed to the bus model should change immediately on the rising clock edge. Delays on signals are not supported. This is the case for peripherals modeled at the pin-accurate abstraction level, and for RTL peripherals connected to the bus model using RTL co-simulation.

Pin-Accurate AHBInitiator Terminals

Terminal	Width	Direction at Initiator
HADDR	1 -> 32	Output
HBURST[1]	3	Output
HBUSREQ	1	Output
HGRANT	1	Input
HLOCK	1	Output
HPROT[1]	4	Output
HRDATA	8-16-32-64	Input
HREADY	1	Input
HRESP	2	Input
HSIZE[1]	3	Output
HTRANS	2	Output
HWDATA	8-16-32-64	Output
HWRITE	1	Output

1. These are optional pins.

Pin-Accurate AHBTarget Terminals

Terminal	Width	Direction at Target
HADDR[1]	0 -> 32	Input
HBURST[1]	3	Input
HMASTER	4	Input
HMASTLOCK[1]	1	Input
HPROT[1]	4	Input
HRDATA	8-16-32-64	Output
HREADYin	1	Input
HREADYout	1	Output
HRESP	2	Output
HSIZE[1]	3	Input
HTRANS	2	Input
HWDATA	8-16-32-64	Input
HWRITE	1	Input
HSPLIT	16	Output

1. These are optional pins.

Pin-Accurate AHBLiteInitiator Terminals

Terminal	Width	Direction at Initiator
HADDR	1 -> 32	Output
HBURST[1]	3	Output
HMASTLOCK[1]	1	Output
HPROT[1]	4	Output
HRDATA	8-16-32-64	Input
HREADY	1	Input
HRESP	2	Input
HSIZE[1]	3	Output
HTRANS	2	Output
HWDATA	8-16-32-64	Output
HWRITE	1	Output

1. These are optional pins.

Pin-Accurate AHBLiteTarget Terminals

Terminal	Width	Direction at Target
HADDR[1]	0 -> 32	Input
HBURST[1]	3	Input
HMASTLOCK[1]	1	Input
HPROT[1]	4	Input
HRDATA	8-16-32-64	Output
HREADYin	1	Input
HREADYout	1	Output
HRESP	2	Output
HSEL	1	Input
HSIZE[1]	3	Input
HTRANS	2	Input
HWDATA	8-16-32-64	Input
HWRITE	1	Input

1. These are optional pins.

AHB Protocol-Common Parameters: Setting Address and Data Width

The ***AHBInitiator***, ***AHBTarget***, ***AHBLiteInitiator***, ***AHBLiteTarget*** protocols all have the same set of parameters.

The available protocol-common parameters are:

- ***address_width*** specifies the number of address lines. This is an integer value ranging from ***0*** to ***32***. For ***AHBInitiator*** and ***AHBLiteInitiator***, the address width must be bigger than ***0***.
- ***data_width*** specifies the data width. This is an integer value. Possible values are ***8***, ***16***, ***32***, ***64***.

The values of the protocol-common parameters are determined in the source code of your peripherals. The values specified in the source code will be the values you see in Platform Creator.

There are two ways to specify the values of the protocol-common parameters of a certain port.

- One option is to pass fixed values to the port templates where the port is instantiated in the source code of your peripheral.

  For example:

  ```
  AHBLiteInitiator_inoutmaster_port<32,16> port;
  ```

  This way, the protocol-common parameters will be fixed to ***32*** for ***`address_width`*** and ***16*** for ***`data_width`***.

- Another option is to use block templates in the port instantiation.

  For example:

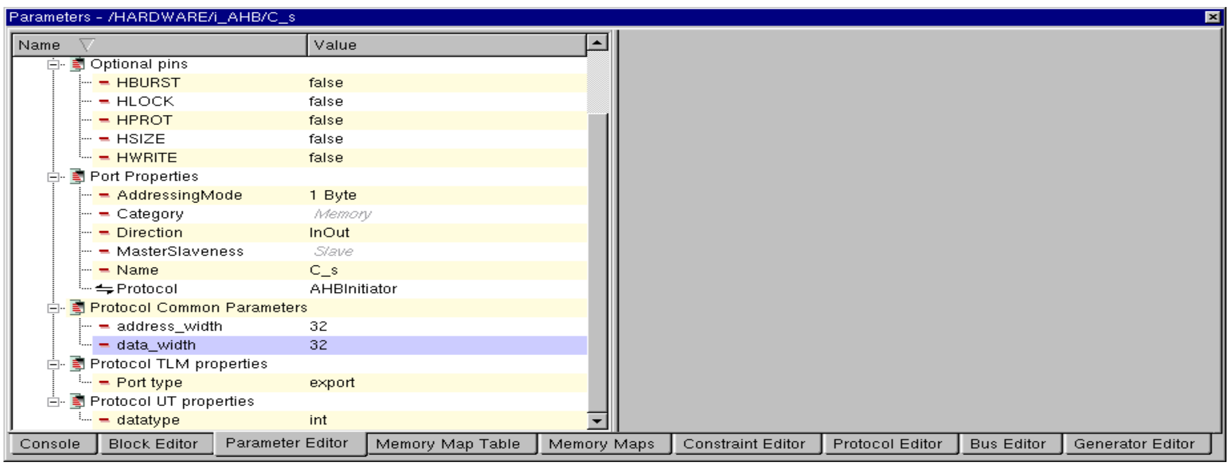
```
template <int address_width = 32, int data_width = 64>
SC_MODULE(themodule)
{
  sc_in_clk clk;
  sc_in<bool> reset;
AHBLiteInitiator_inoutmaster_port<address_width, data_width> p;

}
```

  This way, you can still influence the ***address_width*** and ***data_width*** of your peripherals port by editing the block templates in Platform Creator.

For more information, see “Templates” and “Correcting the Port and Port-Protocol Parameters” in the ***Platform Creator User Manual***.

The following figure shows the protocol-common parameters in the Platform Creator GUI.



NOTE Make sure you have selected the correct protocol for the initiator or target port before you attempt to change the protocol-common parameters.

APB Protocols

- PV APB Protocols
- TLM APB Protocols
- Pin-Accurate APB Protocols
- APB Protocol-Common Parameters: Setting Address and Data Width

PV APB Protocols

At the PV abstraction level, two protocols are available:

- ***APBInitiator***
- ***APBTarget***

You can select these protocols in Platform Creator when editing the properties of an initiator or a target port.

All these protocols are the same at the PV abstraction level. The two protocols exist to allow a top-down flow to the TLM and the pin-accurate abstraction level.

When you want to connect a peripheral containing a PV protocol to one of the AMBA nodes which only supports APB protocols, you have to change the PV protocol of that peripheral into one of these PV APB protocols.

For more information on the PV protocol, see “PV Transport API Syntax” in the ***Platform Creator User Manual***.

TLM APB Protocols

At the TLM abstraction level, two protocols are available:

- ***APBInitiator***
- ***APBTarget***

You can select these protocols in Platform Creator when editing the properties of an initiator or target port.

The following describes:

- TLM APB Ports
- Transfers Available in the TLM APBTarget Protocol
- Cross-Referencing Transfers Through the TLM API
- TLM Transfer Timing
- Mapping TLM Transfers and Transfer Attributes to AMBA APB Signals

TLM APB Ports

For each of the existing protocols, ports are defined for each combination of master/slave and in/out/inout. These are the classes that need to be instantiated when writing a peripheral that will be used with a bus generated by Platform Creator.

APBInitiator Ports

- ***APBInitiator_inoutmaster_port***
- ***APBInitiator_inmaster_port***
- ***APBInitiator_outmaster_port***

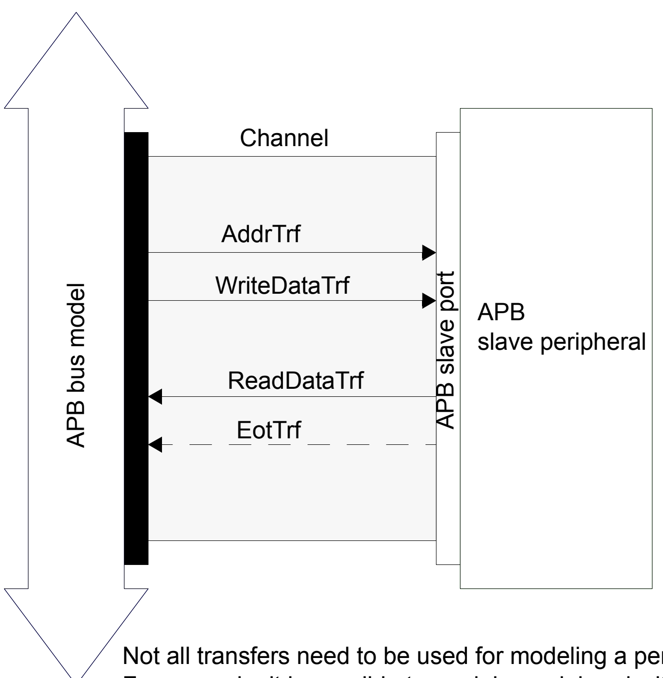
APBTarget Ports

- ***APBTarget_inoutslave_port***
- ***APBTarget_inslave_port***
- ***APBTarget_outslave_port***

All these port types are defined in ***AMBA.h***, which can be found in the ***IPLOCATION/include/AMBA*** directory: They are all derived from the ***sc_port*** class. Other types are available in this header file, but these should not be used when writing a peripheral. They are intended for use in the generated bus itself.

Transfers Available in the TLM APBTarget Protocol

The following figure describes all transfers when using the ***AHBTarget*** protocol.



Not all transfers need to be used for modeling a peripheral.
For example, it is possible to model a peripheral without using *AddrTrf*.

Note that also ***EotTrf*** is available. This is not needed to model an APB peripheral.

The reason why ***EotTrf*** is present in the APB protocol is that it becomes quite easy to port simple TLM peripherals from or to the AHB or AHB-lite protocol. Sending ***EotTrf***, however, has no influence on the simulation when the peripheral uses the ***APBTarget*** protocol, and is connected to an APB bus.

Cross-Referencing Transfers Through the TLM API

When writing TLM peripherals, the TLM API allows you to access attributes of other transfers via the currently available transfer.

For example, when receiving an ***WriteDataTrf***, it is possible to retrieve the address associated with the same transaction for which the ***WriteDataTrf*** was received. This mechanism is referred to as ***cross-referencing transfers***. For more information, see the ***TLM API Manual***.

The syntax in this example would be:

```
address = p.WriteDataTrf->getAddrTrf()->getAddress();
```

From a given transfer, it is not possible to access any other transfer. Only transfers for which a cross-reference exists are accessible.

APBTarget Protocol Transfer Cross-References

The following table shows the ***APBTarget*** protocol transfer cross-references.

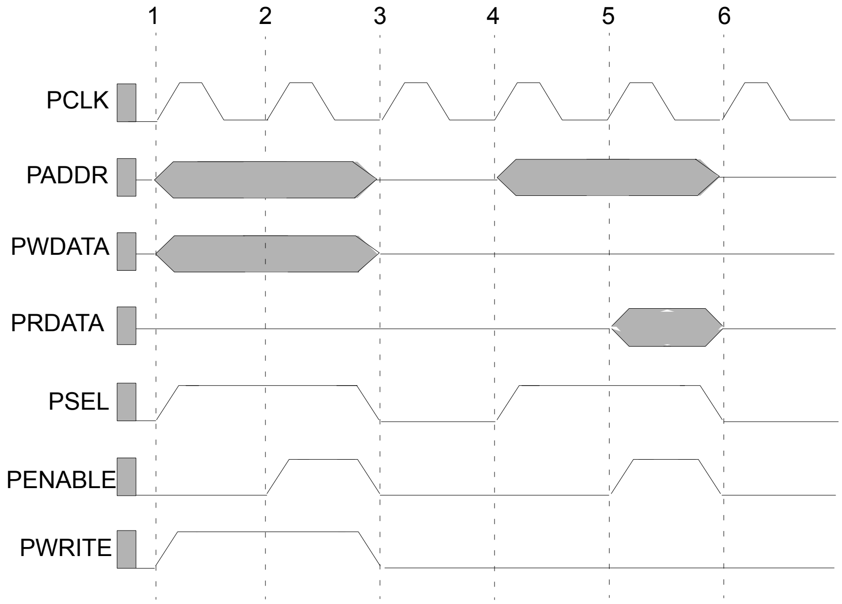
Can refer to	AddrTrf	WriteDataTrf	ReadDataTrf	EotTrf
AddrTrf		÷		
WriteDataTrf	÷			
ReadDataTrf	÷			
EotTrf	÷	÷		

TLM Transfer Timing

The following describes the relation, in time, between the sending and receiving of TLM transfers, and the signals described in the ***ARM AMBA APB Specification***.

Timing for the TLM APBTarget Protocols

The following figure shows the TLM timing of an ***APBTarget*** peripheral.



- The master can send an ***AddrTrf***, and does so with the type attribute set to ***WriteAtAddress.*** Because of this, the master can also immediately send a ***WriteDataTrf.***
- The slave receives the ***AddrTrf*** and the ***WriteDataTrf.***
- The master can send an ***AddrTrf***, but does not.
- The master can send an ***AddrTrf***, and does so with the type attribute set to ***ReadAtAddress.***
- The slave receives the ***AddrTrf*** and can send a ***ReadDataTrf.*** The master cannot send an ***AddrTrf.***
- The master can receive the ***ReadDataTrf.*** The master can send an ***AddrTrf***, but does not.

NOTE For compatibility reasons, the ***EotTrf*** is also present. The slave can send it at the moments ***2*** and ***5***. The master can receive it one clock later.

Mapping TLM Transfers and Transfer Attributes to AMBA APB Signals

The following describes the relation between TLM transfers and the AMBA APB signals, and more specific, the relation between the values of TLM transfer attributes and those signals.

AddTrf

The ***AddTrf*** transfer is received by the APB target peripheral.

This transfer can be sent by an initiator port in the system and received by a target port in the system.

It is received by the peripheral at the end of the setup phase of the transaction.

This transfer has the following attributes:

- ***Address*** is an unsigned integer. Its width is set by the ***address_width*** template. Its value corresponds with the ***HADDR*** signal at the pin-accurate abstraction level.
- ***Type*** is an enumerated type. It indicates the type of access. Possible values are:
  - ***tlmReadAtAddress*** specifies that the transfer is a read transfer. It corresponds with ***PWRITE*** being equal to ***0*** at the pin-accurate abstraction level.
  - ***tlmWriteAtAddress*** specifies that the transfer is a write transfer. It corresponds with ***PWRITE*** being equal to ***1*** at the pin-accurate abstraction level.

WriteDataTrf

The target receives this transfer.

It has the following attribute:

- ***WriteData*** contains the write data. The type is ***unsigned int***.

This transfer corresponds with the ***PWDATA*** signal at the pin-accurate abstraction level.

ReadDataTrf

The target sends this transfer.

It has the following attribute:

- ***ReadData*** contains the read data. The type is ***unsigned int***.

This transfer corresponds with the ***PRDATA*** signal at the pin-accurate abstraction level.

EotTrf

This transfer can be sent by the target. It indicates the end of a transfer.

It has no attribute, and is only provided to increase portability between buses.

It has no corresponding pin-accurate signal.

Pin-Accurate APB Protocols

The following pin-accurate protocol can be selected for the target ports:

- ***APBTarget.*** The ***APBTarget*** terminals are described in "Pin-Accurate APBTarget Terminals" on page 43..

NOTE All signals fed to the bus model should change immediately on the rising clock edge. Delays on signals are not supported. This is the case for peripherals modeled at the pin-accurate abstraction level, and for RTL peripherals connected to the bus model using RTL co-simulation.

Pin-Accurate APBTarget Terminals

Terminal	Width	Direction at Initiator
PADDR[1]	0 -> 32	Input
PWRITE	1	Input
PWDATA	8-16-32	Input
PRDATA	8-16-32	Output
PSEL	1	Input
PENABLE	1	Input

1. This is an optional pin.

APB Protocol-Common Parameters: Setting Address and Data Width

The ***APBTarget*** protocol has the following parameters.

- ***address_width*** specifies the number of address lines. This is an integer value ranging from ***0*** to ***32***.
- ***data_width*** specifies the data width. This is an integer value. Possible values are ***8***, ***16***, ***32***.

The values of the protocol-common parameters are determined in the source code of your peripherals. The values specified in the source code will be the values you see in Platform Creator.

There are two ways to specify the values of the protocol-common parameters of a certain port.

- One option is to pass fixed values to the port templates where the port is instantiated in the source code of your peripheral.

  For example:

  ```
  APBTarget_inoutslave_port<32,16> port;
  ```

  This way, the protocol-common parameters will be fixed to ***32*** for ***`address_width`*** and ***16*** for ***`data_width`***.

- Another option is to use block templates in the port instantiation.

  For example:

  ```
  template <int address_width = 32, int data_width =16>
  SC_MODULE(themodule)
  {
    sc_in_clk clk;
    sc_in<bool> reset;

    APBTarget_inoutslave_port<address_width, data_width> p;

  }
  ```

  This way, you can still influence the ***`address_width`*** and ***`data_width`*** of your peripherals port by editing the block templates in Platform Creator.

For more information, see “Templates” and “Correcting the Port and Port-Protocol Parameters” in the ***Platform Creator User Manual***.

NOTE Make sure you have selected the correct protocol for the target port before you attempt to change the protocol-common parameters.

Chapter 4

AHB and AHB-Lite Bus Model

This chapter describes the Advanced High-performance Bus (AHB) and AHB-Lite Bus Model.

AHB-Bus Definition

The AHB acts as the high-performance system backbone bus, and forms the connection between ***AHBInitiator*** and ***AHBTarget*** protocols. An AHB bus is capable of supporting multiple masters through arbitration.

Advanced mechanisms such as split and retry operations are supported. For more information about these features, see the ***ARM AMBA v2.0 Specification***.

The AHB bus is instantiated in Platform Creator. For more information about this tool, see the ***Platform Creator User Manual***. Platform Creator then creates the source code for the bus topology which you described, based on the information you provided.

You can specify this information in Platform Creator as follows:

1. Select a specific protocol for each peripheral:
   - AHB for initiator ports
   - AHB or AHB-lite for target ports

   For more information, see "AHB Protocols" on page 16.
2. Select the correct address and data width for the initiator and target ports.
3. Select the correct addressing mode for the initiator and target ports.
4. Create the connections between the initiator ports, target ports, and buses. For more information, see "Creating Connections" in the ***Platform Creator User Manual***.
5. Specify the correct arbitration priorities for the initiator ports. See "Specifying the Arbitration Priority of an AHB Bus Target Port" on page 51.
6. Edit the memory map of your system. For more information, see "Memory Map" and "Using the GUI to Create Hardware Platforms" in the ***Platform Creator User Manual***.
7. Connect the clock.

You must use an AHB node for each AHB bus you want to create.

AHB-Lite-Bus Definition

The AHB-lite bus is very similar to the AHB bus. However, some features of the AMBA AHB are not available:

- It is not possible to connect more than one master to an AHB-lite bus, since arbitration is not supported.
- Split and retry are not supported.

The AHB-lite bus is instantiated in Platform Creator. For more information about this tool, see the ***Platform Creator User Manual***. Platform Creator then creates the source code for the bus topology you have described, based on the information you have provided.

You can specify this information in Platform Creator as follows:

1 Select the AHB-lite protocol for each peripheral. For more information, see "AHB Protocols" on page 16.

2 Select the correct address and data width for the initiator and target ports.

3 Select the correct addressing mode for the initiator and target ports.

4 Create the connections between the initiator ports, target ports, and buses. For more information, see "Creating Connections" in the ***Platform Creator User Manual***.

5 Specify the correct arbitration priorities for the initiator ports. See "Specifying the Arbitration Priority of an AHB Bus Target Port" on page 51.

6 Edit the memory map of your system. For more information, see "Memory Map" and "Using the GUI to Create Hardware Platforms" in the ***Platform Creator User Manual***.

7 Connect the clock.

You must use an AHB-lite node for each AHB-lite bus you want to create.

Preconditions

To be able to successfully generate an AHB-lite or AHB bus, all preconditions listed below must be met.

If there is a violation against one or more of these preconditions, error messages will be generated by Platform Creator during the specification of the bus.

The following preconditions must be fulfilled for both the AHB-lite and the AHB bus:

- At least one initiator port and one target port must be connected to the bus.
- The protocol of each initiator port and each target port must be an AHB or AHB-lite protocol, depending on the type of the bus. For information on the available protocols, see "AHB Protocols" on page 16. Both TLM and pin-accurate protocols are available.
- The width of the data terminals of the initiator ports must be ***8***, ***16***, ***32***, or ***64*** bits.
- Initiator ports must use byte addressing.
- The memory locations of AHB slaves must be aligned on a 1 KB boundary, as specified in the ***AMBA Specification*** (Rev 2.0).

The following describes:

- Preconditions for the AHB Bus
- Preconditions for the AHB-Lite Bus

Preconditions for the AHB Bus

The following preconditions must be fulfilled:

- The priority of each initiator port must be larger than or equal to zero and smaller than the number of initiator ports (see “Specifying the Arbitration Priority of an AHB Bus Target Port” on page 51).
- The priority of each master must be unique.
- The protocol of the initiator port must be an ***AHBInitiator*** protocol. You cannot connect an ***AHBLiteInitiator*** protocol to an AHB bus.

  Only one initiator port can be the default master. The default master is the initiator port that is granted the bus when no initiator port requests the bus. Since only one initiator port at a time can be granted the bus, you can specify only one default master. To specify which initiator port is the default master, see “Setting the Default Master” on page 52.
- As suggested by the ARM AMBA AHB specification (see the ***AMBA Specification***), the number of initiator ports is limited to 16. Because a dummy master is required in a split-capable arbiter, the number of initiator ports that can be connected to a bus is limited to 15, when one of the connected target ports is able to handle split accesses.
- Each of the initiator ports must be one of the following:
  - An AHB initiator (user-defined peripheral).
  - An ***AHBLite2AHB*** bridge. For more information, see Chapter 7, “Lite2AHB Bridge,” on page 91.
- Each of the target ports must be one of the following:
  - An AHB target (user-defined peripheral). The protocol of this target port can be either ***AHBTarget*** or ***AHBLiteTarget***.
  - An input-stage bus. For more information, see Chapter 5, “Input-Stage and Output-Stage Bus Model,” on page 65.
  - An APB node. For more information, see Chapter 6, “APB Bus Model,” on page 81.
  - An AHB-lite bus.
  - A ***Downsizer*** bridge, for more information, see Chapter 8, “Downsizer Bridge,” on page 95.

Preconditions for the AHB-Lite Bus

The following preconditions must be fulfilled:

- Only one initiator port can be connected.
- Each of the initiator ports must be one of the following:
  - A user-defined peripheral with an ***AHBLiteInitiator*** protocol.
  - An AHB or AHB-lite bus.
  - An output-stage bus. For more information, see Chapter 5, “Input-Stage and Output-Stage Bus Model,” on page 65.
  - A ***Downsizer*** bridge, for more information, see Chapter 8, “Downsizer Bridge,” on page 95.
- Each of the target ports must be one of the following:
  - A user-defined peripheral with an ***AHBLiteTarget*** protocol.
  - An AHB-lite bus.
  - An input-stage bus. For more information, see Chapter 5, “Input-Stage and Output-Stage Bus Model,” on page 65.
  - An APB bus. For more information, see Chapter 6, “APB Bus Model,” on page 81.
  - A ***Downsizer*** bridge, for more information, see Chapter 8, “Downsizer Bridge,” on page 95.

Setting Parameters

The following describes the parameters that can be set in Platform Creator. These parameters do not have an effect at the PV abstraction level.

- AHB and AHB-Lite Bus Parameters
- AHB-Bus-Specific Parameters

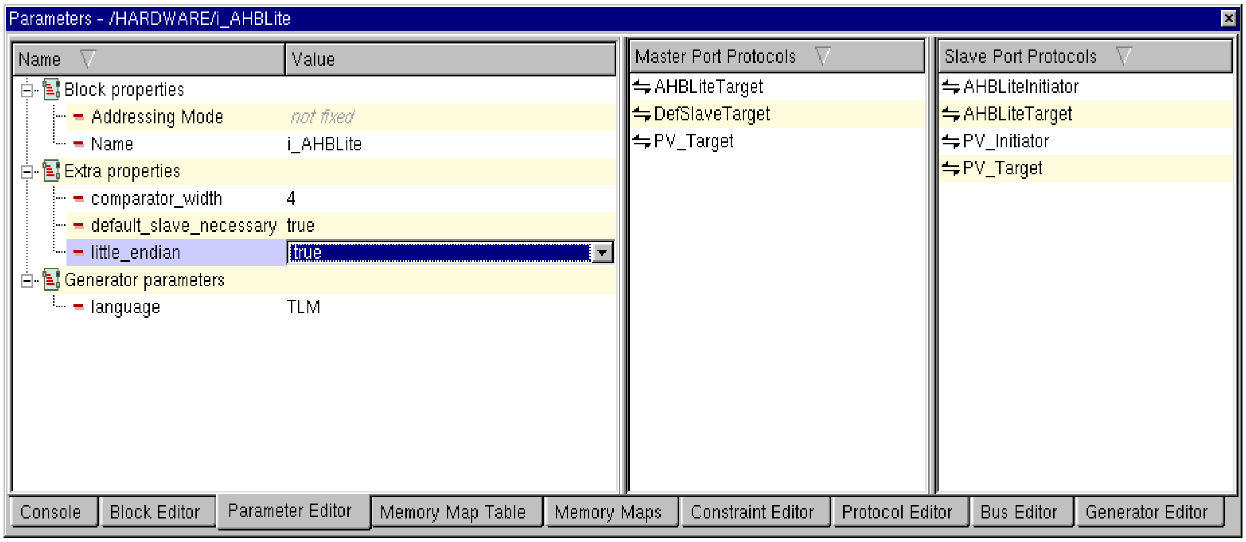
AHB and AHB-Lite Bus Parameters

The parameters described in the following are available on both AHB and AHB-lite buses.

- Specifying the Endianess
- Specifying Whether the Bus Has a Default Slave
- Specifying the Comparator Width

Specifying the Endianess

In the parameters of the bus, you can specify whether the bus uses the little-endian or big-endian mode.



Changing the endianess has no influence when using only TLM peripherals, since this is abstracted away at the TLM abstraction level.

Specifying Whether the Bus Has a Default Slave

For more information about the default slave, see the ***ARM AMBA AHB specification Rev. 2***.

To specify bus parameters:

1 In Platform Creator, select the bus for which you want to edit the default slave setting.

2 Edit the parameters of the bus, as shown in the above figure. Possible values are:

- ***true*** enable the default slave of the bus.
- ***false*** disable the default slave of the bus.

For more information, see “Bus-Node and Port Parameters” in the ***Platform Creator User Manual***.

Specifying the Comparator Width

This parameter has no influence on the generated TLM bus. It is used by the AMBA RTL generator when the address decoder is built. For more information, see Chapter 10, “Generating an RTL Bus,” on page 105.

AHB-Bus-Specific Parameters

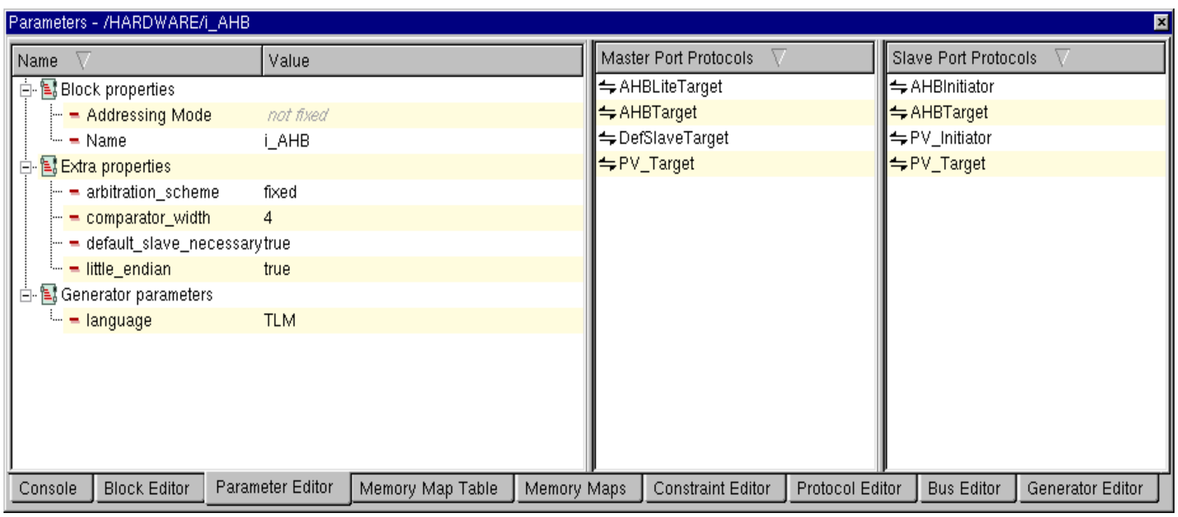
The following parameters are only applicable to the AHB bus. They are not available on an AHB-lite bus.

- Specifying the Arbitration Scheme
- Specifying the Arbitration Priority of an AHB Bus Target Port
- Setting the Default Master

Specifying the Arbitration Scheme

To specify the arbitration scheme:

1. In Platform Creator, select the AHB bus for which you want to set the arbitration scheme.
2. Edit the parameters of the AHB bus as shown in the following figure. The available arbitration schemes are:
   - Fixed-priority arbitration
   - Round-robin arbitration



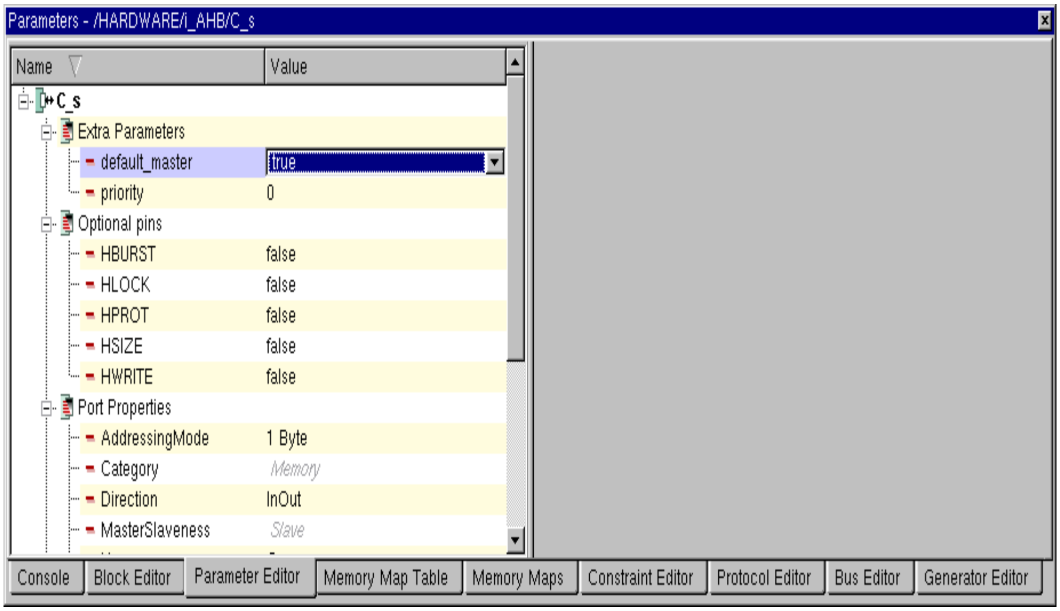
For more information, see “Bus-Node and Port Parameters” in the ***Platform Creator User Manual***.

Specifying the Arbitration Priority of an AHB Bus Target Port

To specify the arbitration priority of an AHB bus target port:

1 In Platform Creator, select the target port on the AHB bus for which you want to set the arbitration priority.

2 Edit the parameters of that port as shown in the following figure.



For more information, see “Bus-Node and Port Parameters” in the ***Platform Creator User Manual***.

NOTE The priority of each target port of the bus must be unique for that bus.

The priority can be anything between ***0*** (highest priority), and ***n–1***, where ***n*** is the number of initiator ports connected to that bus.

When you have selected the round-robin arbitration scheme (see “Specifying the Arbitration Scheme” on page 50), the specified priority reflects the order in which the initiator ports can be granted.

Setting the Default Master

When no initiator port requests the bus, the arbiter grants the default master the bus. When this initiator port then wants to do accesses, it does not have to wait anymore until it is granted by the arbiter. If this default master is chosen well, this can make your system some clock cycles faster.

To set the default master in Platform Creator, select the target port on the AHB bus which is connected to the initiator port that is to be the default master. Edit the parameters of that port. For more information, see “Bus-Node and Port Parameters” in the ***Platform Creator User Manual***.

As shown in the above figure, a ***default_master*** field is available. When you enable it, the port becomes the default master. Any other default master setting on the same bus will be reset. If you now want to select another default master, you first need to select the appropriate target port and repeat the procedure.

Analysis Views

The AMBA Bus Library enables views of the bus analysis library offered by System Designer. These analysis views enable you to get a complete overview of the strong and weak points of your design. The trace views can also be used to debug the design by monitoring the transactions between the masters and the slaves. If the standard analysis views do not give enough information, you can write your own analysis views. The procedure is described in “Creating a User-Defined Bus Analysis View” on page 54.

For information on the views of the bus analysis library, see “Bus Analysis Library” in the ***Analysis Manual***.

The following describes:

- Local Analysis Views for the AHB Bus
- Local Analysis Views for the AHB-Lite Bus
- Global Analysis Views
- Creating a User-Defined Bus Analysis View

Local Analysis Views for the AHB Bus

The following local analysis views are available for the AHB bus:

- Bus contention statistics
- Bus contention trace
- Master wait total
- Master wait trace
- Transaction count
- Transaction duration
- Transaction throughput
- Transaction trace
- Transaction utilization

The start of the wait of a particular master is defined as the moment the master has requested the bus but is not granted the bus. The end of the wait is defined as the moment the master starts a data transaction.

The transaction trace view shows the beginning and the end of all data transactions, that is, the data phase of every transaction.

Local Analysis Views for the AHB-Lite Bus

The following local analysis views are available for the AHB-lite bus:

- Transaction count
- Transaction duration
- Transaction throughput
- Transaction trace
- Transaction utilization

The transaction trace view shows the beginning and the end of all data transactions, that is, the data phase of every transaction.

Global Analysis Views

The following global analysis views are available:

- Transaction count
- Transaction duration
- Transaction throughput
- Transaction trace
- Transaction utilization

The transaction trace view shows the beginning and the end of all data transactions, that is, the data phase of every transaction. If a global master is somewhere not granted by a bus, you will see the name of that bus in the transaction trace view.

Creating a User-Defined Bus Analysis View

- Creating a Class
- Creating the init() and update() Functions
- Functions for Analysis Views
- Hooking Up the Custom Global and Custom Local Analysis Views
- Example of a Custom Bus Analysis View
- Specifying String and Color Mappings
- Example of a Custom Bus Analysis Trace View

Creating a Class

To write a custom bus analysis view, you have to create a class which is derived from the ***AMBATLMAnalysisView*** class, as shown below.

```
class viewName : public AMBATLMAnalysisView {
  private :

  public :
     viewName( AMBANode *aNode,
              const char *aName) :
        AMBATLMAnalysisView(aNode, aName)
  {
  }

  virtual void update() {

  }

  virtual void init() {

  }
};
```

The constructor of your custom view has to call the constructor of the ***AMBATLMAnalysisView*** class.

Creating the init() and update() Functions

You have to create the functions ***init()*** and ***update()***.

- In the ***init()*** function, you have to create all the analysis elements for your view. You can also specify how values will be mapped to strings to make your view more readable. Further, you can specify the colors that will be used to display the different events.
- In the ***update()*** function, you have to specify which values will be put into the analysis database and when they will be put into the database.

For this, you can use the predefined events and you can access all attributes of all transfers to collect data for your analysis view. This is explained in:

- Accessing Attributes of Transfers
- Predefined Events

Accessing Attributes of Transfers

The following example illustrates accessing an attribute of a transfer:

```
AMBATLMEventInfo SofT = pollTLMEvent(curr_master_id, ASofTInitiator);
int EotStatusVal = SofT.transaction->EotTrf.Status;
```

Predefined Events

To determine the moment at which data is sent to the analysis database, you have to use the predefined events. For trace views this will then also be used to determine the moment where actions are started or stopped.

For the AHB bus, these predefined events are:

- ***ARequest*** is the moment at which the master has requested the bus, but is not granted the bus. So this means that ***ARequest*** happens when a data transaction should be started but the transaction is not started because the master is not granted the bus.
- ***AGrant*** is the moment at which the master starts a data transaction.
- ***ASofTInitiator*** is the moment at which the master starts a data transaction.
- ***ASofTTarget*** is the moment at which the data transaction starts at the target side.
- ***AEofTTarget*** is the moment at which the data transaction ends at the target side.

ARequest and ***AGrant*** are master-specific events. This means that these events will only happen when that particular master is accessing or trying to access the bus.

ASofTInitiator, ***ASofTTarget***, and ***AEofTTarget*** are bus-specific events. They happen when one of the masters and one of the slaves of that bus are interacting with each other. ***ASofTInitiator*** and ***ASofTTarget*** are exactly the same for the AHB bus.

For the AHB-lite bus, the same is true as for the AHB bus. The only difference is that ***ARequest*** and ***AGrant*** do not exist, since the bus only has one master by definition, which makes the request-grant mechanism obsolete.

Functions for Analysis Views

You can use the following functions to write your analysis view:

```
const vector<PeripheralInfo> &getAllInitiators() const;
```

The ***getAllInitiators()*** function returns a vector with as many entries as there are peripherals (so both initiators and targets), but only the initiators are filled in. So to check whether an entry is an initiator or not, see if ***vectorelement.port != 0***.

```
const vector<PeripheralInfo> &getAllTargets() const;
```

The ***getAllTargets()*** function returns a vector with as many entries as there are peripherals (so both initiators and targets), but only the targets are filled in. So to check whether an entry is a target or not, see if ***vectorelement.port != 0***.

```
string getInitiatorName(IDType initiatorID) const;

string getTargetName(IDType targetID) const;

const AMBAPort *getInitiator(IDType initiatorID) const;

const AMBAPort *getTarget(IDType targetID) const;

IDType getId(AMBAPort *port) const;
cwrBaseData *getAnalysisDataElement(IDType ID, string elemName);
```

cwrBaseData is the base class of ***cwrAnalysisData***. For more information on the ***cwrAnalysisData*** class, see "cwrAnalysisData Class" in the ***Analysis Manual***.

```
cwrBaseData *addAnalysisDataElement(IDType ID, const char *baseName, const char
*name, DataElementType det, cwrAnalysisView::PreProcessingOption prepProc);
```

```
PreProcessingOption { MIN, MAX, COV, AVE, TAV, SUM, TRC, CNT, SPS, CPS };
```

For more information, see the "cwrAnalysisData Class" of the ***Analysis Manual***.

```
cwrBaseData *getAnalysisDataElement(IDType ID1, IDType ID2, string elemName);

cwrBaseData *addAnalysisDataElement(IDType ID1, IDType ID2,
                                    const char *baseName, const char *name,
                                    DataElementType det,
                              cwrAnalysisView::PreProcessingOption preProc);
AMBATLMEventInfo pollTLMEvent (IDType nodeOrPeripheralID,
                               AMBAAnalysisEvent event);
```

pollTMEvent is of type ***AMBATLMEventInfo***, which has the following members:

- ***bool occurred;***
- ***AMBAPort* initiatorID;***
- ***AMBAPort* targetID;***
- ***AMBATransaction *transaction;***
- ***bool initiatorIsPeripheral;***
- ***bool targetIsPeripheral;***

```
void getDirtyTargets(AMBAAnalysisEvent event, set<IDType> &targetIDs);
```

A "dirty target" is a target which is currently busy with a transaction.

```
void getDirtyInitiators(AMBAAnalysisEvent event, set<IDType> &initIDs);
```

A "dirty initiator" is an initiator which is at this moment in time busy with a transaction.

Hooking Up the Custom Global and Custom Local Analysis Views

This custom analysis view should be written in a separate header file, which can have any name. You have to include this header file in the file that you created to hook up your peripherals to the platform dumped by Platform Creator. Further, you have to hook up your custom global analysis views and your custom local analysis views.

For example (in the ***system.cpp*** file):

```
// Hookup custom global view
MyGlobalView myGlobalView("custom_globalview");

// Hookup custom local views
MyAHBView myAHBView1(&bus.ahb_node1, "custom_ahbview_1");
MyAHBView myAHBView2(&bus.ahb_node2, "custom_ahbview_2");
MyAHBView myAHBView3(&bus.ahb_node3, "custom_ahbview_3");
```

For the global analysis view, you just need to give this view a name. The local analysis views also need the bus to which they belong as parameter. You can find the name of the buses in the ***platform.h*** file generated by Platform Creator. These are the names you have specified in your Platform Creator run.

Example of a Custom Bus Analysis View

The example below is a custom bus analysis view that counts the number of continuous transactions. With the ***setAttribute*** command you can specify the default value for the attributes of the view. These do not need to be set, they can be set manually in System Designer when using the view, but by setting them in your code, they will have a good default value. For more information, see “UnderStanding the Basic Chart Types” in the ***Analysis Manual***.

CustomAnaViews.h contains:

```
#ifndef CUSTOMANAVIEW_H
#define CUSTOMANAVIEW_H

class CustomAHBView : public AMBATLMAnalysisView {
private :

public :
 CustomAHBView( AMBANode *aNode,
               const char *aName ) :
   AMBATLMAnalysisView(aNode, aName)
 {
   setAttribute( "view", "BAR" );
   setAttribute( "scalex", "1000000" );
 }

 virtual void update() {
   set<IDType> Active_Targets;
   getDirtyTargets(ASofTTarget, Active_Targets);
   getDirtyTargets(AEofTTarget, Active_Targets);

   set<IDType>::iterator b = Active_Targets.begin();
   set<IDType>::iterator e = Active_Targets.end();

   for (; b!=e; ++b) {
     AMBATLMEventInfo SofT = pollTLMEvent(*b, ASofTTarget);
     AMBATLMEventInfo EofT = pollTLMEvent(*b, AEofTTarget);

     if (EofT.occurred) {
       if ( SofT.occurred ) {
         if ( EofT.initiatorID == SofT.initiatorID ) {
           // still a continuous transaction
```

```
        } else {
          // transaction with a different master
          // so continuous transaction has ended and can be counted
          string n = "cont_transaction";
          cwrAnalysisData<long> *d =
            static_cast<cwrAnalysisData<long>*>(getAnalysisDataElement
                                    (EofT.initiatorID->analysisID, n));
          d->collect(1);
        }
      } else {
        // continuous transaction has ended and can be counted
        string n = "cont_transaction";
        cwrAnalysisData<long> *d =
         static_cast<cwrAnalysisData<long>*>(getAnalysisDataElement
                                    (EofT.initiatorID->analysisID, n));
        d->collect(1);
      }
    }
  }
}

virtual void init() {
  vector<PeripheralInfo>::const_iterator b = getAllInitiators().begin();
  vector<PeripheralInfo>::const_iterator e = getAllInitiators().end();

  for ( ; b != e; ++b ) {
    if ( (*b).port != 0 ) {
      string n = "cont_transaction_";
      n += (*b).name;

      // Create data element for this slave
      cwrBaseData *d = addAnalysisDataElement( (*b).id, "cont_transaction",
                                 n.c_str(), detLong, cwrAnalysisView::CNT );
    }
  }
}

};

#endif
```

system.cpp contains:

```
#include <systemc.h>
#include "platform.h"
#include "Full_AHB_FRBM.h"
#include "Full_AHB_FRBS.h"
#include "AMBA/cwr_amba_default_slave.h"

#include "CustomAHBView.h"

int sc_main (int argc, char *argv[]) {

   // instantiate some sc_modules

  Full_FRBM<> master1("master1î);
  Full_FRBM<> master2("master2");
  Full_FRBM<> master3("master3");
  Full_FRBS_RW slave1("slave1");
  Full_FRBS_RW slave2("slave2");
  AHBLite_defslave defslave("defslave");

  // instantiate the sc_module that contains the bus simulators
  // this sc_module is generated by PCT.
  // The name of this module ("MyPlatform") is chosen by the Platform
  // Creator user.
  MyPlatform bus("bus");

  sc_clock clk("Clock", 4, SC_NS, 0.5, 0, SC_NS, false);
  bus.clk(clk);
  master1.clk(clk);
  master2.clk(clk);
  master3.clk(clk);
  slave1.clk(clk);
  slave2.clk(clk);

  // hookup masters and slaves to the platform.
  //
  // the names "master1", "master2",... are the names of the ports
  // chosen by the PCT-user.

  master1.p(*bus.master1);
  master2.p(*bus.master2);
  master3.p(*bus.master3);
  defslave.p(*bus.defslave);
  slave1.p(*bus.slave1);
  slave2.p(*bus.slave2);

  CustomAHBView CustomAHBView(&bus.ahb_node, "ContTransCount");

  sc_start();

  return 0;
}
```

Specifying String and Color Mappings

Especially in trace views it can be very useful to do a mapping of the possible values to meaningful strings, like for example the name of the target that is being accessed.

To make your view more clear, you can specify the following:

- The string that will be shown for a specific analysis value
- The string that will appear in the pop-up window when you place your cursor above an analysis element
- The border color of the different analysis elements
- The fill color of the different analysis elements

When you want to specify one of the above items, you need to declare a ***cwrMapAttrib*** for it and you need to include ***cwrMapAttrib.h*** in the header file of your custom analysis view.

You have to use the ***insert(string `value`, string `result`)*** function to specify the mapping.

After you have specified all mappings, you have to set the attributes for the element you want to define. First, you have to use ***setAttribute*** to specify a string for you ***cwrMapAttrib***, then you need to specify for which attribute you want to use this map.

For example (fill color):

```
setAttribute("value2fillcolor_map", fillcolorMap);
setAttribute("fillcolor", "$PARAM_MAP(value2fillcolor_map, $DATAFIELD(value))");
```

You can set the following attributes:

- ***datalabel***
- ***popuplabel***
- ***fillcolor***
- ***bordercolor***

Example of a Custom Bus Analysis Trace View

In the example below a trace view is specified for AHB, which gives information on the name of the target that is being accessed, the type of response that is issued by the target, the mode of the transaction (nonsequential, sequential, or busy) and whether the lock is set on or off. When the response of a transaction is not “ok”, this is highlighted by displaying that transaction in red.

CustomAHBTraceView.h contains:

```
#ifndef CUSTOMAHBTRACEVIEW_H
#define CUSTOMAHBTRACEVIEW_H

#include "cwrMapAttrib.h"

class CustomTraceView : public AMBATLMAnalysisView {
private :

public :
  CustomTraceView( AMBANode *aNode,
                  const char *aName ) :
   AMBATLMAnalysisView(aNode, aName)
  {

  }

  virtual void update() {
    set<IDType> Active_Targets;
    getDirtyTargets(ASofTTarget, Active_Targets);
    getDirtyTargets(AEofTTarget, Active_Targets);

    set<IDType>::iterator b = Active_Targets.begin();
    set<IDType>::iterator e = Active_Targets.end();

    string n = "utilization";

    // Go over all active targets.
    // If the target is busy with the end of a transaction (EofT),
    // set the AnalysisDataElement of the initiator which was performing a
    // transaction with that target, to 0. This means that there is no
    // transaction anymore.
    for (; b!=e; ++b) {
      AMBATLMEventInfo EofT = pollTLMEvent(*b, AEofTTarget);

      if ( EofT.occurred ) {
        cwrAnalysisData<long> *d =
         static_cast<cwrAnalysisData<long>*>(getAnalysisDataElement
                                  (EofT.initiatorID->analysisID, n));
        d->collect(0);
      }
    }

    b = Active_Targets.begin();

    // Go over all active targets.
    // If the target is busy with the start of a transaction (SofT),
    // find out information on that transaction:
    // response: ok, error, retry or split
    // mode: nonsequential, sequential or busy
    // lock: lock or no lock
    // Set the AnalysisDataElement of the initiator which was performing a
    // transaction with that target, to the value according to the transaction
    // information.
    for (; b!=e; ++b) {
      AMBATLMEventInfo SofT = pollTLMEvent(*b, ASofTTarget);

      if ( SofT.occurred ) {
           int EotStatusVal = SofT.transaction->EotTrf.Status;
           if (EotStatusVal == tlmSplit) { // check if there is a split response
            cwrAnalysisData<long> *d
                    static_cast<cwrAnalysisData<long>*>(getAnalysisDataElement
                                          (SofT.initiatorID->analysisID, n));
            d->collect(SofT.targetID->analysisID + 0x3000);
            } else if (EotStatusVal == tlmRetry) { // check if there is a retry response
            cwrAnalysisData<long> *d =
             static_cast<cwrAnalysisData<long>*>(getAnalysisDataElement
                                          (SofT.initiatorID->analysisID, n));
            d->collect(SofT.targetID->analysisID + 0x2000);
            } else if (EotStatusVal == tlmError) { // check if there is an error response
            cwrAnalysisData<long> *d =
             static_cast<cwrAnalysisData<long>*>(getAnalysisDataElement
                                          (SofT.initiatorID->analysisID, n));
            d->collect(SofT.targetID->analysisID + 0x1000);
            } else { // there is a "normal" ok response
```

```
            int TransTypeVal = SofT.transaction->AddrTrf.Type;
            int LockVal = SofT.transaction->LockTrf.Lock;
            int BurstGroupVal = SofT.transaction->AddrTrf.Group;
            if (TransTypeVal == tlmReadAtAddress ||
                TransTypeVal == tlmWriteAtAddress) {
             if (BurstGroupVal == tlmSingle ||
              BurstGroupVal == tlmBurstStart) { // there is a nonsequential transfer
              cwrAnalysisData<long> *d =
                static_cast<cwrAnalysisData<long>*>(getAnalysisDataElement
                                          (SofT.initiatorID->analysisID, n));
              d->collect(0x10000*(2*(LockVal == 0) + 5*(LockVal == 1)) +
              SofT.targetID->analysisID);
             } else if (BurstGroupVal == tlmBurstCont) { // there is sequential transfer
             cwrAnalysisData<long> *d =
                static_cast<cwrAnalysisData<long>*>(getAnalysisDataElement
                                          (SofT.initiatorID->analysisID, n));
              d->collect(0x10000*(3*(LockVal == 0) + 6*(LockVal == 1)) + SofT.targetID->analysisID);
             }
             } else if (TransTypeVal == tlmIdle && BurstGroupVal ==
                                    tlmBurstIdle) { // there is a busy transfer
             cwrAnalysisData<long> *d =
                static_cast<cwrAnalysisData<long>*>(getAnalysisDataElement
                                          (SofT.initiatorID->analysisID, n));
             d->collect(0x10000*((LockVal == 0) + 4*(LockVal == 1)) + SofT.targetID->analysisID);
             }
           }
          }
     }
 }

 virtual void init() {
   // The "getAllInitiators" function returns a vector with as many entries as
   // there are peripherals (so both initiators and targets), but only the
   // initiators are filled in.
   // So to check whether an entry is an initiator or not, see if
   // vector-element.port != 0 (this check is used later on in the code).
   vector<PeripheralInfo>::const_iterator b_i = getAllInitiators().begin();
   vector<PeripheralInfo>::const_iterator e_i = getAllInitiators().end();

   cwrMapAttrib value2string;
   cwrMapAttrib popupMap;
   cwrMapAttrib fillcolorMap;
   cwrMapAttrib bordercolorMap;

   string tra_color = "$ffffb0"; // sets the color to an RGB value of light yellow

   // Go over all initiators.
   // Set the data label and the colors for all possible values.
   for ( ; b_i != e_i; ++b_i ) {
     // we need to check whether this entry of the peripherals vector is
     // an initiator or not
     if ( (*b_i).port != 0 ) {
       string n = "utilization";
       n += (*b_i).name; // name of the initiator

      // Create data element for this initiator
      cwrBaseData *d = addAnalysisDataElement( (*b_i).id, "utilization", n.c_str(), detLong,
                                                                cwrAnalysisView::TRC );

      // The "getAllTargets" function returns a vector with as many entries as
      // there are peripherals (so both initiators and targets), but only the
      // targets are filled in.
      // So to check whether an entry is a target or not, see if
      // vector-element.port != 0 (this check is used later on in the code).
      vector<PeripheralInfo>::const_iterator b_t = getAllTargets().begin();
      vector<PeripheralInfo>::const_iterator e_t = getAllTargets().end();

      // Go over all initiators.
      // Set the data label and the colors for all possible values.
      for ( ; b_t != e_t; ++b_t ) {
       // we need to check whether this entry of the peripherals vector is
       // a target or not
       if ( (*b_t).port != 0 ) {
        // Here the string that will appear in the analysis view
        // can be specified. As well as the string that is used
        // in the popup window and the colors that will be used to
```

```
// display the data.
// The string that will appear in the analysis view will be
// set to the name of the target.
// The popup window will display information like
// the name of the target, the type of response,
// the mode of the transaction and information on the lock.
// Split, retry and error response will be displayed in red.
// The "ok" responses will be colored in light yellow.
char str[100];
sprintf(str, "%i", b_t->id + 0x3000);
value2string.insert(str, (*b_t).name);
char t[1000];
sprintf(t, "target = %s\n"
        "response = split",(*b_t).name.c_str());
// When the analysis data equals the target id plus
// 0x3000, we know that there is a split response
// issued by that target (see function update() at
// the top of this file).
popupMap.insert(str, t);
fillcolorMap.insert(str, "RED");
bordercolorMap.insert(str, "BLACK");
sprintf(str, "%i", b_t->id + 0x2000);
value2string.insert(str, (*b_t).name);
sprintf(t, "target = %s\n"
       "response = retry",(*b_t).name.c_str());
fillcolorMap.insert(str, "RED");
bordercolorMap.insert(str, "BLACK");
popupMap.insert(str, t);
sprintf(str, "%i", b_t->id + 0x1000);
value2string.insert(str, (*b_t).name);
sprintf(t, "target = %s\n"
        "response = error",(*b_t).name.c_str());
fillcolorMap.insert(str, "RED");
bordercolorMap.insert(str, "BLACK");
popupMap.insert(str, t);
sprintf(str, "%i", b_t->id + 131072);
value2string.insert(str, (*b_t).name);
sprintf(t, "target = %s\n"
        "response = ok\n"
        "mode = nonsequential\n"
        "lock = off",(*b_t).name.c_str());
popupMap.insert(str, t);
fillcolorMap.insert(str, tra_color);
bordercolorMap.insert(str, "BLACK");
sprintf(str, "%i", b_t->id + 327680);
value2string.insert(str, (*b_t).name);
sprintf(t, "target = %s\n"
        "response = ok\n"
        "mode = nonsequential\n"
        "lock = on",(*b_t).name.c_str());
popupMap.insert(str, t);
fillcolorMap.insert(str, tra_color);
bordercolorMap.insert(str, "BLACK");
sprintf(str, "%i", b_t->id + 196608);
value2string.insert(str, (*b_t).name);
sprintf(t, "target = %s\n"
        "response = ok\n"
        "mode = sequential\n"
        "lock = off",(*b_t).name.c_str());
popupMap.insert(str, t);
fillcolorMap.insert(str, tra_color);
bordercolorMap.insert(str, "BLACK");
sprintf(str, "%i", b_t->id + 393216);
value2string.insert(str, (*b_t).name);
sprintf(t, "target = %s\n"
        "response = ok\n"
        "mode = sequential\n"
        "lock = on",(*b_t).name.c_str());
popupMap.insert(str, t);
fillcolorMap.insert(str, tra_color);
bordercolorMap.insert(str, "BLACK");
sprintf(str, "%i", b_t->id + 0x10000);
value2string.insert(str, (*b_t).name);
sprintf(t, "target = %s\n"
        "response = ok\n"
        "mode = busy\n"
```

```
                "lock = off",(*b_t).name.c_str());
        popupMap.insert(str, t);
        fillcolorMap.insert(str, tra_color);
        bordercolorMap.insert(str, "BLACK");
        sprintf(str, "%i", b_t->id + 262144);
        value2string.insert(str, (*b_t).name);
        sprintf(t, "target = %s\n"
                "response = ok\n"
                "mode = busy\n"
                "lock = on",(*b_t).name.c_str());
        popupMap.insert(str, t);
        fillcolorMap.insert(str, tra_color);
        bordercolorMap.insert(str, "BLACK");
        bordercolorMap.insert("0", "NONE");
       }
      }
     }
    }
    setAttribute("value2string_map", value2string);
    setAttribute("datalabel", "$PARAM_MAP(value2string_map, $DATAFIELD(value))");
    setAttribute("value2popup_map", popupMap);
    setAttribute("popuplabel", "$PARAM_MAP(value2popup_map, $DATAFIELD(value))");
    setAttribute("value2fillcolor_map", fillcolorMap);
    setAttribute("fillcolor", "$PARAM_MAP(value2fillcolor_map, $DATAFIELD(value))");
    setAttribute("value2bordercolor_map", bordercolorMap);
    setAttribute("bordercolor", "$PARAM_MAP(value2bordercolor_map, $DATAFIELD(value))")
  }

};

#endif
```

Chapter 5

Input-Stage and Output-Stage Bus Model

This chapter describes the Input-Stage and Output-Stage Bus Model.

- Multilayer Definition
- Preconditions
- Setting Parameters
- Analysis Views

Multilayer Definition

A multilayer structure consists of a number of input stages and a number of output stages. There is no real multilayer bus model, but a combination of input-stage an output-stage bus models.

The input-stage an output-stage bus models together form the multilayer AHB interconnection scheme, as described in “Multilayer AHB Overview” of the AMBA user documentation. This multilayer AHB interconnection scheme, which is based on the AHB protocol, enables parallel access paths between multiple initiators and targets in a system. This is achieved by using a more complex interconnection matrix and gives the benefit of increased overall bus bandwidth, and a more flexible system architecture.

The following describes:

- Input-Stage Node Definition
- Output-Stage Node Definition
- Forming a Multilayer Bus with Input-Stage and Output-Stage Buses

Input-Stage Node Definition

An input stage is responsible for holding the address and control information when the transfer to a shared slave cannot start immediately. The generated input stage is based on the input stage of the ***AHB Example AMBA System - ARM 1DDI 0170A***.

Output-Stage Node Definition

An output stage is used to select which of the various input layers is routed to the target. The generated output stage is based on the output stage of the ***AHB Example AMBA System - ARM DDI 0170A***.

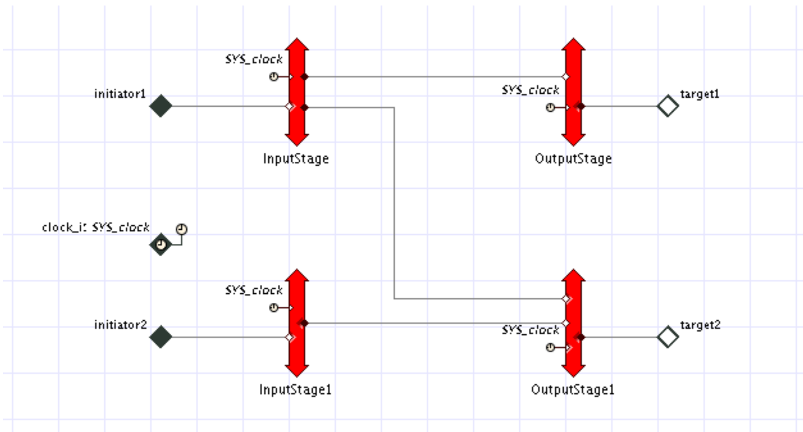
Forming a Multilayer Bus with Input-Stage and Output-Stage Buses

For every initiator connected to the multilayer structure, you need an input-stage bus.

For every target connected to the multilayer structure, you need an output-stage bus.

You then need to connect the input-stage bus to the output-stage bus which is connected to a target which the input-stage bus (or the initiator connected to it) must be able to access.

This gives you, for example, the following structure:



In this case, ***Initiator1*** can access ***target1*** and ***target2***.

Initiator2 can only access ***target2***.

Preconditions

To be able to successfully generate an input-stage or output-stage bus, all preconditions listed below must be met.

If there is a violation against one or more of these preconditions, error messages will be generated by Platform Creator during the specification of the bus.

The following preconditions must be fulfilled for both the input-stage and the output-stage bus:

- At least one initiator port and one target port must be connected to the bus.
- The protocol of each initiator port and each target port must be an AHB or AHB-lite protocol, depending on the type of the bus. For information on the available protocols, see "AHB Protocols" on page 16. Both TLM and pin-accurate protocols are available.
- The width of the data terminals of the initiator and target ports must be ***8***, ***16***, ***32***,or ***64*** bits.
- Initiator ports must use byte addressing.
- The memory locations of AHB slaves must be aligned on a 1 KB boundary, as specified in the ***AMBA Specification*** (Rev 2.0).
- For each remap ID, there must be at least one peripheral with a memory region for that remap ID.
- Each peripheral must have a memory region for at least one remap ID.

The following describes:

- Preconditions for the Input-Stage Bus
- Preconditions for the Output-Stage Bus

Preconditions for the Input-Stage Bus

The following preconditions must be fulfilled:

- The priority of each master must be unique.
- Only one initiator port can be connected to the input-stage.
- Multiple output stages can be connected to an input stage.
- Each of the initiator ports must be one of the following:
  - An AHB initiator (user-defined peripheral).
  - An AHB-lite initiator (user-defined peripheral).
  - An AHB-lite bus (see Chapter 4, “AHB and AHB-Lite Bus Model,” on page 45).
  - An AHB bus (see Chapter 4, “AHB and AHB-Lite Bus Model,” on page 45).
  - An output-stage node.
  - A ***Downsize*** bridge. For more information, see Chapter 8, “Downsizer Bridge,” on page 95.
- Each of the target ports must be:
  - An output-stage bus

Preconditions for the Output-Stage Bus

The following preconditions must be fulfilled:

- Only one target port can be connected.
- Each of the initiator ports must be one of the following:
  - A input-stage bus
- Multiple input-stage buses can be connected to the output-stage bus.
- Each of the target ports must be one of the following:
  - A user-defined peripheral with an AHB-lite protocol.
  - An AHB-lite bus.
  - An input-stage bus.
  - An APB bus. For more information, see Chapter 6, “APB Bus Model,” on page 81.
  - A ***Downsize*** bridge. For more information, see Chapter 8, “Downsizer Bridge,” on page 95.

For more information about the AHB and AHB-lite protocols, see “AHB Protocols” on page 16.

Setting Parameters

The following describes the parameters that can be set in Platform Creator. These parameters do not have an effect at the PV abstraction level.

- Input-Stage and Output-Stage Bus Parameters
- Input-Stage-Bus-Specific Parameters
- Output-Stage-Bus-Specific Parameters

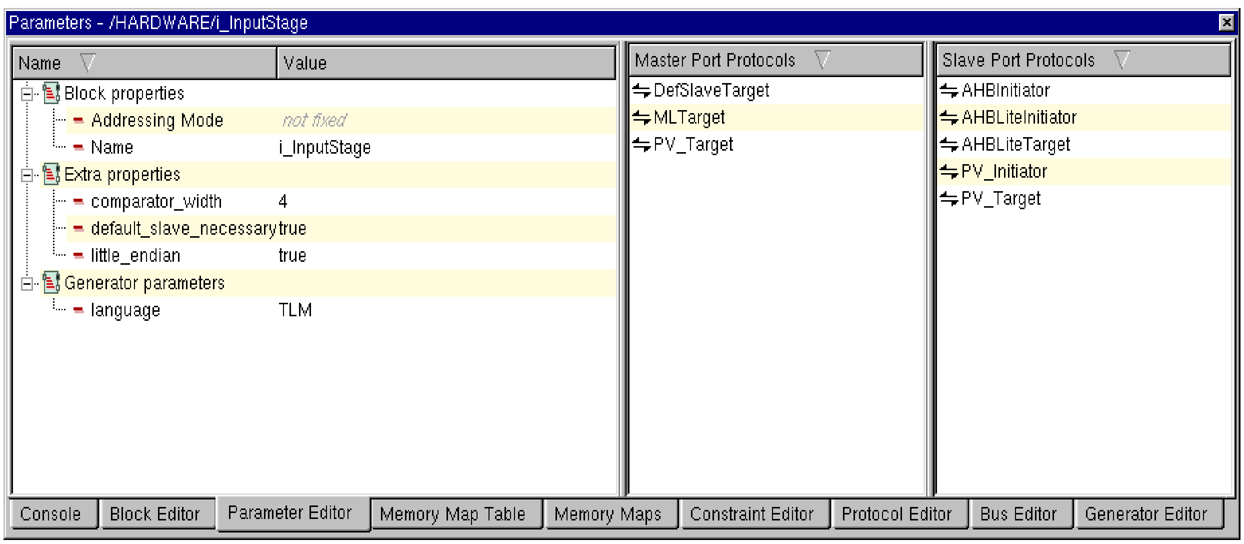
Input-Stage and Output-Stage Bus Parameters

The parameters described in the following are available on both input-stage and output-stage buses.

- Specifying the Endianess

Specifying the Endianess

In the parameters of the bus, you can specify whether the bus uses the little-endian or big-endian mode.



Changing the endianess has no influence when using only TLM peripherals, since this is abstracted away at the TLM abstraction level.

Input-Stage-Bus-Specific Parameters

The following parameters are only applicable to the input-stage bus. They are not available on an output-stage bus.

- Specifying Whether the Bus Has a Default Slave
- Specifying the Comparator Width

Specifying Whether the Bus Has a Default Slave

For more information about the default slave, see the ***ARM AMBA AHB Specification*** (Rev. 2).

To specify whether the bus has a default slave:

1 In Platform Creator, select the bus for which you want to edit the default slave setting.

2 Edit the parameters of the bus as shown in the above figure. The options are:

- ***true*** enables the default slave of the node.
- ***false*** disables the default slave of the node.

NOTE If no default slave is present, and an access is done to memory space not allocated to a certain peripheral, the simulation will exit.

For more information, see “Bus-Node and Port Parameters” in the ***Platform Creator User Manual***.

Specifying the Comparator Width

This parameter has no influence on the generated TLM bus. It is used by the AMBA RTL generator when the address decoder is built. For more information, see Chapter 10, “Generating an RTL Bus,” on page 105.

Output-Stage-Bus-Specific Parameters

- Specifying the Arbitration Scheme
- Specifying the Arbitration Priority of an Output-Stage Bus Target Port

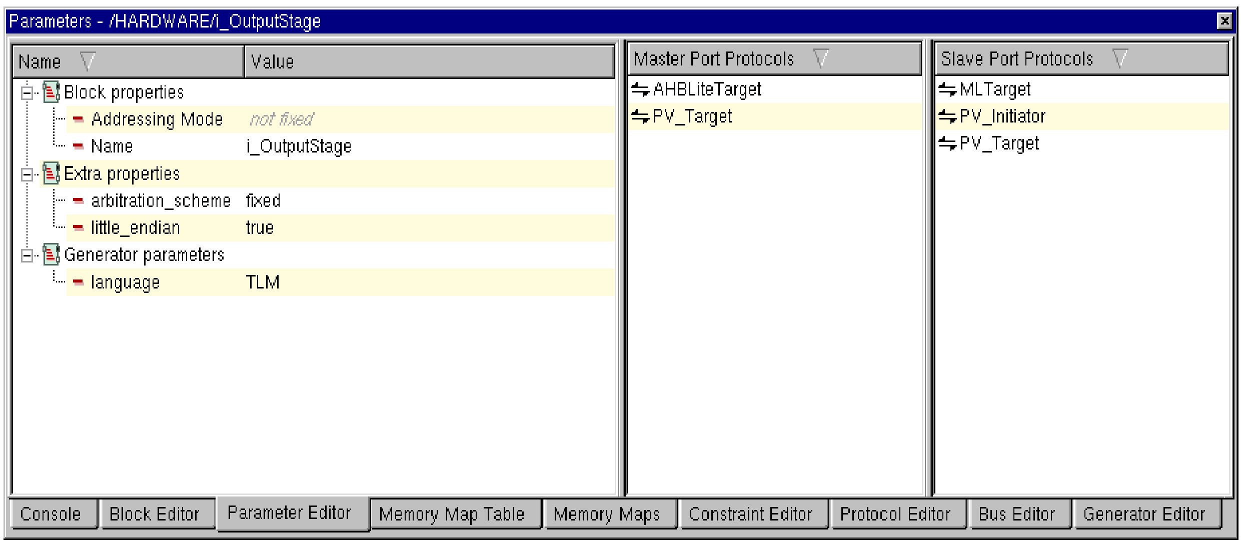
Specifying the Arbitration Scheme

To specify the arbitration scheme:

1 In Platform Creator, select the output-stage bus for which you want to set the arbitration scheme.

2 Edit the parameters of the output-stage bus as shown in the following figure. The available arbitration schemes are:

- Fixed-priority arbitration
- Round-robin arbitration



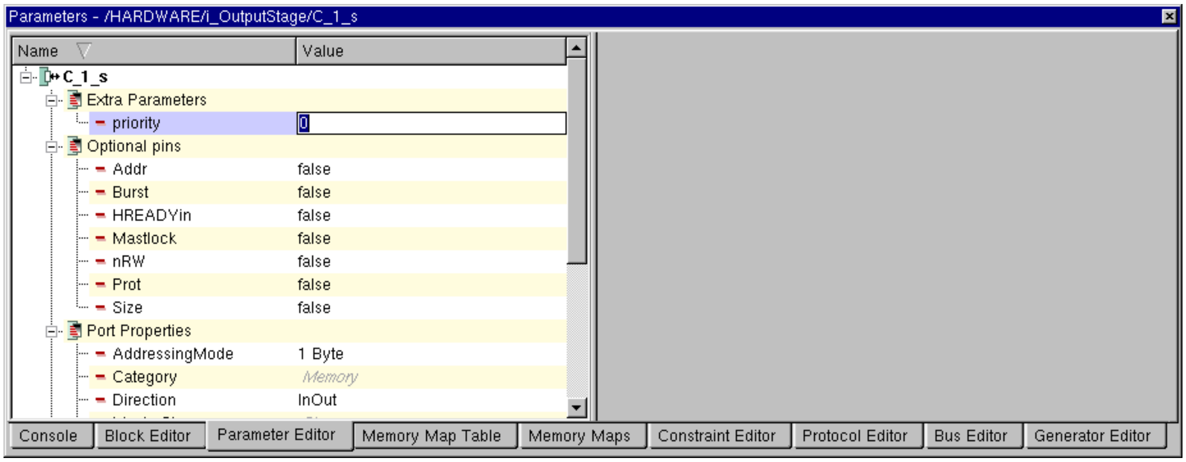
For more information, see “Bus-Node and Port Parameters” in the ***Platform Creator User Manual***.

Specifying the Arbitration Priority of an Output-Stage Bus Target Port

To specify the arbitration priority of an output-stage bus:

1 In Platform Creator, select the target port on the output-stage bus for which you want to set the arbitration priority.

2 Edit the parameters of that port as shown in the following figure.



For more information, see “Bus-Node and Port Parameters” in the ***Platform Creator User Manual***.

NOTE The priority of each target port of the bus must be unique for that bus.

The priority can be anything between ***0*** (highest priority), and ***n***–***1***, where ***n*** is the number of initiator ports connected to that bus.

When you have selected the round-robin arbitration scheme (see “Specifying the Arbitration Scheme” on page 70), the specified priority reflects the order in which the initiator ports can be granted.

Analysis Views

The AMBA Bus Library enables views of the bus analysis library offered by System Designer. These analysis views enable you to get a complete overview of the strong and weak points of your design. The trace views can also be used to debug the design by monitoring the transactions between the masters and the slaves. If the standard analysis views do not give enough information, you can write your own analysis views. The procedure is described in "Creating a User-Defined Bus Analysis View" on page 73.

For information on the views of the bus analysis library, see "Bus Analysis Library" in the ***Analysis Manual***.

The following describes:

- Local Analysis Views for the Output-Stage Bus
- Local Analysis Views for the Input-Stage Bus
- Global Analysis Views
- Creating a User-Defined Bus Analysis View

Local Analysis Views for the Output-Stage Bus

The following local analysis views are available for the AHB bus:

- Bus contention statistics
- Bus contention trace
- Master wait total
- Master wait trace
- Transaction count
- Transaction duration
- Transaction throughput
- Transaction trace
- Transaction utilization

The start of the wait of a particular master is defined as the moment the master has requested the bus but is not granted the bus. The end of the wait is defined as the moment the master starts a data transaction.

The transaction trace view shows the beginning and the end of all data transactions, that is, the data phase of every transaction.

Local Analysis Views for the Input-Stage Bus

The following local analysis views are available for the AHB-lite bus:

- Transaction count
- Transaction duration
- Transaction throughput
- Transaction trace
- Transaction utilization

The transaction trace view shows the beginning and the end of all data transactions, that is, the data phase of every transaction.

Global Analysis Views

The following global analysis views are available:

- Transaction count
- Transaction duration
- Transaction throughput
- Transaction trace
- Transaction utilization

The transaction trace view shows the beginning and the end of all data transactions, that is, the data phase of every transaction. If a global master is somewhere not granted by a bus, you will see the name of that bus in the transaction trace view.

Creating a User-Defined Bus Analysis View

- Creating a Class
- Creating the init() and update() Functions
- Functions for Analysis Views
- Hooking Up the Custom Global and Custom Local Analysis Views
- Example of a Custom Bus Analysis View
- Specifying String and Color Mappings
- Example of a Custom Bus Analysis Trace View

Creating a Class

To write a custom bus analysis view, you have to create a class which is derived from the ***AMBATLMAnalysisView*** class, as shown below.

```
class viewName : public AMBATLMAnalysisView {
  private :

  public :
     viewName( AMBANode *aNode,
               const char *aName) :
               AMBATLMAnalysisView(aNode, aName)
  {
  }

  virtual void update() {

  }

  virtual void init() {

  }
};
```

The constructor of your custom view has to call the constructor of the ***AMBATLMAnalysisView*** class.

Creating the init() and update() Functions

You have to create the functions ***init()*** and ***update()***.

- In the ***init()*** function, you have to create all the analysis elements for your view. You can also specify how values will be mapped to strings to make your view more readable. Further, you can specify the colors that will be used to display the different events.
- In the ***update()*** function, you have to specify which values will be put into the analysis database and when they will be put into the database.

For this, you can use the predefined events and you can access all attributes of all transfers to collect data for your analysis view. This is explained in:

- Accessing Attributes of Transfers
- Predefined Events

Accessing Attributes of Transfers

The following example illustrates accessing an attribute of a transfer:

```
AMBATLMEventInfo SofT = pollTLMEvent(curr_master_id, ASofTInitiator);
int EotStatusVal = SofT.transaction->EotTrf.Status;
```

Predefined Events

To determine the moment at which data is sent to the analysis database, you have to use the predefined events. For trace views this will then also be used to determine the moment where actions are started or stopped.

For the output-stage bus, these predefined events are:

- ***ARequest*** is the moment at which the master has requested the bus, but is not granted the bus. So this means that ***ARequest*** happens when a data transaction should be started but the transaction is not started because the master is not granted the bus.
- ***AGrant*** is the moment at which the master starts a data transaction.
- ***ASofTInitiator*** is the moment at which the master starts a data transaction.
- ***ASofTTarget*** is the moment at which the data transaction starts at the target side.
- ***AEofTTarget*** is the moment at which the data transaction ends at the target side.

ARequest and ***AGrant*** are master-specific events. This means that these events will only happen when that particular master is accessing or trying to access the bus.

ASofTInitiator, ***ASofTTarget***, and ***AEofTTarget*** are bus-specific events. They happen when one of the masters and one of the slaves of that bus are interacting with each other. ***ASofTInitiator*** and ***ASofTTarget*** are exactly the same for the input-stage bus.

For the output-stage bus, the same is true as for the input-stage bus. The only difference is that ***ARequest*** and ***AGrant*** do not exist, since the bus only has one master by definition, which makes the request-grant mechanism obsolete.

Functions for Analysis Views

You can use the following functions to write your analysis view:

```
const vector<PeripheralInfo> &getAllInitiators() const;
```

```
const vector<PeripheralInfo> &getAllTargets() const;
```

```
string getInitiatorName(IDType initiatorID) const;
```

```
string getTargetName(IDType targetID) const;
```

```
const AMBAPort *getInitiator(IDType initiatorID) const;
```

```
const AMBAPort *getTarget(IDType targetID) const;
```

```
IDType getId(AMBAPort *port) const;
```

```
cwrBaseData *getAnalysisDataElement(IDType ID, string elemName);
```

cwrBaseData is the base class of ***cwrAnalysisData***. For more information on the ***cwrAnalysisData*** class, see “cwrAnalysisData Class” in the ***Analysis Manual***.

```
cwrBaseData *addAnalysisDataElement(IDType ID, const char *baseName,
                                    const char *name, DataElementType det,
                                cwrAnalysisView::PreProcessingOption preProc);
```

```
PreProcessingOption { MIN, MAX, COV, AVE, TAV, SUM, TRC, CNT, SPS, CPS };
```

For more information, see the “cwrAnalysisData Class” of the ***Analysis Manual***.

```
cwrBaseData *getAnalysisDataElement(IDType ID1, IDType ID2, string elemName);
cwrBaseData *addAnalysisDataElement(IDType ID1, IDType ID2,
                                    const char *baseName, const char *name,
                                    DataElementType det,
                                  cwrAnalysisView::PreProcessingOption preProc);
```

```
AMBATLMEventInfo pollTLMEvent (IDType nodeOrPeripheralID,
                               AMBAAnalysisEvent event);
```

pollTMEvent is of type ***AMBATLMEventInfo***, which has the following members:

- ***bool occurred;***
- ***AMBAPort* initiatorID;***
- ***AMBAPort* targetID;***
- ***AMBATransaction *transaction;***
- ***bool initiatorIsPeripheral;***
- ***bool targetIsPeripheral;***

```
void getDirtyTargets(AMBAAnalysisEvent event, set<IDType> &targetIDs);
```

A “dirty target” is a target which is currently busy with a transaction.

```
void getDirtyInitiators(AMBAAnalysisEvent event, set<IDType> &initIDs);
```

A “dirty initiator” is an initiator which is at this moment in time busy with a transaction.

Hooking Up the Custom Global and Custom Local Analysis Views

This custom analysis view should be written in a separate header file, which can have any name. You have to include this header file in the file that you created to hook up your peripherals to the platform dumped by Platform Creator. Further, you have to hook up your custom global analysis views and your custom local analysis views.

For example (in the ***system.cpp*** file):

```
// Hookup custom global view
MyGlobalView myGlobalView("custom_globalview");

// Hookup custom local views
MyInputStView myInputStView1(&bus.inputstage_node1, "custom_inputstview_1");
MyInputStView myInputStView2(&bus.inputstage_node2, "custom_inputstview_2");
MyInputStView myInputStView3(&bus.inputstage_node3, "custom_inputstview_3");
```

For the global analysis view, you just need to give this view a name. The local analysis views also need the bus to which they belong as parameter. You can find the name of the buses in the ***platform.h*** file generated by Platform Creator. These are the names you have specified in your Platform Creator run.

Example of a Custom Bus Analysis View

The example below is a custom bus analysis view that counts the number of continuous transactions. With the ***setAttribute*** command you can specify the default value for the attributes of the view. These do not need to be set, they can be set manually in System Designer when using the view, but by setting them in your code, they will have a good default value. For more information, see “UnderStanding the Basic Chart Types” in the ***Analysis Manual***.

CustomAnaViews.h contains:

```
#ifndef CUSTOMANAVIEW_H
#define CUSTOMANAVIEW_H

class CustomInputStView : public AMBATLMAnalysisView {
private :

public :
 CustomInputStView( AMBANode *aNode,
               const char *aName ) :
   AMBATLMAnalysisView(aNode, aName)
 {
   setAttribute( "view", "BAR" );
   setAttribute( "scalex", "1000000" );
 }

 virtual void update() {
   set<IDType> Active_Targets;
   getDirtyTargets(ASofTTarget, Active_Targets);
   getDirtyTargets(AEofTTarget, Active_Targets);

   set<IDType>::iterator b = Active_Targets.begin();
   set<IDType>::iterator e = Active_Targets.end();

   for (; b!=e; ++b) {
     AMBATLMEventInfo SofT = pollTLMEvent(*b, ASofTTarget);
     AMBATLMEventInfo EofT = pollTLMEvent(*b, AEofTTarget);

     if (EofT.occurred) {
       if ( SofT.occurred ) {
         if ( EofT.initiatorID == SofT.initiatorID ) {
           // still a continuous transaction
         } else {
           // transaction with a different master
           // so continuous transaction has ended and can be counted
           string n = "cont_transaction";
           cwrAnalysisData<long> *d =
             static_cast<cwrAnalysisData<long>*>(getAnalysisDataElement
                                 (EofT.initiatorID->analysisID, n));
           d->collect(1);
         }
       } else {
         // continuous transaction has ended and can be counted
         string n = "cont_transaction";
         cwrAnalysisData<long> *d =
          static_cast<cwrAnalysisData<long>*>(getAnalysisDataElement
                                 (EofT.initiatorID->analysisID, n));
         d->collect(1);
       }
     }
   }
 }


 virtual void init() {
   vector<PeripheralInfo>::const_iterator b = getAllInitiators().begin();
   vector<PeripheralInfo>::const_iterator e = getAllInitiators().end();

   for ( ; b != e; ++b ) {
     if ( (*b).port != 0 ) {
       string n = "cont_transaction_";
       n += (*b).name;
```

```
        // Create data element for this slave
        cwrBaseData *d = addAnalysisDataElement( (*b).id, "cont_transaction", n.c_str(), detLong,
                                                                   cwrAnalysisView::CNT );
      }
    }
  }

};
```

system.cpp contains:

```
#include <systemc.h>
#include "platform.h"
#include "Full_AHB _FRBM.h"
#include "Full_AHB_FRBS.h"
#include "AMBA/cwr_amba_default_slave.h"

#include "CustomInputStView.h"

int sc_main (int argc, char *argv[]) {

   // instantiate some sc_modules

  Full_FRBM<> master1("master1î);
  Full_FRBM<> master2("master2");
  Full_FRBM<> master3("master3");
  Full_FRBS_RW slave1("slave1");
  Full_FRBS_RW slave2("slave2");
  AHBLite_defslave defslave("defslave");

  // instantiate the sc_module that contains the bus simulators
  // this sc_module is generated by PCT.
  // The name of this module ("MyPlatform") is chosen by the PCT-user.
  MyPlatform bus("bus");

  sc_clock clk("Clock", 4, SC_NS, 0.5, 0, SC_NS, false);
  bus.clk(clk);
  master1.clk(clk);
  master2.clk(clk);
  master3.clk(clk);
  slave1.clk(clk);
  slave2.clk(clk);

  // hookup masters and slaves to the platform.
  //
  // the names "master1", "master2",... are the names of the ports
  // chosen by the PCT-user.

  master1.p(*bus.master1);
  master2.p(*bus.master2);
  master3.p(*bus.master3);
  defslave.p(*bus.defslave);
  slave1.p(*bus.slave1);
  slave2.p(*bus.slave2);

CustomInputStView CustomInputStView(&bus.inputstage_node, "ContTransCount");

  sc_start();

  return 0;
}
```

Specifying String and Color Mappings

Especially in trace views it can be very useful to do a mapping of the possible values to meaningful strings, like for example the name of the target that is being accessed.

To make your view more clear, you can specify the following:

- The string that will be shown for a specific analysis value
- The string that will appear in the pop-up window when you place your cursor above an analysis element
- The border color of the different analysis elements
- The fill color of the different analysis elements

When you want to specify one of the above items, you need to declare a ***cwrMapAttrib*** for it and you need to include ***cwrMapAttrib.h*** in the header file of your custom analysis view.

You have to use the ***insert(string `value`, string `result`)*** function to specify the mapping.

After you have specified all mappings, you have to set the attributes for the element you want to define. First, you have to use ***setAttribute*** to specify a string for you ***cwrMapAttrib***, then you need to specify for which attribute you want to use this map.

For example (fill color):

```
setAttribute("value2fillcolor_map", fillcolorMap);
setAttribute("fillcolor", "$PARAM_MAP(value2fillcolor_map, $DATAFIELD(value))");
```

You can set the following attributes:

- ***datalabel***
- ***popuplabel***
- ***fillcolor***
- ***bordercolor***

Example of a Custom Bus Analysis Trace View

For an example, see “Example of a Custom Bus Analysis Trace View” on page 90.

Chapter 6

APB Bus Model

This chapter describes the Advanced Peripheral Bus (APB) Bus Model.

APB Node Definition

The APB is used to interface low-bandwidth peripherals which do not require the high performance of a pipelined bus interface.

The AMBA Bus Library is able to simulate the connection between AHB initiator and APB target ports.

Note that the current implementation of the AMBA Bus Library does not allow you to connect an APB initiator port or another APB bus to the APB bus.

The AHB bus is instantiated in Platform Creator. For more information about this tool, see the ***Platform Creator User Manual***. Platform Creator then creates the source code for the bus topology which you described, based on the information you provided.

You can specify this information in Platform Creator as follows:

1. Select a specific protocol for each peripheral:
   - APB for target ports

   For more information, see “APB Protocols” on page 38.
2. Select the correct address and data width for the target ports.
3. Select the correct addressing mode for the target ports.
4. Create the connections between the initiator ports, target ports, and buses. For more information, see “Creating Connections” in the ***Platform Creator User Manual***.
5. Edit the memory map of your system. For more information, see “Memory Map” and “Using the GUI to Create Hardware Platforms” in the ***Platform Creator User Manual***.

   NOTE The APB node is seen as a single target by the node it is connected to. The space it occupies in the memory map will be of such a size that it is large enough to cover all target peripherals and the holes between them. For more information, see “Contiguous Range Buses” in the ***Platform Creator User Manual***.
6. Connect the clock.

You must use an APB node for each APB bus you want to create.

Preconditions

To be able to successfully generate an APB bus, all preconditions listed below must be met.

If there is a violation against one or more of these preconditions, error messages will be generated by Platform Creator during the specification of the bus.

The following preconditions must be fulfilled for the APB bus:

- At least one target port must be connected to the bus.
- One and only one initiator port must be connected to the bus.
- The protocol of each target port must be an APB protocol. For information on the available protocols, see “APB Protocols” on page 38. Both TLM and pin-accurate protocols are available.
- The width of the data terminals of the target ports must be ***8***, ***16***, or ***32*** bits.
- The initiator port must be one of the following:
  - An AHB bus
  - An AHB-lite bus
  - An output-stage bus
- Each of the target ports must be one of the following:
  - An APB target (a user-defined peripheral)

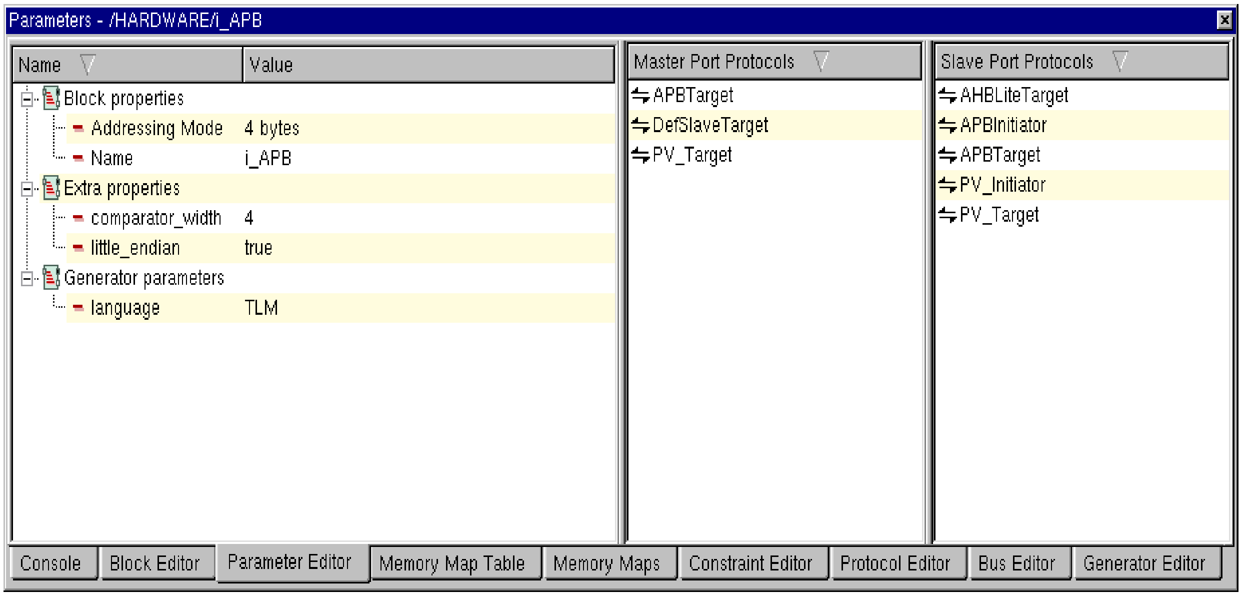
Setting Parameters

The following describes the parameters that can be set in Platform Creator. These parameters do not have an effect at the PV abstraction level.

- Specifying the Endianess
- Specifying the Comparator Width

Specifying the Endianess

In the parameters of the bus, you can specify whether the bus uses the little-endian or big-endian mode.



Changing the endianess has no influence when using only TLM peripherals, since this is abstracted away at the TLM abstraction level.

Specifying the Comparator Width

This parameter has no influence on the generated TLM bus. It is used by the AMBA RTL generator when the address decoder is built. For more information, see Chapter 10, “Generating an RTL Bus,” on page 105.

Analysis Views

The AMBA Bus Library enables views of the bus analysis library offered by System Designer. These analysis views enable you to get a complete overview of the strong and weak points of your design. The trace views can also be used to debug the design by monitoring the transactions between the masters and the slaves. If the standard analysis views do not give enough information, you can write your own analysis views. The procedure is described in “Creating a User-Defined Bus Analysis View” on page 84.

For information on the views of the bus analysis library, see “Bus Analysis Library” in the ***Analysis Manual***.

The following describes:

- Local Analysis Views for the APB Bus
- Global Analysis Views
- Creating a User-Defined Bus Analysis View

Local Analysis Views for the APB Bus

The following local analysis views are available for the AHB bus:

- Transaction count
- Transaction duration
- Transaction throughput
- Transaction trace
- Transaction utilization

The transaction trace view shows the beginning and the end of all data transactions, that is, the data phase of every transaction.

Global Analysis Views

The following global analysis views are available:

- Transaction count
- Transaction duration
- Transaction throughput
- Transaction trace
- Transaction utilization

The transaction trace view shows the beginning and the end of all data transactions, that is, the data phase of every transaction. If a global master is somewhere not granted by a bus, you will see the name of that bus in the transaction trace view.

Creating a User-Defined Bus Analysis View

- Creating a Class
- Creating the init() and update() Functions
- Functions for Analysis Views
- Hooking Up the Custom Global and Custom Local Analysis Views
- Example of a Custom Bus Analysis View
- Specifying String and Color Mappings
- Example of a Custom Bus Analysis Trace View

Creating a Class

To write a custom bus analysis view, you have to create a class which is derived from the ***AMBATLMAnalysisView*** class, as shown below.

```
class viewName : public AMBATLMAnalysisView {
  private :

  public :
     viewName( AMBANode *aNode,
              const char *aName) :
        AMBATLMAnalysisView(aNode, aName)
  {
  }

  virtual void update() {

  }

  virtual void init() {

  }
};
```

The constructor of your custom view has to call the constructor of the ***AMBATLMAnalysisView*** class.

Creating the init() and update() Functions

You have to create the functions ***init()*** and ***update()***.

- In the ***init()*** function, you have to create all the analysis elements for your view. You can also specify how values will be mapped to strings to make your view more readable. Further, you can specify the colors that will be used to display the different events.
- In the ***update()*** function, you have to specify which values will be put into the analysis database and when they will be put into the database.

For this, you can use the predefined events and you can access all attributes of all transfers to collect data for your analysis view. This is explained in:

- Accessing Attributes of Transfers
- Predefined Events

Accessing Attributes of Transfers

The following example illustrates accessing an attribute of a transfer:

```
AMBATLMEventInfo SofT = pollTLMEvent(curr_master_id, ASofTInitiator);
int BurstGroupVal = SofT.transaction->AddrTrf.Group;
```

Predefined Events

To determine the moment at which data is sent to the analysis database, you have to use the predefined events. For trace views this will then also be used to determine the moment where actions are started or stopped.

For the APB bus, these predefined events are:

- ***ASofTInitiator*** is the moment at which the master starts a data transaction.
- ***ASofTTarget*** is the moment at which the data transaction starts at the target side.
- ***AEofTTarget*** is the moment at which the data transaction ends at the target side.

ASofTInitiator, ***ASofTTarget***, and ***AEofTTarget*** are bus-specific events. They happen when one of the masters and one of the slaves of that bus are interacting with each other. ***ASofTInitiator*** and ***ASofTTarget*** are exactly the same for the APB bus.

Functions for Analysis Views

You can use the following functions to write your analysis view:

```
const vector<PeripheralInfo> &getAllInitiators() const;
```

The ***getAllInitiators()*** function returns a vector with as many entries as there are peripherals (so both initiators and targets), but only the initiators are filled in. So to check whether an entry is an initiator or not, see if ***vectorelement.port != 0***.

```
const vector<PeripheralInfo> &getAllTargets() const;
```

The ***getAllTargets()*** function returns a vector with as many entries as there are peripherals (so both initiators and targets), but only the targets are filled in. So to check whether an entry is a target or not, see if ***vectorelement.port != 0***.

```
string getInitiatorName(IDType initiatorID) const;
```

```
string getTargetName(IDType targetID) const;
```

```
const AMBAPort *getInitiator(IDType initiatorID) const;
```

```
const AMBAPort *getTarget(IDType targetID) const;
```

```
IDType getId(AMBAPort *port) const;
```

```
cwrBaseData *getAnalysisDataElement(IDType ID, string elemName);
```

cwrBaseData is the base class of ***cwrAnalysisData***. For more information on the ***cwrAnalysisData*** class, see “cwrAnalysisData Class” in the ***Analysis Manual***.

```
cwrBaseData *addAnalysisDataElement(IDType ID, const char *baseName,
                                    const char *name, DataElementType det,
                                cwrAnalysisView::PreProcessingOption preProc);
```

```
PreProcessingOption { MIN, MAX, COV, AVE, TAV, SUM, TRC, CNT, SPS, CPS };
```

For more information, see “cwrAnalysisData Class” in the ***Analysis Manual***.

```
cwrBaseData *getAnalysisDataElement(IDType ID1, IDType ID2, string elemName);

cwrBaseData *addAnalysisDataElement(IDType ID1, IDType ID2,
                                    const char *baseName, const char *name,
                                    DataElementType det,
                                cwrAnalysisView::PreProcessingOption preProc);

AMBATLMEventInfo pollTLMEvent (IDType nodeOrPeripheralID,
                               AMBAAnalysisEvent event);
```

pollTMEvent is of type ***AMBATLMEventInfo***, which has the following members:

- ***bool occurred;***
- ***AMBAPort* initiatorID;***
- ***AMBAPort* targetID;***
- ***AMBATransaction *transaction;***
- ***bool initiatorIsPeripheral;***
- ***bool targetIsPeripheral;***

```
void getDirtyTargets(AMBAAnalysisEvent event, set<IDType> &targetIDs);
```

A “dirty target” is a target which is currently busy with a transaction.

```
void getDirtyInitiators(AMBAAnalysisEvent event, set<IDType> &initIDs);
```

A “dirty initiator” is an initiator which is at this moment in time busy with a transaction.

Hooking Up the Custom Global and Custom Local Analysis Views

This custom analysis view should be written in a separate header file, which can have any name. You have to include this header file in the file that you created to hook up your peripherals to the platform dumped by Platform Creator. Further, you have to hook up your custom global analysis views and your custom local analysis views.

For example (in the ***system.cpp*** file):

```
// Hookup custom global view
MyGlobalView myGlobalView("custom_globalview");

// Hookup custom local views
MyAPBView myAPBView1(&bus.apb_node1, "custom_apbview_1");
MyAPBView myAPBView2(&bus.apb_node2, "custom_apbview_2");
MyAPBView myAPBView3(&bus.apb_node3, "custom_apbview_3");
```

For the global analysis view, you just need to give this view a name. The local analysis views also need the bus to which they belong as parameter. You can find the name of the buses in the ***platform.h*** file generated by Platform Creator. These are the names you have specified in your Platform Creator run.

Example of a Custom Bus Analysis View

The example below is a custom bus analysis view that counts the number of continuous transactions. With the ***setAttribute*** command you can specify the default value for the attributes of the view. These do not need to be set, they can be set manually in System Designer when using the view, but by setting them in your code, they will have a good default value. For more information, see “UnderStanding the Basic Chart Types” in the ***Analysis Manual***.

CustomAnaViews.h contains:

```
#ifndef CUSTOMANAVIEW_H
#define CUSTOMANAVIEW_H

class CustomAPBView : public AMBATLMAnalysisView {
private :

public :
 CustomAPBView( AMBANode *aNode,
                const char *aName) :
   AMBATLMAnalysisView(aNode, aName )
 {
   setAttribute( "view", "BAR" );
   setAttribute( "scalex", "1000000" );
 }

 virtual void update() {
   set<IDType> Active_Targets;
   getDirtyTargets(ASofTTarget, Active_Targets);
   getDirtyTargets(AEofTTarget, Active_Targets);

   set<IDType>::iterator b = Active_Targets.begin();
   set<IDType>::iterator e = Active_Targets.end();

   for (; b!=e; ++b) {
     AMBATLMEventInfo SofT = pollTLMEvent(*b, ASofTTarget);
     AMBATLMEventInfo EofT = pollTLMEvent(*b, AEofTTarget);

     if (EofT.occurred) {
       if ( SofT.occurred ) {
         if ( EofT.initiatorID == SofT.initiatorID ) {
           // still a continuous transaction
         } else {
           // transaction with a different master
           // so continuous transaction has ended and can be counted
           string n = "cont_transaction";
           cwrAnalysisData<long> *d =
             static_cast<cwrAnalysisData<long>*>(getAnalysisDataElement
                                     (EofT.initiatorID->analysisID, n));
           d->collect(1);
         }
       } else {
         // continuous transaction has ended and can be counted
         string n = "cont_transaction";
         cwrAnalysisData<long> *d =
          static_cast<cwrAnalysisData<long>*>(getAnalysisDataElement
                               (EofT.initiatorID->analysisID, n));
         d->collect(1);
       }
     }
   }
 }

 virtual void init() {
   vector<PeripheralInfo>::const_iterator b = getAllInitiators().begin();
   vector<PeripheralInfo>::const_iterator e = getAllInitiators().end();
```

```
    for ( ; b != e; ++b ) {
      if ( (*b).port != 0 ) {
        string n = "cont_transaction_";
        n += (*b).name;

        // Create data element for this slave
        cwrBaseData *d = addAnalysisDataElement( (*b).id, "cont_transaction", n.c_str(), detLong,
                                                               cwrAnalysisView::CNT );
      }
    }
  }

};

#endif
```

system.cpp contains:

```
#include <systemc.h>
#include "platform.h"
#include "Full_AHB_FRBM.hî
#include "APB _FRBS.h"
#include "AMBA/cwr_amba_default_slave.h"

#include "CustomAPBView.h"

int sc_main (int argc, char *argv[]) {

    // instantiate some sc_modules

  Full_FRBM<> master1("master1î);
  Full_FRBM<> master2("master2");
  Full_FRBM<> master3("master3");
  APB_FRBS_RW slave1("slave1");
  APB_FRBS_RW slave2("slave2");
  AHBLite_defslave defslave("defslave");

  // instantiate the sc_module that contains the bus simulators
  // this sc_module is generated by PCT.
  // The name of this module ("MyPlatform") is chosen by the Platform
  // Creator user.
  MyPlatform bus("bus");

  sc_clock clk("Clock", 4, SC_NS, 0.5, 0, SC_NS, false);
  bus.clk(clk);
  master1.clk(clk);
  master2.clk(clk);
  master3.clk(clk);
  slave1.clk(clk);
  slave2.clk(clk);

  // hookup masters and slaves to the platform.
  //
  // the names "master1", "master2",... are the names of the ports
  // chosen by the PCT-user.

  master1.p(*bus.master1);
  master2.p(*bus.master2);
  master3.p(*bus.master3);
  defslave.p(*bus.defslave);
  slave1.p(*bus.slave1);
  slave2.p(*bus.slave2);

  CustomAPBView CustomAPBView(&bus.apb_node, "ContTransCount");

  sc_start();

  return 0;
}
```

Specifying String and Color Mappings

Especially in trace views it can be very useful to do a mapping of the possible values to meaningful strings, like for example the name of the target that is being accessed.

To make your view more clear, you can specify the following:

- The string that will be shown for a specific analysis value
- The string that will appear in the pop-up window when you place your cursor above an analysis element
- The border color of the different analysis elements
- The fill color of the different analysis elements

When you want to specify one of the above items, you need to declare a ***cwrMapAttrib*** for it and you need to include ***cwrMapAttrib.h*** in the header file of your custom analysis view.

You have to use the ***insert(string `value`, string `result`)*** function to specify the mapping.

After you have specified all mappings, you have to set the attributes for the element you want to define. First, you have to use ***setAttribute*** to specify a string for you ***cwrMapAttrib***, then you need to specify for which attribute you want to use this map.

For example (fill color):

```
setAttribute("value2fillcolor_map", fillcolorMap);
setAttribute("fillcolor", $PARAM_MAP(value2fillcolor_map,$DATAFIELD(value))");
```

You can set the following attributes:

- ***datalabel***
- ***popuplabel***
- ***fillcolor***
- ***bordercolor***

Example of a Custom Bus Analysis Trace View

For an example, see "Example of a Custom Bus Analysis Trace View" on page 60.

Chapter 7

Lite2AHB Bridge

This chapter describes the ***Lite2AHB*** bridge.

Lite2AHB Bridge Definition

The ***Lite2AHB*** bridge enables the designer to attach an AHB-lite initiator port to a full AHB system. The ***Lite2AHB*** bridge will take care of generating the request signal needed for arbitration. When the AHB-lite initiator is attempting to perform an access while the bridge is not granted, the bridge will assert the request and buffer the current AHB-lite access until it becomes granted.

For more detailed explanation on the ***Lite2AHB*** bridge, see the ARM ADK documentation.

Lite2AHB Bridge Preconditions

To be able to successfully generate the ***Lite2AHB*** bridge, all preconditions listed below must be met.

- It is not possible to attach the ***Lite2AHB*** bridge as a target of another node. The initiator port connected to the ***Lite2AHB*** bridge must be a user block.
- Only one AHB-lite initiator can be connected to the bridge.
- The protocol of the initiator must be ***AHBLiteInitiator***.
- The target connected to the ***Lite2AHB*** bridge must be an AHB node.
- Only one AHB node can act as a target for the bridge.

NOTE The ***Lite2AHB*** bridge currently does not support split/retry responses.

Setting Parameters

The following describes the parameters that can be set in Platform Creator. These parameters do not have an effect at the PV abstraction level.

For more information on specifying parameters of nodes and bridges, see the ***Platform Creator User Manual***.

- Specifying Block Properties
- Specifying Extra Properties

Specifying Block Properties

The ***Lite2AHB*** bridge has no block properties.

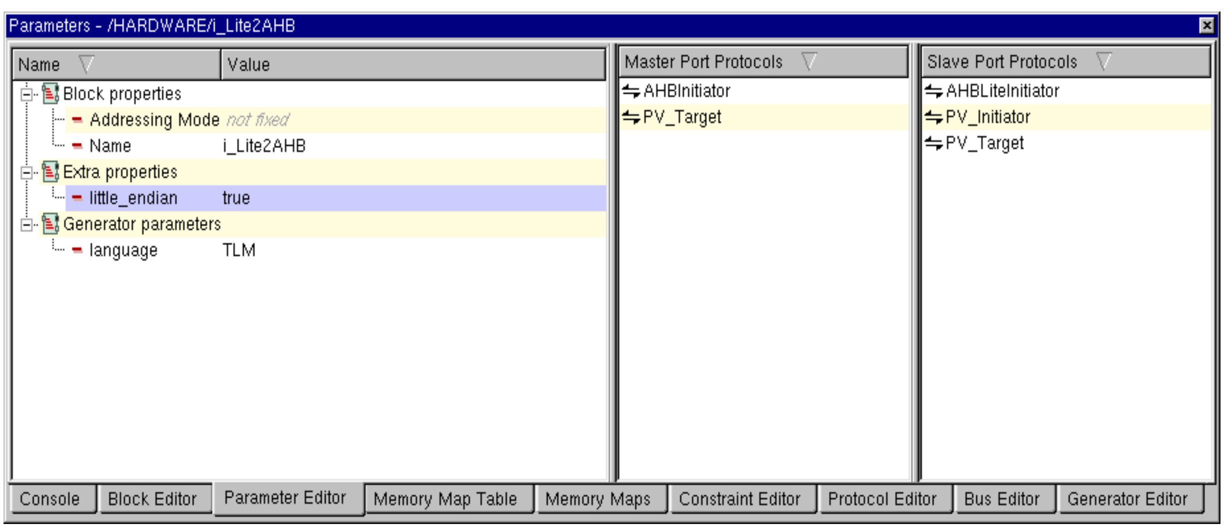
Specifying Extra Properties

- Specifying the Endianess

Specifying the Endianess

You have to specify whether the node uses the little-endian or big-endian mode. This is important when a pin-accurate peripheral is connected to the node. At the TLM abstraction level however, endianess is abstracted away.

To specify the little-endian or big-endian mode, enable or disable ***little_endian*** in the Parameters Panel of the Platform Creator GUI, as shown below.



Analysis Views

The AMBA Bus Library enables views of the bus analysis library offered by System Designer. These analysis views enable you to get a complete overview of the strong and weak points of your design. The trace views can also be used to debug the design by monitoring the transactions between the masters and the slaves. If the standard analysis views do not give enough information, you can write your own analysis views. The procedure is described in "Creating a User-Defined Bus Analysis view" on page 93.

For information on the views of the bus analysis library, see "Bus Analysis Library" in the ***Analysis Manual***.

The following describes:

- Local Analysis Views for the AHBLite2AHB Bridge
- Global Analysis Views for the AHBLite2AHB Bridge
- Creating a User-Defined Bus Analysis view

Local Analysis Views for the AHBLite2AHB Bridge

The following local analysis views are available for the AHB-lite bus:

- Transaction count
- Transaction duration
- Transaction throughput
- Transaction trace
- Transaction utilization

The transaction trace view shows the beginning and the end of all data transactions, that is, the data phase of every transaction.

Global Analysis Views for the AHBLite2AHB Bridge

The following global analysis views are available:

- Transaction count
- Transaction duration
- Transaction throughput
- Transaction trace
- Transaction utilization

The transaction trace view shows the beginning and the end of all data transactions, that is, the data phase of every transaction. If a global master is somewhere not granted by a bus, you will see the name of that bus in the transaction trace view.

Creating a User-Defined Bus Analysis view

To learn how to create a user-defined analysis view for the ***Lite2AHB*** bridge, see "Creating a User-Defined Bus Analysis View" on page 54. The predefined analysis events are the same as for the AHB-lite node.

Chapter 8

Downsizer Bridge

This chapter describes the ***Downsizer*** bridge.

Downsizer Bridge Definition

The ***Downsizer*** bridge enables the designer to connect a 32-bit-wide peripheral or bus to a 64-bit-wide bus, and perform 64-bit accesses to it. The ***Downsizer*** bridge will take care of splitting the 64-bit access into two 32-bit accesses. 32-bit accesses originating from the 64-bit bus will pass trough the bridge unmodified.

For detailed information on the functionality of the ***Downsizer*** bridge, see the ***AMBA Extensions Infrastructure Blocks*** (ARM DDI 0283A).

NOTE The ***Downsizer*** bridge does not support split/retry responses.

Downsizer Bridge Preconditions

To be able to successfully generate the ***Downsizer*** bridge, all preconditions listed below must be met.

- The initiator connected to the ***Downsizer*** bridge must be a 64-bit-wide AHB-lite, AHB, or output-stage node.
- One and only one initiator can be connected to the ***Downsizer*** bridge.
- The target connected to the ***Downsizer*** bridge can be one of the following:
  - An APB node
  - An AHB-lite node
  - An input-stage node
  - An AHB target peripheral
  - An AHB-lite target peripheral
- One and only one target can be connected to the ***Downsizer*** bridge.
- The width of the data terminals of the initiator must be 64 bits.
- The width of the data terminals of the target must be 32 bits.
- The protocol of the initiator connected to the bridge must be ***AHBLiteTarget***.

- The protocol of the target connected to the bridge must be ***AHBLiteTarget*** or ***AHBTarget***.

Setting Parameters

The following describes the parameters that can be set in Platform Creator. These parameters do not have an effect at the PV abstraction level.

For more information on specifying parameters of nodes and bridges, see the ***Platform Creator User Manual***.

- Specifying Block Properties
- Specifying Extra Properties

Specifying Block Properties

The ***Downsizer*** bridge has no block properties.

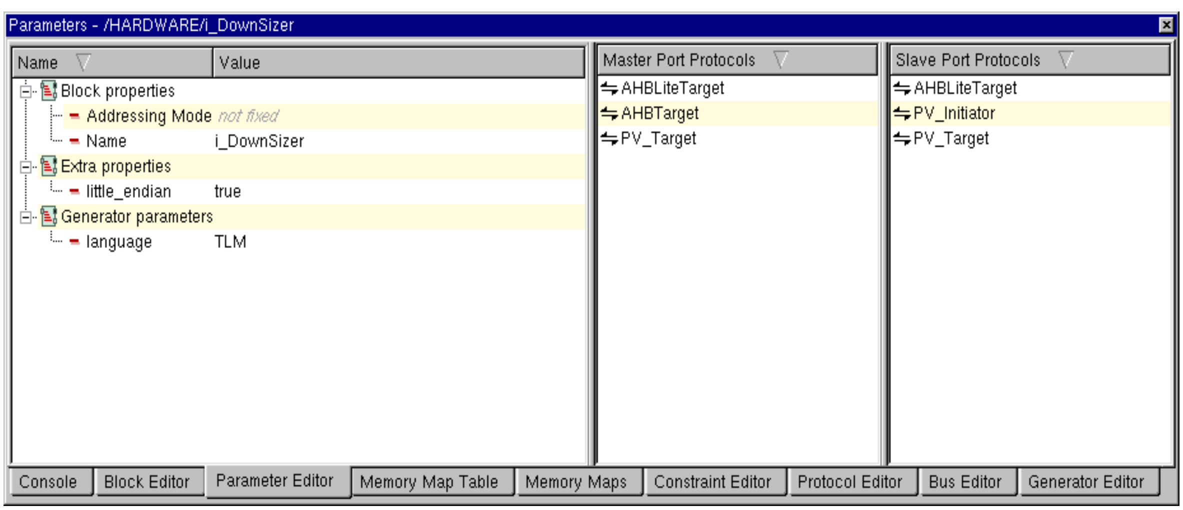
Specifying Extra Properties

- Specifying the Endianess

Specifying the Endianess

You have to specify whether the node uses the little-endian or big-endian mode. This is important when a pin-accurate peripheral is connected to the node. At the TLM abstraction level however, endianess is abstracted away.

To specify the little-endian or big-endian mode, enable or disable ***little_endian*** in the Parameters Panel of the Platform Creator GUI, as shown below.



Analysis views

The AMBA Bus Library enables views of the bus analysis library offered by System Designer. These analysis views enable you to get a complete overview of the strong and weak points of your design. The trace views can also be used to debug the design by monitoring the transactions between the masters and the slaves. If the standard analysis views do not give enough information, you can write your own analysis views. The procedure is described in "Creating a User-Defined Bus Analysis view" on page 93.

For information on the views of the bus analysis library, see "Bus Analysis Library" in the ***Analysis Manual***.

The following describes:

- Local Analysis Views for the Downsizer Bridge
- Global Analysis Views for the Downsizer Bridge
- Creating a User-Defined Bus Analysis view

Local Analysis Views for the Downsizer Bridge

The following local analysis views are available for the ***Downsizer*** bridge:

- Transaction count
- Transaction duration
- Transaction throughput
- Transaction trace
- Transaction utilization

The transaction trace view shows the beginning and the end of all data transactions, that is, the data phase of every transaction.

NOTE The analysis data is based on the downsized (32-bit-wide) side of the bridge.

Global Analysis Views for the Downsizer Bridge

If a ***DownSizer*** bridge is used in your design, global analysis will not work.

Creating a User-Defined Bus Analysis view

To learn how to create a user-defined analysis view for the ***Downsizer*** bridge, see "Creating a User-Defined Bus Analysis View" on page 54. The predefined analysis events are the same as for the AHB-lite node.

Chapter 9

AHB2Lite Block

This chapter describes the ***AHB2Lite*** block.

AHB2Lite Block Definition

The ***AHB2Lite*** block is actually one half of the ***AHB2AHB*** bridge. This ***AHB2AHB*** bridge is used to connect two AHB buses. It converts an ***AHBLiteTarget*** protocol to an ***AHBInitiator*** protocol. In the AMBA Bus Library, like in the ARM reference implementation, the ***AHB2AHB*** bridge consists of two parts: the ***AHB2Lite*** block and the ***Lite2AHB*** bridge. So in fact, the ***AHB2Lite*** block is necessary to connect the target side of an AHB node with the initiator side of a ***Lite2AHB*** bridge. The ***AHB2Lite*** bridge supports address translation. For more information see "Doing Address Translation within the AHB2Lite block" on page 100.

AHB2Lite Block Preconditions

To successfully use the ***AHB2Lite*** block, the following preconditions have to be met:

- The ***AHB2Lite*** block has to be connected as an AHB-lite target to an AHB or AHB-lite node.
- The ***AHB2Lite*** block has to be connected as an AHB-lite initiator to a ***Lite2AHB*** bridge.

NOTE The ***AHB2Lite*** block currently does not support split/retry responses.

NOTE The ***AHB2Lite*** block is only available at the TLM abstraction level. You can, however, switch this ***AHB2Lite*** TLM block with an equivalent RTL block if available.

Doing Address Translation within the AHB2Lite block

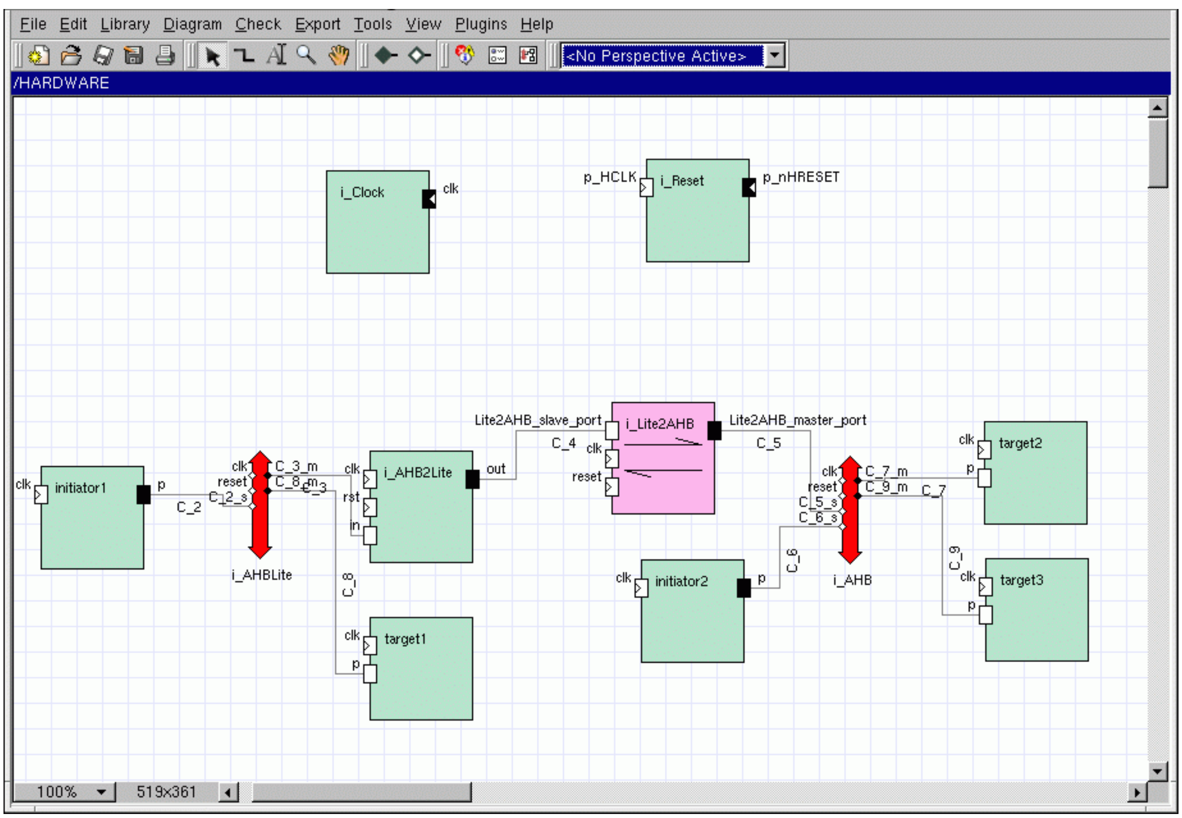
The ***AHB2Lite*** block offers the possibility to translate the incoming address into a different outgoing address. When an address is specified in a file, the name of which needs to be specified as a constructor argument for the ***AHB2Lite*** block (see section "Specifying Constructor Arguments" on page 103), the logical operator ***OR*** will be executed on this address and the incoming address to form the outgoing address:

```
outgoing address = incoming address | address from input file;
```

The address needs to be specified in hexadecimal numbers, but without ***0x*** in front of it, for example, ***A0000000***.

Since the ***AHB2Lite*** block performs address translation, the address map will look different for the blocks that can be reached by the initiator side of the ***AHB2Lite*** block. For this reason, both memory ports of the ***AHB2Lite*** block have to be put in the memory map.

For example, suppose you have a system with two initiators, each connected to their own AHB node, and a couple of targets. And that one initiator also needs to have access to the targets connected to the other initiators AHB node. For this you need to use an ***AHB2AHB*** bridge. The system looks as follows in Platform Creator:



Suppose you want to do the following address translation: always put the 31st bit of the address to 1. You need to write address ***80000000*** in the input file. You give the name of this input file as constructor argument to the ***AHB2Lite*** block.

initiator2 can access two targets:

- ***target2*** at address ***0x80000000***
- ***target3*** at address ***0xe0000000***

initiator1 can access one target without the ***AHB2AHB*** bridge:

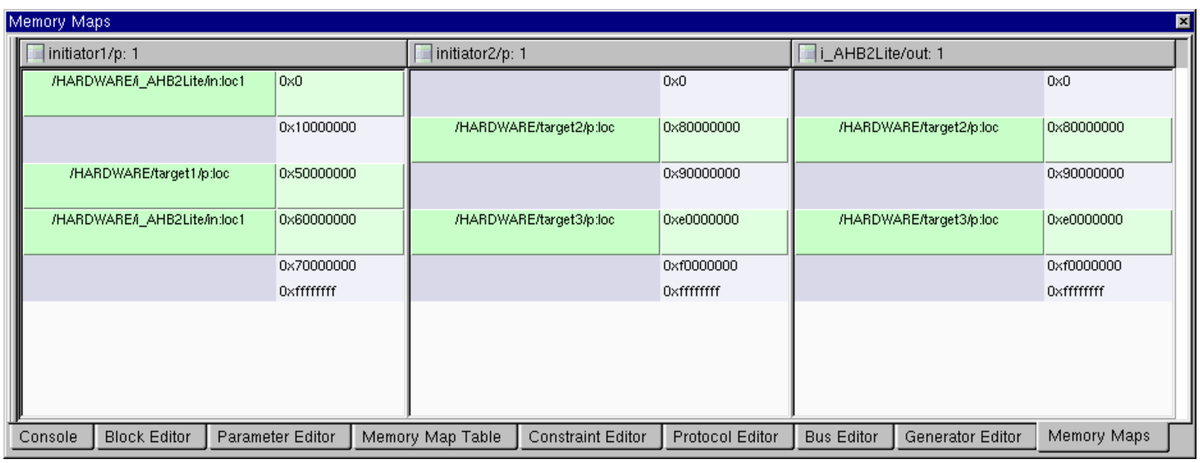
- ***target1*** at address ***0x50000000***

initiator1 can access two targets through the ***AHB2AHB*** bridge. Since the ***AHB2Lite*** block does address translation, it sets bit 31 to ***1*** when going from the incoming to the outgoing address. ***initiator1*** will see the two targets on the address with bit 31 still ***0***:

- ***target2*** at address ***0x00000000***
- ***target3*** at address ***0x60000000***

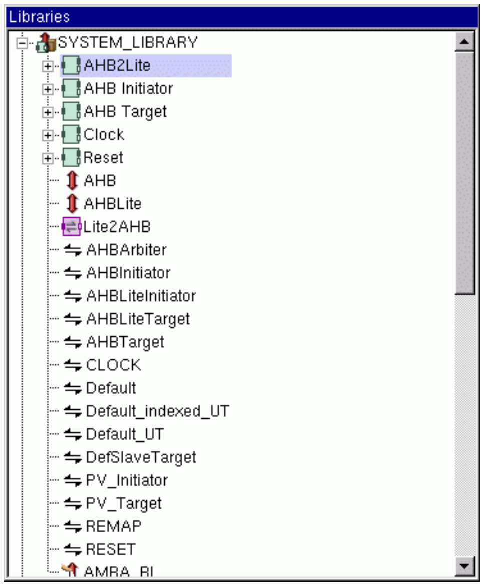
In fact, the two targets will not show up in the memory map of ***initiator1***, the initiator sees the ***AHB2Lite*** block as target, with two regions. The ***AHB2Lite*** block is an initiator or the second subsystem and actually sees ***target2*** at address ***0x80000000*** and ***target3*** at address ***0xe0000000***.

The following figure shows the memory map.



You are also allowed to specify no input file as constructor argument or you can use address ***0*** for the address translation. In that case, the ***AHB2Lite*** block actually acts as a bridge that is transparent for the incoming and outgoing addresses. To specify to Platform Creator that the ***AHB2Lite*** block acts as a bridge and is transparent for the memory map:

1 Right-click on the ***AHB2Lite*** block in the ***SYSTEM_LIBRARY*** section within the Library Drawer of Platform Creator (left subwindow of the Platform Creator window), as shown below.



2 From the pop-up menu, select ***Convert to Bridge***. From this moment on, the ports of the ***AHB2Lite*** bridge do not have to be put in the memory map any more. For more information, see the ***Platform Creator User Manual***.

Once you converted the ***AHB2Lite*** block to a bridge, the block will always be seen as a bridge by Platform Creator. So if there is at least one place in your system where you want to do address translation with an ***AHB2Lite*** block, you should not convert the ***AHB2Lite*** block to a bridge!

Setting Parameters

The following describes the parameters that can be set in Platform Creator.

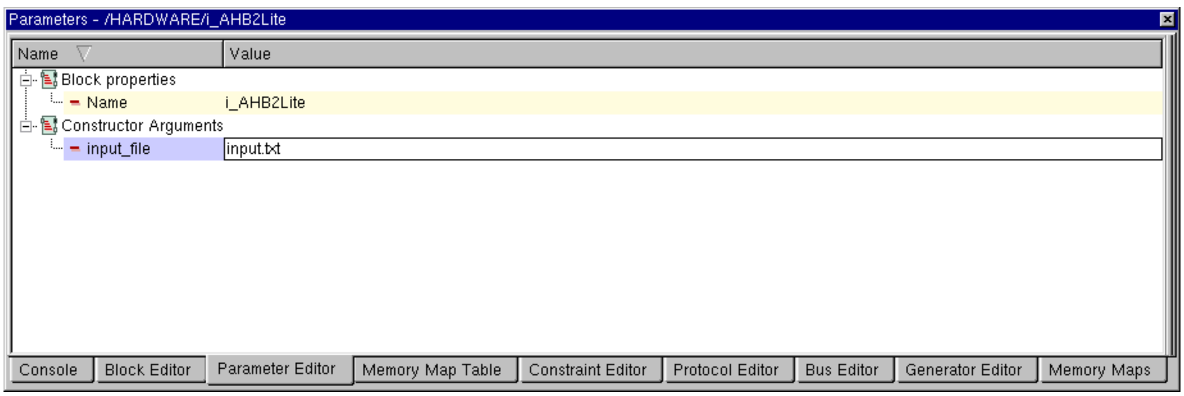
For more information on specifying parameters of nodes and bridges, see the ***Platform Creator User Manual***.

- Specifying Constructor Arguments

Specifying Constructor Arguments

The ***AHB2Lite*** block has one constructor argument: ***input_file***. You have to specify the name of the file in which you have put the address translation that has to be done by the ***AHB2Lite*** block. The specified file has to be located in the directory where you started Platform Creator.

The following figure shows the ***input_file*** constructor argument.



You are allowed to specify no file name. In that case the ***AHB2Lite*** block will not do any address translation. For more information, see“Doing Address Translation within the AHB2Lite block” on page 100.

Analysis Views

The ***AHB2Lite*** block does not provide any analysis views. From an analysis point of view, the block divides your system into two subsystems, so the global analysis data will not cross the boundary of the ***AHB2Lite*** block.

Chapter 10

Generating an RTL Bus

If all the connections between the created bus and other system components have a pin-accurate interface, the complete bus architecture can be generated in VHDL or Verilog by means of the AMBA Register-Transfer-Level (RTL) generator. To achieve this, the bus generator language must be set to VHDL or Verilog rather than the default TLM. The RTL generation settings and commands can be found in the Platform Creator ***Tools*** menu. For more information, see the ***Platform Creator User Manual***.

The RTL bus generation is executed in Platform Creator during system export, hardware export, or HDL export.

The system export and hardware export commands will generate the bus architecture in RTL and create a SystemC wrapper to enable co-simulation with other pin-accurate SystemC components. These can be HDL blocks wrapped in HDL SystemC foreign modules or pin-accurate SystemC blocks.

The HDL export, on the other hand, requires all the system components to be modeled as HDL blocks wrapped in HDL SystemC foreign modules. The SystemC infrastructure will be removed during the export and a pure HDL system will be created. Only single-language HDL export is supported; mixing of VHDL and Verilog is not allowed.

This chapter describes the generation of the ARM AMBA 2.0 bus for each bus node.

- AHB Node RTL Generator
- APB Node RTL Generator
- Input-Stage and Output-Stage Node RTL Generator
- Lite2AHB Bridge RTL Generator
- DownSizer Bridge RTL Generator

AHB Node RTL Generator

The RTL generator generates the required AHB blocks (see the ***AMBA Specification*** (Rev 2.0)), depending on the number of initiators and targets, the protocols (the actual pin mappings) that are used to connect the peripherals, and the parameters specified in the Platform Creator GUI.

The AHB node can be composed of the following blocks:

- AHB Arbiter
- AHB Decoder
- AHB Master-to-Slave Multiplexer
- AHB Slave-to-Master Multiplexer
- Lite-to-AHB Wrapper
- AHB Default Slave

- Dummy Master

For a description of the AHB data width, see “AHB Data Width” on page 107.

AHB Arbiter

The bus arbiter ensures that only one bus initiator at a time is allowed to initiate data transfers. You can choose between a fixed priority and a round-robin arbitration scheme. In the case of a fixed priority arbitration scheme, the AHB generator generates an AHB arbiter based on ***ddi0243A-01_adk-trm***. The arbiter with round-robin arbitration scheme has the same basic structure with modifications to the actual arbitration scheme.

The generator looks at the number of initiators to determine whether or not an arbiter is necessary. From the moment more than one initiator is connected to the node, the generator creates an arbitration block. The complexity of the generated arbiter depends on the signals used by the connected initiators and targets. If none of the targets supports ***Split*** accesses, the generated arbiter does not support ***Split***. The same is true for burst and locked transfers from the initiator side. If it is not supported by any initiator, the generator does not create logic for it.

AHB Decoder

The AHB decoder is used to decode the address of each transfer and provides a select signal for the target that is involved in the transfer. The generator generates a decoder, which minimizes the necessary logic.

For logic optimization purposes, the decoding of the address is split into small comparators (switches). The ***comparator_width*** parameter (see “Specifying the Comparator Width” on page 49) allows you to specify how many address lines are compared in one switch. This enables you to target the generated address decoder towards your technology library.

AHB Master-to-Slave Multiplexer

The master-to-slave multiplexer (***MuxM2S***) multiplexes the AHB initiator output signals that are involved in the transfer. If only one initiator is connected to the node, the generator generates a simplified block called ***Con_M2S*** (connection master to slave).

AHB Slave-to-Master Multiplexer

The slave-to-master multiplexer (***MuxS2M***) multiplexes the AHB target output signals that are involved in the transfer. If only one target is connected to the node, the generator generates a simplified block called ***Con_S2M*** (connection slave to master).

Lite-to-AHB Wrapper

The lite-to-AHB wrapper (***Lite2AHB***) converts an AHB-lite interface into a full AHB interface. If a lite-to-AHB wrapper is necessary in the generated system, the AHB generator generates the lite-to-AHB wrapper of the ***AHB File Reader Bus Master Technical Reference Manual***. The generator decides whether a lite-to-AHB wrapper is necessary based on the AHB protocols that are used by the AHB blocks, which in turn are connected to the node.

AHB Default Slave

The AHB default slave responds to transfers that are made to undefined regions of memory, where no AHB slaves are mapped. If a default slave is necessary in the generated system, the AHB generator generates the default slave of the ***AHB Example AMBA System Technical Reference Manual***. It is not necessary to have a default slave in every node, because the default slave in the node directly connected to the AHB master catches all possible transfers which the initiator tries to make to undefined memory regions. With this rule as basis, the generator determines whether or not a default slave is required. However, you can overrule the rule through a node parameter in Platform Creator.

Dummy Master

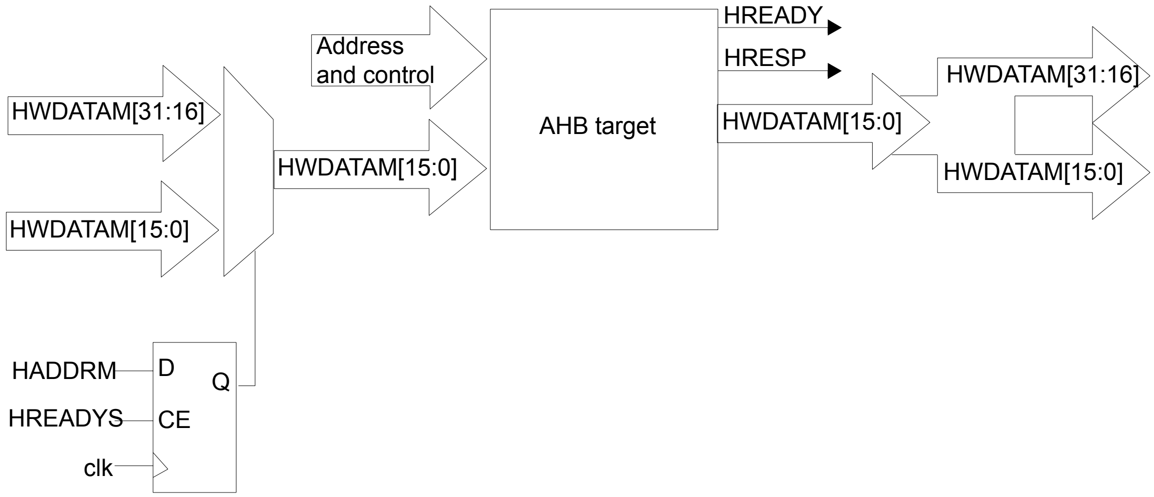
If several initiators are connected to the node and one of the connected targets is able to do ***Split*** accesses, a ***Split***-capable arbiter is generated. This ***Split***-capable arbiter needs a dummy master. This is not really a separate initiator or a separate block, it is simply a part of the master-to-slave multiplexer (***MuxM2S***). This dummy master is an internal initiator, which is granted the bus when all the real initiators are being split by the peripherals. It enables the bus to remain synchronized when no real initiators are granted the bus. When the dummy master is granted and ***HMASTER*** is ***0*** (the dummy master is set as initiator 0), the ***HTRANS*** signal is set to ***IDLE*** in the master-to-slave multiplexer.

AHB Data Width

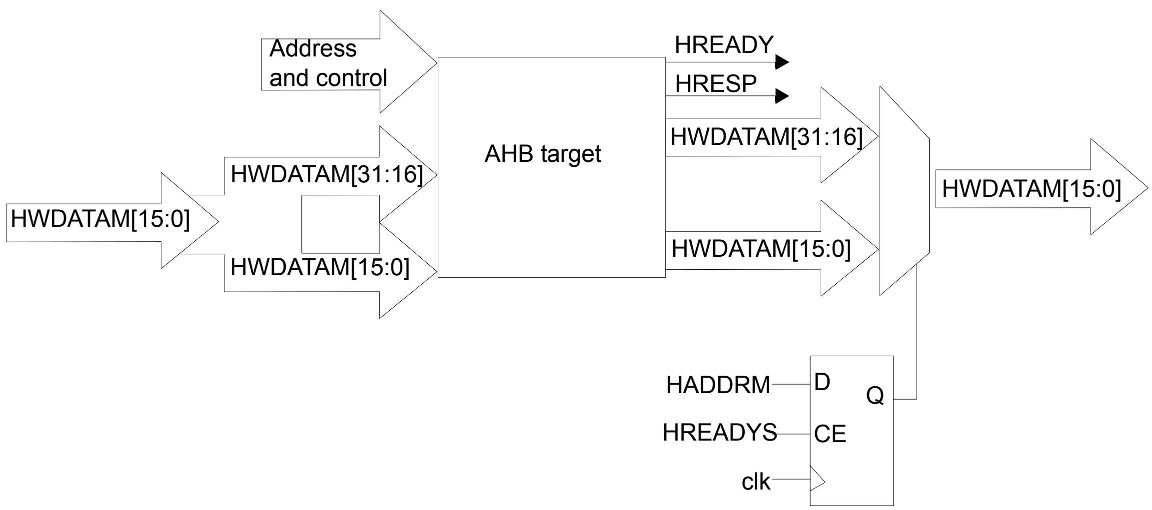
The data width of the bus is equal to the largest data width of the initiators. This means that, in the case of an initiator with a data width of 16 bits and an initiator with a data width of 32 bits, the generated bus has a data width of 32 bits. The reason is that AHB initiators cannot be made to work on a narrow bus. It is, however easy to make a bus initiator to work on a wide bus: the data of the bus initiator is then replicated onto the different parts of the data signal of the bus.

If a target gets connected to a bus of which the data width differs with its own data width, logic is provided by the generator to convert the data to the right size.

The following figure illustrates implementing a narrow target on a wider bus.



The following figure illustrates implementing a wide target on a narrow bus.



As shown in the figures above, when a narrow data signal must be converted to a wider data signal, the data is just replicated. And when a wide data signal needs to be converted to a narrow data signal, the data gets multiplexed to select the right byte lanes.

The following describes how the byte lanes of the ***HWDATA*** signal and the ***HRDATA*** signal for a data width of up to 32 bits are multiplexed, respectively. For data widths bigger than 32 bits, an equivalent multiplexing of the byte lanes is valid.

- HWDATA Byte Lanes
- HRDATA Byte Lanes

HWDATA Byte Lanes

The following tables show how the byte lanes of the ***HWDATA*** signal are multiplexed. In these tables, ***x*** indicates the unconnected data bytes and ***C*** indicates the data bytes that get connected from the active initiator to the active target (the actual valid data bytes depend on the transfer size).

- Initiator: 32-bit data width
  Target: 16-bit data width

reg_HADDRM[1:0]	HWDATA[31:0] Little endian	HWDATA[31:0] Big endian
0x0	xxCC	CCxx
0x1	xxCC	CCxx
0x2	CCxx	xxCC
0x3	CCxx	xxCC

The following example illustrates this for little endian.

```
always @(initiator_HWDATA or reg_HADDRM)
begin
  case (reg_HADDRM)
  2'b00   : target_HWDATA = initiator_HWDATA[15 : 0];
  2'b01   : target_HWDATA = initiator_HWDATA[15 : 0];
  2'b10   : target_HWDATA = initiator_HWDATA[31 : 16];
  2'b11   : target_HWDATA = initiator_HWDATA[31 : 16];
  default : target_HWDATA = 15'b0;
end
```

- Initiator: 32-bit data width
  Target: 8-bit data width

reg_HADDRM[1:0]	HWDATA[31:0] Little endian	HWDATA[31:0] Big endian
0x0	xxxC	Cxxx
0x1	xxCx	xCxx
0x2	xCxx	xxCx
0x3	Cxxx	xxxC

The following example illustrates this for little endian.

```
always @(initiator_HWDATA or reg_HADDRM)
begin
  case (reg_HADDRM)
  2'b00   : target_HWDATA = initiator_HWDATA[7 : 0];
  2'b01   : target_HWDATA = initiator_HWDATA[15 : 8];
  2'b10   : target_HWDATA = initiator_HWDATA[23 : 16];
  2'b11   : target_HWDATA = initiator_HWDATA[31 : 24];
  default : target_HWDATA = 15'b0;
end
```

- Initiator: 16-bit data width
  Target: 8-bit data width

reg_HADDRM[0:0]	HWDATA[15:0] Little endian	HWDATA[15:0] Big endian
0x0	xC	Cx
0x1	Cx	xC

The following example illustrates this for little endian.

```
always @(target_HRDATA or reg_HADDRM)
begin
  case (reg_HADDRM)
  1'b0    : initiator_HRDATA = target_HRDATA[7 : 0];
  1'b1    : initiator_HRDATA = target_HRDATA[15 : 8];
  default : initiator_HRDATA = 8'b0;
end
```

HRDATA Byte Lanes

The following tables show how the byte lanes of the ***HRDATA*** signal are multiplexed. In these tables, ***x*** indicates the unconnected data bytes and ***C*** indicates the data bytes that get connected from the active initiator to the active target (the actual valid data bytes depend on the transfer size).

- Initiator: 16-bit data width
  Target: 32-bit data width

reg_HADDRM[1:0]	HRDATA[31:0] Little endian	HRDATA[31:0] Big endian
0x0	xxCC	CCxx
0x1	xxCC	CCxx
0x2	CCxx	xxCC
0x3	CCxx	xxCC

The following example illustrates this for little endian.

```
always @(target_HRDATA or reg_HADDRM)
begin
  case (reg_HADDRM)
  2'b00   : initiator_HRDATA = target_HRDATA[15 : 0];
  2'b01   : initiator_HRDATA = target_HRDATA[15 : 0];
  2'b10   : initiator_HRDATA = target_HRDATA[31 : 16];
  2'b11   : initiator_HRDATA = target_HRDATA[31 : 16];
  default : initiator_HRDATA = 15'b0;
end
```

- Initiator: 8-bit data width
  Target: 32-bit data width

reg_HADDRM[1:0]	HRDATA[31:0] Little endian	HRDATA[31:0] Big endian
0x0	xxxC	Cxxx
0x1	xxCx	xCxx
0x2	xCxx	xxCx
0x3	Cxxx	xxxC

The following example illustrates this for little endian.

```
always @(target_HRDATA or reg_HADDRM)
begin
  case (reg_HADDRM)
  2'b00   : initiator_HRDATA = target_HRDATA[7 : 0];
  2'b01   : initiator_HRDATA = target_HRDATA[15 : 8];
  2'b10   : initiator_HRDATA = target_HRDATA[23 : 16];
  2'b11   : initiator_HRDATA = target_HRDATA[31 : 24];
  default : initiator_HRDATA = 15'b0;
end
```

- Initiator: 8-bit data width
  Target: 16-bit data width

reg_HADDRM[0:0]	HRDATA[31:0] Little endian	HRDATA[31:0] Big endian
0x0	xC	Cx
0x1	Cx	xC

The following example illustrates this for little endian.

```
always @(target_HRDATA or reg_HADDRM)
begin
  case (reg_HADDRM)
  1'b0    : initiator_HRDATA = target_HRDATA[7 : 0];
  1'b1    : initiator_HRDATA = target_HRDATA[15 : 8];
  default : initiator_HRDATA = 8'b0;
end
```

APB Node RTL Generator

An APB node is generated as one block containing:

- An AHB-to-APB bridge, which converts system-bus AHB transfers to APB transfers. The generated bridge is based on the AHB-to-APB bridge of the ***AHB Example AMBA System Technical Reference Manual***.
- A local address decoder, which minimizes the necessary logic. For logic optimization purposes, the decoding of the address is split into small comparators (switches). The ***comparator_width*** parameter allows you to specify how many address lines are compared in one switch. This enables you to target the generated address decoder towards your technology library.
- A peripheral-to-bus multiplexer, which multiplexes the APB target output signals that are involved in the transfer. If only one target is connected to the node, these multiplexers are not generated.

When there is an APB initiator, the generator generates the same kind of block, called ***APB***, but without creating an AHB-to-APB bridge.

Input-Stage and Output-Stage Node RTL Generator

The input-stage generator and the output-stage generator generate the required multilayer architecture, depending on the number of initiators and targets, the protocols (the actual pin mapping) that are used to connect the peripherals, and the parameters specified in the Platform Creator GUI.

Input-stage and output-stage nodes can be composed of the following blocks:

- Input Stage
- Address Decoder
- Output Stage
- Output Arbiter
- AHB Default Slave

The input stage, the address decoder, and the AHB default slave are part of the input-stage node, while the output stage and the output arbiter belong to the output-stage node.

For a description of the AMBA multilayer data width, see "Multilayer Bus Data Width" on page 114.

Input Stage

An input stage is responsible for holding the address and control information when the transfer to a shared slave cannot start immediately. An input stage is generated for each multilayer initiator. The generated input stage is based on the input stage of the ***AHB Example AMBA System - ARM DDI 0170A***.

NOTE The logic of the input stage is optimized to handle multiple output stages. If only one output stage is connected to an input stage, the generated logic of the input stage will not be optimal.

Address Decoder

A decoder is associated with each input stage. This decoder is used to determine the output stage that is required to complete an access. The input stage generator generates a decoder, which minimizes the necessary logic. For logic optimization purposes, the decoding of the address is split into small comparators (switches). The ***comparator_width*** parameter allows you to specify how many address lines are compared in one switch. This enables you to target the generated address decoder towards your technology library. For more information, see "Specifying the Comparator Width" on page 69.

Output Stage

An output stage is used to select which of the various input layers is routed to the target. An output stage is generated for each target. The generated output stage is based on the output stage of the ***AHB Example AMBA System - ARM DDI 0170A***.

NOTE The logic of the output stage is optimized to handle multiple input stages. If only one input stage is connected to an output stage, the generated logic of the output stage will not be optimal.

Output Arbiter

The arbiter checks which of the input stages has to perform a transfer to the shared target and decides which one currently has the highest priority. Each output stage contains an output arbiter.

The generated output arbiter with arbitration scheme round robin or fixed priority (see "Specifying the Arbitration Scheme" on page 70), is based on the output arbiter of the ***AHB Example AMBA System - ARM DDI 0170A***.

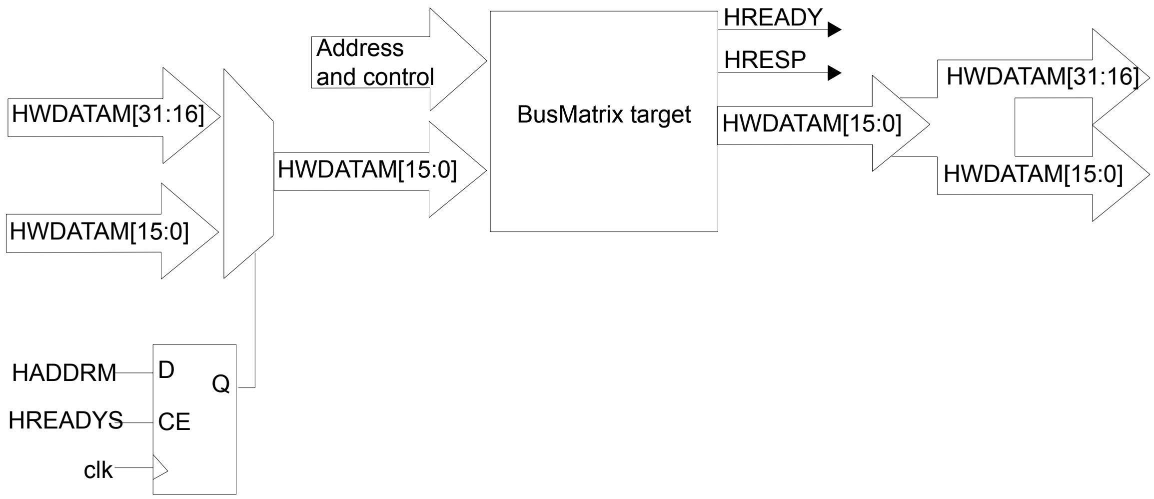
AHB Default Slave

The AHB default slave responds to transfers that are made to undefined regions of memory, where no AHB system slaves are mapped. If a default slave is necessary in the generated system, the AHB generator generates the default slave of the ***AHB Example AMBA System Technical Reference Manual***. It is not necessary to have a default slave in every node, because the default slave in the node directly connected to the AHB bus master will catch all possible transfers which the master tries to make to undefined memory regions. With this rule as basis, the generator tries to determine whether or not a default slave is necessary. However, you can overrule the rule through a parameter in Platform Creator. For more information on how the generator decides to generate a default slave or not and how to influence this, see "Specifying Whether the Bus Has a Default Slave" on page 69.

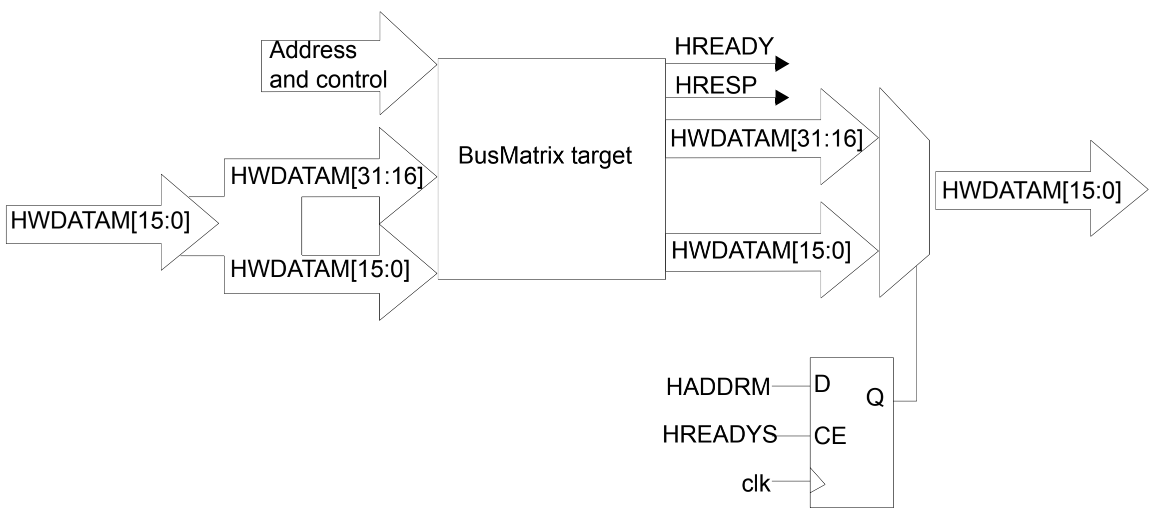
Multilayer Bus Data Width

If a target gets connected to a bus, the data width of which differs from its own data width, logic is provided by the generator to convert the data to the right size.

The following figure illustrates implementing a narrow target on a wider bus.



The following figure illustrates implementing a wide target on a narrow bus.



As shown in the figures above, when a narrow data signal needs to be converted to a wider data signal, the data is just replicated. And when a wide data signal needs to be converted to a narrow data signal, the data gets multiplexed to select the right byte lanes.

The following describes how the byte lanes of the ***HWDATA*** signal and the ***HRDATA*** signal for a data width up to 32 bits are multiplexed, respectively. For data widths greater than 32 bits, an equivalent multiplexing of the byte lanes is valid.

- HWDATA Byte Lanes
- HRDATA Byte Lanes

HWDATA Byte Lanes

The following tables show how the byte lanes of the ***HWDATA*** signal are multiplexed. In these tables, ***x*** indicates the unconnected data bytes and ***C*** indicates the data bytes that get connected from the active initiator to the active target (the actual valid data bytes depend on the transfer size).

- Initiator: 32-bit data width
  Target: 16-bit data width

reg_HADDR[1:0]	**HWDATA[31:0]**	
	Little endian	**Big endian**
0x0	xxCC	CCxx
0x1	xxCC	CCxx
0x2	CCxx	xxCC
0x3	CCxx	xxCC

The following example illustrates this for little endian.

```
always @(initiator_HWDATA or reg_HADDRM)
begin
  case (reg_HADDRM)
  2'b00   : target_HWDATA = initiator_HWDATA[15 : 0];
  2'b01   : target_HWDATA = initiator_HWDATA[15 : 0];
  2'b10   : target_HWDATA = initiator_HWDATA[31 : 16];
  2'b11   : target_HWDATA = initiator_HWDATA[31 : 16];
  default : target_HWDATA = 15'b0;
end
```

- Initiator: 32-bit data width
  Target: 8-bit data width

reg_HADDR[1:0]	**HWDATA[31:0]**	
	Little endian	**Big endian**
0x0	xxxC	Cxxx
0x1	xxCx	xCxx
0x2	xCxx	xxCx
0x3	Cxxx	xxxC

The following example illustrates this for little endian.

```
always @(initiator_HWDATA or reg_HADDRM)
begin
  case (reg_HADDRM)
  2'b00   : target_HWDATA = initiator_HWDATA[7 : 0];
  2'b01   : target_HWDATA = initiator_HWDATA[15 : 8];
  2'b10   : target_HWDATA = initiator_HWDATA[23 : 16];
  2'b11   : target_HWDATA = initiator_HWDATA[31 : 24];
  default : target_HWDATA = 15'b0;
end
```

- Initiator: 16-bit data width
  Target: 8-bit data width

reg_HADDR[0:0]	**HWDATA[15:0]**	
	Little endian	**Big endian**
0x0	xC	Cx
0x1	Cx	xC

The following example illustrates this for little endian.

```
always @(target_HRDATA or reg_HADDRM)
begin
  case (reg_HADDRM)
  1'b0    : initiator_HRDATA = target_HRDATA[7 : 0];
  1'b1    : initiator_HRDATA = target_HRDATA[15 : 8];
  default : initiator_HRDATA = 8'b0;
end
```

HRDATA Byte Lanes

The following tables show how the byte lanes of the ***HRDATA*** signal are multiplexed. In these tables, ***x*** indicates the unconnected data bytes and ***C*** indicates the data bytes that get connected from the active initiator to the active target (the actual valid data bytes depend on the transfer size).

- Initiator: 16-bit data width
  Target: 32-bit data width

reg_HADDR[1:0]	**HRDATA[31:0]**	
	Little endian	**Big endian**
0x0	xxCC	CCxx
0x1	xxCC	CCxx
0x2	CCxx	xxCC
0x3	CCxx	xxCC

The following example illustrates this for little endian.

```
always @(target_HRDATA or reg_HADDRM)
begin
  case (reg_HADDRM)
  2'b00   : initiator_HRDATA = target_HRDATA[15 : 0];
  2'b01   : initiator_HRDATA = target_HRDATA[15 : 0];
  2'b10   : initiator_HRDATA = target_HRDATA[31 : 16];
  2'b11   : initiator_HRDATA = target_HRDATA[31 : 16];
  default : initiator_HRDATA = 15'b0;
end
```

- Initiator: 8-bit data width
  Target: 32-bit data width

reg_HADDR[1:0]	**HRDATA[31:0]**	
	Little endian	**Big endian**
0x0	xxxC	Cxxx
0x1	xxCx	xCxx
0x2	xCxx	xxCx
0x3	Cxxx	xxxC

The following example illustrates this for little endian.

```
always @(target_HRDATA or reg_HADDRM)
begin
  case (reg_HADDRM)
  2'b00   : initiator_HRDATA = target_HRDATA[7 : 0];
  2'b01   : initiator_HRDATA = target_HRDATA[15 : 8];
  2'b10   : initiator_HRDATA = target_HRDATA[23 : 16];
  2'b11   : initiator_HRDATA = target_HRDATA[31 : 24];
  default : initiator_HRDATA = 15'b0;
end
```

- Initiator: 8-bit data width
  Target: 16-bit data width

reg_HADDR[0:0]	HRDATA[15:0]	
	Little endian	Big endian
0x0	xC	Cx
0x1	Cx	xC

The following example illustrates this for little endian.

```
always @(target_HRDATA or reg_HADDRM)
begin
  case (reg_HADDRM)
  1'b0    : initiator_HRDATA = target_HRDATA[7 : 0];
  1'b1    : initiator_HRDATA = target_HRDATA[15 : 8];
  default : initiator_HRDATA = 8'b0;
end
```

Lite2AHB Bridge RTL Generator

The ***Lite2AHB*** generator generates a bridge that does the protocol conversion of an ***AHBLiteInitiator*** protocol to an ***AHBInitiator*** protocol.

DownSizer Bridge RTL Generator

The ***DownSizer*** generator generates a bridge can connect a 32-bit-wide peripheral or bus to a 64-bit-wide bus and perform 64-bit accesses to it.

Chapter 11

User-Defined Arbiter

The AMBA Bus Library has a number of predefined arbiters for both the AHB and the output-stage bus. It is possible to select a fixed-priority arbitration scheme or a round-robin arbitration scheme. For more information, see "Specifying the Arbitration Scheme" on page 50 and section "Specifying the Arbitration Scheme" on page 70.

However, using a user-defined arbiter is also supported, both at the TLM and the RTL abstraction level. These arbiters need to follow specific rules to ensure correct simulation with the TLM bus model or generated RTL code. These rules are explained in "AHB bus" on page 119 and "AHB bus" on page 119. This chapter also explains how to connect your user-defined arbiter to either the AHB or output-stage node in Platform Creator.

This chapter describes:

- AHB bus
- Output-stage Bus
- Connecting the User-Defined Arbiter in Platform Creator

AHB bus

This section describes:

- TLM
- RTL

TLM

This section describe the different parts that need to be present in a TLM arbiter that has to be connected to an AHB bus model.

- The Arbiter Interface
- The init() Function
- The arbitrate() Function
- The Arbiter During Reset
- Example of a User-Defined Arbiter for an AHB bus

The Arbiter Interface

The arbiters of the bus are ***sc_channel***s. The arbiter implements the arbiter interface of the bus. The class for your AHB arbiter has to be derived from the following arbiter interface: ***AMBA_AHB_Arbiter_if***.

The arbiter interface has the following purely virtual methods that need to be implemented in the arbiter:

- ***void init(ArbiterApi& api)***
- ***void arbitrate()***

The ***init()*** method is called by the bus simulator at initialization. The ***arbitrate()*** method is called by the bus simulator every cycle. The arbiter API that is passed when the ***init()*** function is called allows the arbiter developer to access port and node attributes or to query the state of the bus or to change the state of the bus.

The arbiter API for an AHB node looks as follows:

```
class AMBAAHBNodeArbiterApi : public AMBAAHBNodeApi {
  public:
    AMBAAHBNodeArbiterApi (AMBAAHBNode& n) : AMBAAHBNodeApi(n) {}
    ~AMBAAHBNodeArbiterApi () {}

    AMBATransaction* getCurrentTransaction ();
    AMBATransaction* getPreviousTransaction ();
    AMBATransaction* getNewTransaction ();
    void setArbitrated (InitiatorId id);
    BMSignal& getBmSignalSplitResume ();
};

class AMBAAHBNodeApi {
  public:
    typedef AMBAAHBNode Node;
    typedef AMBAAHBInitPort InitiatorPort;
    typedef AMBAAHBTargetPort TargetPort;
    typedef InitiatorPort* InitiatorId;
    typedef TargetPort* TargetId;

    AMBAAHBNodeApi (AMBAAHBNode& n) : mNode(n) {}
    ~AMBAAHBNodeApi () {}

    unsigned int getInitiatorCount () const;
    AMBAAHBInitPort* getInitiator (unsigned int index);
    AMBAAHBInitPort* getInitiator (const string& name);

    unsigned int getTargetCount () const;
    AMBAAHBTargetPort* getTarget (unsigned int index);
    AMBAAHBTargetPort* getTarget (const string& name);

    // userApi for bus
    bool queryHREADYHigh ();
    bool queryTwoCycleResponse ();


    // userApi for initiator
    void doEventGrantInitiator (const InitiatorId id);
    void doEventStartInitiator (const InitiatorId id);
    bool queryRequestInitiator (const InitiatorId id);
    bool queryLockInitiator (const InitiatorId id);


    // userApi for target

  protected:
    AMBAAHBNode& mNode;
};
```

The init() Function

The pointers to all the initiator ports are not available at construction time. Therefore, the ***init()*** member function of the arbiter is called at the end of elaboration. At that moment, the pointers to the ports are initialized. You can then retrieve the pointers by means of the ***getInitiator()*** function, and store them. The argument of this function is the name of a port. This list of names can be an argument of the arbiter constructor.

The ***init()*** function could look as follows:

```
void init (ArbiterApi& api)
{
ports = new AMBAAHBInitPort*[size];
mArbiterApi = &api;
ports[0] = mArbiterApi->getInitiator (<name given in Platform Creator>);
ports[1] = ...
}
```

The arbitrate() Function

The ***arbitrate()*** function of an arbiter has to decide which initiators can have access on the bus. At the end of the function, ***setArbitrated*** should be called on the ***arbiterApi***. The correct port, the port that can start a transaction, should be passed with the call.

An arbiter can query the state of the bus simulator by means of the ***arbiterApi***:

```
bool querypropertyName();
```

For an AHB arbiter, following the properties can be queried:

Property	Usage
HREADYHigh	If ***queryHREADYHigh()*** returns ***1***, this is equivalent with ***HREADY*** being high at the RTL level.
TwoCycleResponse	If ***queryTwoCycleResponse()*** returns ***1***, an ***eotTrf*** transfer has been sent with attribute ***status*** different from ***ok***.
Request	If ***queryRequest()*** returns ***1*** for a specific ***AMBAAHBInitPort***, the initiator that is connected to this port has requested the bus.
Lock	If ***queryLock()*** returns ***1*** for a specific ***AMBAAHBInitPort***, the initiator that is connected to this port requests to locked transactions.

The arbiter has to send back information to the bus simulator. The AHB arbiter has to indicate which initiator will be granted and when a transaction can be started.

This is done by the means of:

```
void doEventeventName();
```

Triggering an event corresponds with setting the input corresponding with the import to true for that clock cycle.

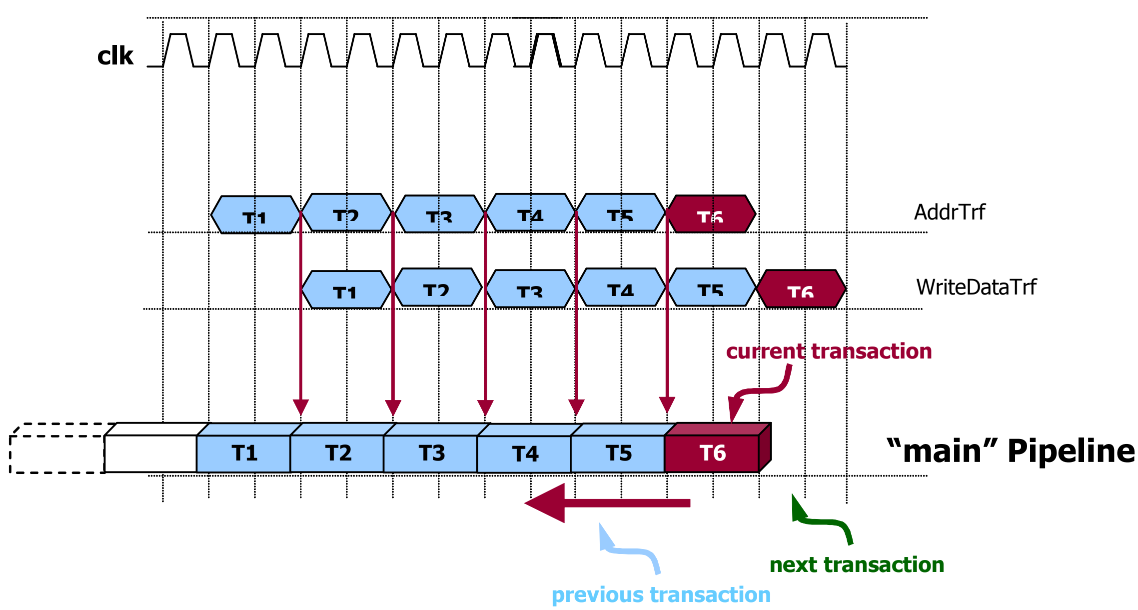
For an AHB arbiter, the following events needs to be signaled:

Event	Usage
Grant	Triggers ***doEventGrant()*** for a specific ***AMBAAHBInitPort*** to indicate that the initiator connected to this port can be granted the bus.
Start	Triggers ***doEventStart()*** for a specific ***AMBAAHBInitPort*** to indicate that the initiator connected to this port can start a transaction. This event has to be triggered when this ***AMBAAHBInitPort*** got a ***doEventGrant()*** last clock cycle and ***queryHREADYHigh()*** returns ***1*** this clock cycle.

Access to the transaction and its transfers is possible by means of:

- `Transaction* getCurrentTransaction();`
- `Transaction* getPreviousTransaction();`
- `Transaction* getNextTransaction();`

The following figure illustrates this.



The transaction class is the internal representation of the transaction in the bus simulator. It gives access to transfer attributes by means of the following syntax:

```
transaction->transferName.attributeName
```

For example:

```
AMBATransaction* currentTra = mArbiterApi->getCurrentTransaction();

if (currentTra && currentTra->AddrTrf.Group == tlmBurstStart
    && currentTra->AddrTrf.BurstLength != 0x3FF)
{
// This piece of code will be executed when attribute "Group" of the AddrTrf
// transfer of the current transaction equals "tlmBurstStart" and when attribute
// "BurstLength" of the same transaction does not equals "0x3FF. In AMBA HDL
// terms this means that the code will be executed when it is the start of a
// burst and it is not a burst of undefined length
}
```

The Arbiter During Reset

For both arbiters and decoders, there is a method ***void reset()*** which you can override if you want to. This function is called during reset and can be used if your arbiter implementation needs to be aware of reset.

Example of a User-Defined Arbiter for an AHB bus

```
#ifndef CWR_AMBA_AHB_ARBITER_Fixed_H
#define CWR_AMBA_AHB_ARBITER_Fixed_H

#include <systemc.h>
#include "AMBA/AMBA.h"              // header for userblocks
#include "AMBA/AMBABusModel.h"      // header for busmodel

namespace AMBA {

  class AMBA_AHB_Arbiter_Fixed: public AMBA_AHB_Arbiter_if, public sc_channel
  {
  public:

   AMBAAHBNodeArbiterScExport mArbiterPort;

   AMBA_AHB_Arbiter_Fixed(sc_module_name name) :
    sc_channel(name),
    mArbiterPort(*this),
    size(3),
    ports(0),
    mArbiterApi(0)
   {
    LockedTransfer = false;
    bHReadyHigh = false;
    BurstInProgress = 0;
    SplittedMaster = 0;
    splitint = 0;
    defaultMasterIndex = 2;

    CurrentlyGranted = defaultMasterIndex;
    tobeGranted = defaultMasterIndex;
   }

   ~AMBA_AHB_Arbiter_Fixed()
   {
    if (ports) {
      delete [] ports;
   }

  }

  // Simple example of arbiter function
  virtual void arbitrate()
  {
    splitResumeValue = mArbiterApi->getBmSignalSplitResume().getValue();
    splitint &= ~(splitResumeValue);
    AMBATransaction* currentTra = mArbiterApi->getCurrentTransaction();

    if(CurrentlyGranted == 0 && (LockedTransfer || splitResumeValue == 0))
      {
        if(!((splitint >> SplittedMaster) & 1))
        {
          CurrentlyGranted = SplittedMaster;
          if ( currentTra ) {
               currentTra->AddrTrf.MasterId = SplittedMaster;
         }
          ports[SplittedMaster-1]->doEventGrant();
          mArbiterApi->setArbitrated(ports[SplittedMaster-1]);
          return;
        }
        mArbiterApi->setArbitrated(0);
        return;
      }

    bHReadyHigh = mArbiterApi->queryHREADYHigh();
    if(mArbiterApi->queryTwoCycleResponse()){
      if(currentTra && currentTra->EotTrf.Status == tlmSplit){
        int MasterIDSplit = currentTra->AddrTrf.MasterId;
        splitint |= (1 << MasterIDSplit);
        if(LockedTransfer){
          SplittedMaster = MasterIDSplit;
          BurstInProgress = 0;
          CurrentlyGranted = 0;
          mArbiterApi->setArbitrated(0);
```

```
      return;
    }
  }
  BurstInProgress = 0;
}
else if(bHReadyHigh){
if (currentTra && currentTra->AddrTrf.Group == tlmBurstStart
    && currentTra->AddrTrf.BurstLength != 0x3FF) {
    BurstInProgress = currentTra->AddrTrf.BurstLength -1;
}
if (BurstInProgress != 0) {
  BurstInProgress--;
    if (currentTra && currentTra->AddrTrf.Group == tlmBurstIdle
        && currentTra->AddrTrf.Type == tlmIdle) {
        BurstInProgress++;
    }
}
bHReadyHigh = true;

if(CurrentlyGranted != 0) {
  ports[CurrentlyGranted-1]->doEventStart();
  }
}

if(!LockedTransfer || bHReadyHigh)
  {
    if(CurrentlyGranted != 0 && ports[CurrentlyGranted-1]->queryLock() && bHReadyHigh){
     LockedTransfer = true;
     }
    else{
     LockedTransfer = false;
     }
  }

  if(!LockedTransfer && BurstInProgress == 0){
    if(splitint == 0) {
      tobeGranted = defaultMasterIndex;
      for(unsigned int i = 1; i <= size; i++){
       if(ports[i-1]->queryRequest()){
        tobeGranted = i;
        break;
      }
      }
    }

     else {

      if((splitint >> defaultMasterIndex) & 1) {
       SplittedMaster = defaultMasterIndex;
       tobeGranted = 0;
      }
      else{
       tobeGranted = defaultMasterIndex;
     }

      for(unsigned int i = 1; i <= size; i++){
       if(ports[i-1]->queryRequest() && !((splitint >> i) & 1)){
         tobeGranted = i;
        break;
      }
      }
    }
  }

  if(tobeGranted != 0) {
    ports[tobeGranted-1]->doEventGrant();
    CurrentlyGranted = tobeGranted;
     mArbiterApi->setArbitrated(ports[tobeGranted-1]);
     return;
  }

  CurrentlyGranted = tobeGranted;
  mArbiterApi->setArbitrated(0);
  return;
}
```

```
    virtual void init(ArbiterApi& api)
    {
      ports = new AMBAAHBInitPort*[size];
      mArbiterApi = &api;
      ports[0] = mArbiterApi->getInitiator("initiator2");
      ports[1] = mArbiterApi->getInitiator("initiator1");
      ports[2] = mArbiterApi->getInitiator("initiator3");
      assert(ports[0]);
      assert(ports[1]);
      assert(ports[2]);
    }

  protected:
      const unsigned int size;
      AMBAAHBInitPort **ports;
      ArbiterApi* mArbiterApi;

      bool LockedTransfer;
      bool bHReadyHigh;
      int BurstInProgress;
      int SplittedMaster;
      int defaultMasterIndex;
      int CurrentlyGranted;
      int tobeGranted;
      int splitint;
      int splitResumeValue;
  };

} // namespace AMBA

#endif
```

RTL

It is possible to use an arbiter with support for up to 16 initiators. To allow this kind of flexibility, a number of terminals, like ***HBUSREQ***, ***HGRANT***, ***HLOCK***, and ***HSPLIT*** have a width which depends on the number of initiators that can be arbitrated by the user-defined arbiter. This implies that a user-defined arbiter that is designed to arbitrate four initiators should have a width of ***4*** for terminals ***HBUSREQ***, ***HGRANT***, ***HLOCK***, and ***HSPLIT***. To give even more flexibility in the kind of user-defined arbiter that is supported by the AHB RTL generator, for many terminals you can specify whether or not they are used within the code of the arbiter. However, some terminals are always necessary and do not have this option. The following tables give an overview of the required and the optional terminals.

- Required terminals:

Name	Width	Direction
HBUSREQ	Number of initiators	in
HGRANT	Number of initiators	out

- Optional terminals:

Name	Width	Direction
HSIZE	3	in
HBURST	3	in
HREADY	1	in
HRESP	2	in
HTRANS	2	in
HPROT	4	in
HLOCK	Number of initiators	in
HMASTER	4	out
HMASTERD	4	out
HMASTLOCK	1	out
HSPLIT	Number of initiators	in

Since the user-defined arbiter has to work together correctly with the generated AHB bus parts like the master-to-slave multiplexer, some rules have to be followed within the arbiter.

- The order of the request values in the ***HBUSREQ*** terminal is determined by the ***priority*** parameter that can be set in Platform Creator. For more information on this parameter, see "Specifying the Arbitration Priority of an AHB Bus Target Port" on page 51.

  For example, suppose you have three initiators ***Init1***, ***Init2***, and ***Init3*** with the following priority setting in Platform Creator:

Initiator	Priority setting
Init1	2
Init2	0
Init3	1

In this case, the ***HBUSREQ*** terminal consists of three bits. Bit 0 represents a request from initiator ***Init2***, bit 1 represents a request from initiator ***Init3***, and bit 2 represents a request from initiator ***Init1***.

Bit	Request from	HBUSREQ value (binary)
HBUSREQ(0)	Init2	XX1
HBUSREQ(1)	Init3	X1X
HBUSREQ(2)	Init1	1XX

- For the ***HGRANT*** terminal, the same rule has to be applied within the user-defined arbiter. This implies that the writer of the user-defined arbiter code has to make sure that the order of the grant values in the ***HGRANT*** terminals matches the ***priority*** parameter settings in Platform Creator.

  For example, this means that in the above example value ***1*** for ***HGRANT*** bit 0 means that initiator ***Init2*** is granted by the user-defined arbiter.

Bit	Grant to	HGRANT value (binary)
HGRANT(0)	Init2	001
HGRANT(1)	Init3	010
HGRANT(2)	Init1	100

- The values that have to be used for terminals ***HMASTER*** and ***HMASTERD*** are also determined by the values of the ***priority*** parameter settings. However, they are also determined by the presence of the ***HSPLIT*** terminals. If the ***HSPLIT*** terminal is present in at least one of the targets that are connected to the AHB node, the AHB node assumes that the AMBA 2.0 split feature is supported. This also implies the necessity to have a dummy master which is granted when all other masters are put in split mode. Value ***0*** of ***HMASTER*** and ***HMASTERD*** is used for this dummy master.
- In case no split is supported by the AHB node, it is a prerequisite for the user-defined arbiter to give ***HMASTER*** value ***0*** if the initiator, with ***priority*** parameter value set to ***0***, is granted. The ***HMASTER*** value is used to determine the address and control values that need to be selected in the master-to-slave multiplexer. If the AHB node does support split, value ***0*** has to be used for the dummy master and the user-defined arbiter has to increment the ***priority*** parameter value with one to give a value to ***HMASTER***.

  For example, in the above defined example, this would mean that the following values should be used for ***HMASTER***.

Grant to	HMASTER value if split is not supported	HMASTER value if split is supported
Init1	2	3
Init2	0	1
Init3	1	2

The same is valid for the ***HMASTERD*** terminal, which is used to determine the ***HWDATA*** value that is selected in the master-to-slave multiplexer.

Grant to (in address phase)	HMASTERD value if split is not supported (in data phase)	HMASTERD value if split is supported (in data phase)
Init1	2	3
Init2	0	1
Init3	1	2

- The order of the lock values in the ***HLOCK*** terminal is, like with the ***HBUSREQ*** terminal, set by the AHB generator and is also based on the ***priority*** parameter settings in Platform Creator.

  For example, in the above examples, this means that the following table will be used.

Bit	Lock from	HLOCK value (binary)
HLOCK(0)	Init2	XX1
HLOCK(1)	Init3	X1X
HLOCK(2)	Init1	1XX

Output-stage Bus

- TLM
- RTL

TLM

This section describes the different parts that need to be present in a TLM arbiter that has to be connected to an output-stage bus model.

- The Arbiter Interface
- The init() Function
- The arbitrate() Function
- The Arbiter During Reset
- Example of a User-Defined Arbiter for an Output Stage Bus

The Arbiter Interface

The arbiters of the bus are ***sc_channel***s. The arbiter implements the arbiter interface of the bus. The class for your AHB arbiter has to be derived from the following arbiter interface: ***AMBA_OutputStage_Arbiter_if***.

The arbiter interface has the following purely virtual methods that need to be implemented in the arbiter:

- ***void init(ArbiterApi& api)***
- ***void arbitrate()***

The ***init()*** method is called by the bus simulator at initialization. The ***arbitrate()*** method is called by the bus simulator every cycle. The arbiter API that is passed when the ***init()*** function is called allows the arbiter developer to access port and node attributes or to query the state of the bus or to change the state of the bus.

The arbiter API for an output-stage node looks as follows:

```
class AMBAOutputStageNodeArbiterApi : public AMBAOutputStageNodeApi {
  public:
    AMBAOutputStageNodeArbiterApi (AMBAOutputStageNode& n) : AMBAOutputStageNodeApi(n) {}
    ~AMBAOutputStageNodeArbiterApi () {}

    AMBATransaction* getCurrentTransaction ();
    AMBATransaction* getPreviousTransaction ();
    AMBATransaction* getNewTransaction ();
    void setArbitrated (InitiatorId id);
};

class AMBAOutputStageNodeApi {
  public:
    typedef AMBAOutputStageNode Node;
    typedef AMBAOutputStageInitPort InitiatorPort;
    typedef AMBAOutputStageTargetPort TargetPort;
    typedef InitiatorPort* InitiatorId;
    typedef TargetPort* TargetId;

    AMBAOutputStageNodeApi (AMBAOutputStageNode& n) : mNode(n) {}
    ~AMBAOutputStageNodeApi () {}

    unsigned int getInitiatorCount () const;
    AMBAOutputStageInitPort* getInitiator (unsigned int index);
    AMBAOutputStageInitPort* getInitiator (const string& name);

    unsigned int getTargetCount () const;
    AMBAOutputStageTargetPort* getTarget (unsigned int index);
    AMBAOutputStageTargetPort* getTarget (const string& name);

    // userApi for bus
    bool queryHREADYHigh ();


    // userApi for initiator
    void doEventGrantInitiator (const InitiatorId id);
    bool querySelectInitiator (const InitiatorId id);
    bool queryRequestInitiator (const InitiatorId id);


    // userApi for target

  protected:
    AMBAOutputStageNode& mNode;
};
```

The init() Function

The pointers to all the initiator ports are not available at construction time. Therefore, the ***init()*** member function of the arbiter is called at the end of elaboration. At that moment, the pointers to the ports are initialized. You can then retrieve the pointers by means of the ***getInitiator()*** function, and store them. The argument of this function is the name of a port. This list of names can be an argument of the arbiter constructor.

The ***init()*** function could look as follows:

```
void init (ArbiterApi& api)
{
  ports = new AMBAOutputStageInitPort*[size];
  mArbiterApi = &api;
  ports[0] = mArbiterApi->getInitiator (<name given in Platform Creator>);
  ports[1] = Ö
}
```

The arbitrate() Function

The ***arbitrate()*** function of an arbiter has to decide which initiators can have access on the bus. At the end of the function, ***setArbitrated()*** should be called on the ***arbiterApi***. The correct port, the port that can start a transaction, should be passed with the call.

An arbiter can query the state of the bus simulator by means of the ***arbiterApi***:

```
bool querypropertyName();
```

For an AHB arbiter, the following properties can be queried:

Property	Usage
HREADYHigh	If ***queryHREADYHigh()*** returns ***1***, this is equivalent with ***HREADY*** being high at the RTL level.
Select	If ***querySelect()*** returns ***1*** for a specific ***AMBAOutputStageInitPort***, the input stage that is connected to this port is selecting this output stage (the incoming address in intended for this output stage).
Request	If ***queryRequest()*** returns ***1*** for a specific ***AMBAOutputStageInitPort***, the input stage that is connected to this port is selecting this output stage and will start a transaction, so in HDL ***HTRANS*** would be different from ***idle*** or ***busy***.

The arbiter has to send information back to the bus simulator. The AHB arbiter has to indicate which initiator will be granted and when a transaction can be started. This is done by the means of:

```
void doEventeventName();
```

Triggering an event corresponds with setting the input corresponding with the import to true for that clock cycle.

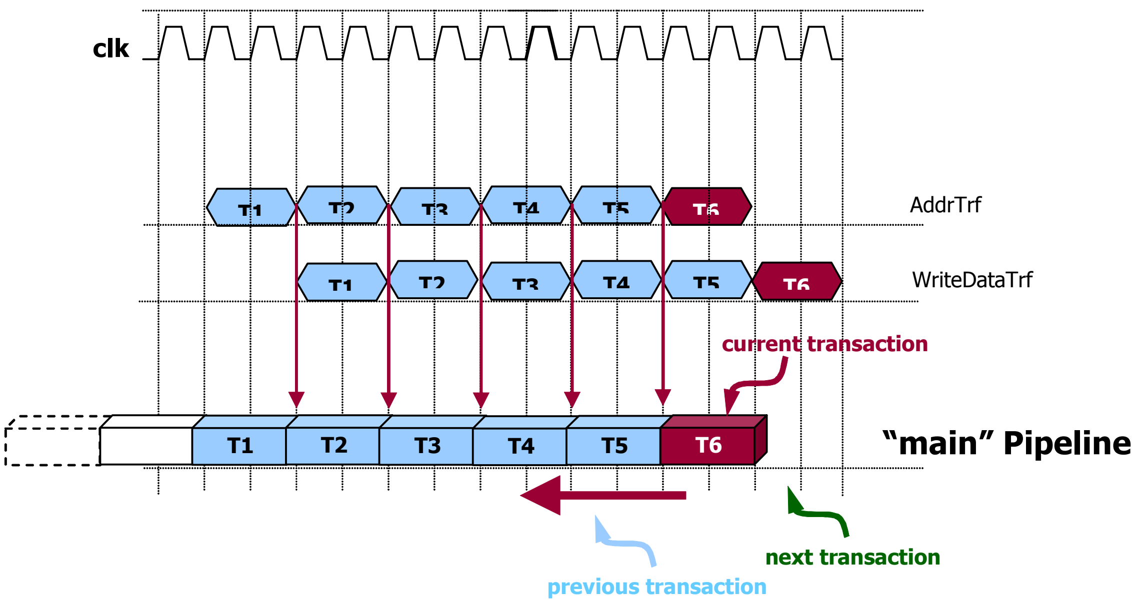
For an AHB arbiter, the following events need to be signaled:

Event	Usage
Grant	Trigger ***doEventGrant()*** for a specific ***AMBAOutputStageInitPort*** to indicate that the input stage connected to this port can be granted the bus.

Access to the transaction and its transfers, is possible by means of:

- `Transaction* getCurrentTransaction();`
- `Transaction* getPreviousTransaction();`
- `Transaction* getNextTransaction();`

The following figure illustrates this.



The transaction class is the internal representation of the transaction in the bus simulator. It gives access to transfer attributes by means of the following syntax:

transaction->***transferName.attributeName***

For example:

```
AMBATransaction* currentTra = mArbiterApi->getCurrentTransaction();

if (currentTra && currentTra->AddrTrf.Group == tlmBurstStart
    && currentTra->AddrTrf.BurstLength != 0x3FF)
{
// This piece of code will be executed when attribute "Group" of the AddrTrf
// transfer of the current transaction equals "tlmBurstStart" and when attribute
// "BurstLength" of the same transaction does not equals "0x3FF. In AMBA HDL
// terms this means that the code will be executed when it is the start of a
// burst and it is not a burst of undefined length
}
```

The Arbiter During Reset

For both arbiters and decoders, there is a method ***void reset()*** that you can override if you want to. This function is called during reset and can be used if your arbiter implementation needs to be aware of reset.

Example of a User-Defined Arbiter for an Output Stage Bus

```
#ifndef CWR_AMBA_OUTPUTSTAGE_ARBITER_FIXED_H
#define CWR_AMBA_OUTPUTSTAGE_ARBITER_FIXED_H

#include <systemc.h>
#include "AMBA/AMBA.h"            // header for userblocks
#include "AMBA/AMBABusModel.h"    // header for busmodel

namespace AMBA {

  class AMBA_OutputStage_Arbiter_Fixed: public AMBA_OutputStage_Arbiter_if, public sc_channel
  {

   AMBAOutputStageNodeArbiterScExport mArbiterPort;

   AMBA_OutputStage_Arbiter_Fixed(sc_module_name name) :
     sc_channel(name),
     mArbiterPort(*this),
     CurrentlyGranted(0),
     size(3),
     mArbiterApi(0),
     ports(0)
   {

   }

   ~AMBA_OutputStage_Arbiter_Fixed()
   {
    if (ports) {
      delete [] ports;
    }
  }

  virtual void arbitrate()
   {
    int tobeGranted = 0;
    bool bMastLock = false;
    AMBATransaction *currentTra = mArbiterApi->getCurrentTransaction();
    bool bIdle = currentTra ? true : false;

    if ( currentTra && CurrentlyGranted != 0 ) {
     bIdle = currentTra->AddrTrf.Type == tlmIdle &&
             currentTra->AddrTrf.Group != tlmBurstIdle;
     bMastLock = currentTra->LockTrf.Lock;
    }
    if(!bIdle && !bMastLock && mArbiterApi->queryHREADYHigh()) {
             tobeGranted = 0;
     for ( unsigned int i = 1; i <= size; i++ ){
       if (ports[i-1]->queryRequest()||
         (CurrentlyGranted == i && ports[i-1]->querySelect() ) ) {
             tobeGranted = i;
             break;
            }
       }
    }else if( bIdle && !bMastLock && mArbiterApi->queryHREADYHigh()) {
         tobeGranted = 0;
     for (unsigned int i = 1; i <= size; i++ ){
       if (ports[i-1]->queryRequest()){
             tobeGranted = i;
             break;
            }
       }
        if(tobeGranted == 0 && ports[CurrentlyGranted-1]->querySelect()) {
         tobeGranted = CurrentlyGranted;
        }
    } else {
         tobeGranted = CurrentlyGranted;
    }
    CurrentlyGranted = tobeGranted;
    if(tobeGranted != 0){
      ports[tobeGranted-1]->doEventGrant();
    }
    mArbiterApi->setArbitrated(ports[tobeGranted-1]);
    return;
  }
```

```
  virtual void init(ArbiterApi& api)
  {
    ports = new AMBAOutputStageInitPort*[size];
    mArbiterApi = &api;
    ports[0] = mArbiterApi->getInitiator("inputstage2");
    ports[1] = mArbiterApi->getInitiator("inputstage1");
    ports[2] = mArbiterApi->getInitiator("inputstage3");
    assert(ports[0]);
    assert(ports[1]);
    assert(ports[2]);
  }


  protected:
    unsigned int CurrentlyGranted;
    const unsigned int size;
    ArbiterApi* mArbiterApi;
    AMBAOutputStageInitPort **ports;
  };

}

#endif
```

RTL

To enable the user-defined arbiter to support arbitration for a specific number of initiators, the ***Request*** terminal has a width which depends on the number of initiators that can be arbitrated by the user-defined arbiter. Consequently, the width of the ***AddrInPort*** is determined by the number of bits that are necessary to represent the supported number of initiators. This implies that a user-defined arbiter that is designed to arbitrate four initiators, should have a width of ***4*** for its ***Request*** terminal and a width of ***2*** for its ***AddrInPort*** terminal. To give even more flexibility in the kind of user-defined arbiter that is supported by the output-stage RTL generator, for many terminals you can specify whether or not they are used within the code of the arbiter. However, some terminals are always necessary and do not have this option. The following tables give an overview of the required and the optional terminals.

- Required terminals:

Name	Width	Direction
Request	Number of initiators	in
AddrInPort	Number of initiators bits	out
NoPort	1	out

- Optional terminals:

Name	width	Direction
Burst	3	in
Size	3	in
Prot	4	in
Sel	1	in
Mastlock	1	in
Trans	2	in
Ready	1	in
Resp	1	in

Since the user-defined arbiter has to work together correctly with the generated output-stage bus, some rules have to be followed within the arbiter.

- The order of the request values in the ***Request*** terminal is determined by the ***priority*** parameter that can be set in Platform Creator. For more information on this parameter, see "Specifying the Arbitration Priority of an Output-Stage Bus Target Port" on page 71.

  For example, suppose you have three initiators ***Init1***, ***Init2***, and ***Init3*** with the following priority setting in Platform Creator:

Initiator	Priority setting
Init1	2
Init2	0
Init3	1

In this case, the ***Request*** terminal consists of three bits. Bit 0 represents a request from initiator ***Init2***, bit 1 represents a request from initiator ***Init3***, and bit 2 represents a request from initiator ***Init1***.

Bit	Request from	HBUSREQ value (binary)
HBUSREQ(0)	Init2	XX1
HBUSREQ(1)	Init3	X1X
HBUSREQ(2)	Init1	1XX

- The value that has to be give to the ***AddrInPort*** terminal in the user-defined arbiter also needs to be based on the ***priority*** parameter. But the ***AddrInPort*** does not use one-hot encoding. So the ***priority*** parameter does not determine the active bit, but the value which needs to be passed to the ***AddrInPort*** terminal.

  For example, in the above example value 2 has to be given to the ***AddrInPort*** terminal by the user-defined arbiter if initiator ***Init1*** is granted access to the output-stage bus .

Grant to	**AddrInPort value**
Init1	2
Init2	0
Init3	1

Connecting the User-Defined Arbiter in Platform Creator

You can thread your arbiter as any other user block in Platform Creator.

To connect the user-defined arbiter in Platform Creator:

1. In the System Diagram, instantiate the AHB or output-stage node to which you want to connect your arbiter.
2. Right click on the node to bring up the pop-up menu and from the pop-up menu, select ***Instantiate Port > arbiter***.
3. Connect this arbiter port to the correct port on your arbiter.
4. Propagate the port properties by selecting ***Tools > Propagate Port Properties*** from the menu bar. The arbiter port of the node will automatically take over the properties (for example the terminals to use) from the connected port of your arbiter.

NOTE The ***Propagate Port Properties*** menu command also checks the system, so you are advised to select this menu command when your complete system is assembled.

Chapter 12

Using the Generated Bus Model in Incisive

This chapter describes how to use the generated bus model in Incisive.

Use Model

It is not advisable to manually create the interconnect system by instantiating the appropriate classes. The supported use model is that you use Platform Creator to create the system. If required, the code generated by Platform Creator must then manually be modified to be able to run in Incisive.

Installing the AMBA Bus Library for Incisive

You should follow the steps as described in the ***ConvergenSC Installation Manual*** when installing the AMBA Bus Library. During the process, you will get the opportunity to select compilation for Incisive.

This will precompile the AMBA Bus Library.

You can then use the installed bus library in Incisive by setting the correct include and library paths when compiling your system.

Compiling the System for Incisive

The AMBA Bus Library can be used in Incisive 5.3 and Incisive 5.4.

- Incisive 5.3
- Incisive 5.4

Incisive 5.3

Make sure your environment is set up correctly for using Incisive 5.3.
Make sure you have selected compilation for Incisive during the installation of the AMBA Bus Library.

To start Incisive 5.3:

Type:

```
ncsc_run -FILE sc_build.cmd
```

where ***sc_build.cmd*** is a file containing the following options:

```
-I${COWAREHOME}/cowaresc_stub/include
-I${IPLOCATION}/AMBA_BL/SystemC/include
test.cc
-L${IPLOCATION}/AMBA_BL/Incisive_5.3/lib/${COWAREHT}
-L${COWAREHOME}/cowaresc_stub/lib/Incisive_5.3_gcc-2.95.3
-Wl,-E
-lAMBABusModel
-lcowaresc_analysis_stub
-lcowaresc_simulator_stub
-SC_MAIN
-GCC_VERS 2.95.3
-GNU
-STATIC
```

${IPLOCATION} must point to the directory where the AMBA Bus Library has been installed. It can be replaced by the actual path if desired.

NOTE These include paths and libraries are also the ones needed to create libraries (binary IP) that can be used in simulations of ConvergenSC and Incisive 5.3.

Incisive 5.4

Make sure your environment is set up correctly for using Incisive 5.4.
Make sure you have selected compilation for Incisive during the installation of the AMBA Bus Library.

To start Incisive 5.4:

Type:

```
ncsc_run -FILE sc_build.cmd
```

where ***sc_build.cmd*** is a file containing the following options:

```
-I${COWAREHOME}/cowaresc_stub/include
-I${IPLOCATION}/AMBA_BL/SystemC/include
test.cc
-L${IPLOCATION}/AMBA_BL/Incisive_5.4/lib/${COWAREHT}
-L${COWAREHOME}/cowaresc_stub/lib/Incisive_5.4_gcc-3.2.3
-Wl,-E
-lAMBABusModel
-lcowaresc_analysis_stub
-lcowaresc_simulator_stub
-SC_MAIN
-GCC_VERS 3.2.3
-GNU
-STATIC
```

${IPLOCATION} must point to the directory where the AMBA Bus Library has been installed. It can be replaced by the actual path if desired.

NOTE These include paths and libraries are also the ones needed to create libraries (binary IP) that can be used in simulations of ConvergenSC and Incisive 5.4.

Enabling Analysis in the AMBA Bus Models

To enable SystemC Verification (SCV) transaction recording, every node of the bus structure dumped by Platform Creator has a method ***enable_scv_recording(void)***, which is responsible for creating all the needed recording streams and transaction generators, as well as for adding the SCV recording actions to the action lists. You can call this at any point in time (that is, during construction, elaboration, or simulation).

The creation of the database is **not** the responsibility of the bus models: it is your responsibility to create the database and set the database that you whish the bus models to record to as the default database. By default, the default database is the last created database, but you can explicitly set the default database by calling the static method ***scv_tr_db::set_default_db()***.

In practise, to enable analysis in incisive you need to:

1. Add the following include statement to the file with ***sc_main()***:

   ```
   #include "cve.h" // Needed for cve_tr_sdi_init()
   ```

2. Add the following statements at the start of ***sc_main()***:

   ```
   cve_tr_sdi_init(); // To add Cadence's analysis callbacks to scv.
   scv_tr_db mydb("mydb"); // Generic initialization of the database.
   ```

3. Call ***enable_scv_recording()*** for each node for which you want to perform analysis.

Appendix A

AMBA TLM API Quick Reference

This chapter describes:

Protocols and Port Types

- AHBInitiator
- AHB Target
- AHBLite Initiator
- AHBLite Target
- APB Target

AHBInitiator

The port types are:

```
AHBInitiator_inoutmaster_port<address_width, data_width>
AHBInitiator_inmaster_port<address_width, data_width>
AHBInitiator_outmaster_port<address_width, data_width>
```

AHB Target

The port types are:

```
AHBTarget_inoutslave_port<address_width, data_width>
AHBTarget_inslave_port<address_width, data_width>
AHBTarget_outslave_port<address_width, data_width>
```

AHBLite Initiator

The port types are:

```
AHBLiteInitiator_inoutmaster_port<address_width, data_width>
AHBLiteInitiator_inmaster_port<address_width, data_width>
AHBLiteInitiator_outmaster_port<address_width, data_width>
```

AHBLite Target

The port types are:

```
AHBLiteTarget_inoutslave_port<address_width, data_width>
AHBLiteTarget_inslave_port<address_width, data_width>
AHBLiteTarget_outslave_port<address_width, data_width>
```

APB Target

The port types are:

```
APBTarget_inoutslave_port<address_width, data_width>
APBTarget_inslave_port<address_width, data_width>
APBTarget_outslave_port<address_width, data_width>
```

TLM API Methods

```
<port>.sendTransaction()
<port>.getTransaction()

<port>.canReceiveTrfName()
<port>.canSendTrfName()

<port>.sendDelayedTrfName()
<port>.sendTrfName()

<port>.getReceiveTrfNameEventFinder()
<port>.getReceiveTrfNameEvent()

<port>.getSendTrfNameEvent()
<port>.getSendTrfNameEventFinder()
```

For a list of available transfers, see "Methods Available in Each Port Type" on page 145.

For a list of available transfer attributes, see "AHB Transfer Attributes and API Guide" on page 146.

Methods Available in Each Port Type

- API Usage

API Usage

S= send***TrfName***, canSend***TrfName***, getSend***TrfName***Event, getSend***TrfName***EventFinder

D= sendDelayed***TrfName***

R= canReceive***TrfName***, getReceive***TrfName***EventFinder, getReceive***TrfName***Event

	ReqTrf	UnreqTrf	GrantTrf	AddrTrf	WriteDataTrf	ReadDataTrf	EotTrf	SplitResumeTrf	LockTrf	ShiftTrf	CancelTrf
AHBInitiator_inoutmaster_port	S	S	R	S	S	R	R		S	R	S
AHBInitiator_inmaster_port	S	S	R	S		R	R		S	R	S
AHBInitiator_outmaster_port	S	S	R	S	S		R		S	R	S
AHBTarget_inoutslave_port				R	R	SD	SD	S	R		
AHBTarget_inslave_port				R	R		SD	S	R		
AHBTarget_outslave_port				R		SD	SD	S	R		
AHBLiteInitiator_inoutmaster_port				S	S	R	R		S	R	
AHBLiteInitiator_inmaster_port				S		R	R		S	R	

AHBLiteInitiator_outmaster_port	S	S		R		S	R
AHBLiteTarget_inoutslave_port	R	R	SD	SD		R	
AHBLiteTarget_inslave_port	R	R		SD		R	
AHBLiteTarget_outslave_port	R		SD	SD		R	
APBTarget_inoutslave_port	R	S	S				
APBTarget_inslave_port	R	S					
APBTarget_outslave_port	R		S				

AHB Transfer Attributes and API Guide

ReqTrf (Initiator)

Attributes

Attribute	Value	Mapping to Signal
ReqMode	tlmReqOneCycle	HREQ = 1 for one cycle
	tlmReqUntilGrant	HREQ = 1 until HGRANT = 1
	tlmReqUntilUnreq	HREQ = 1 until UnreqTrf is sent

General Usage

```
if (port.canSendReqTrf()){
    port.getReqTrf()->setReqMode(tlmReqOneCycle);
    port.sendReqTrf();
}
```

Static Sensitivity

```
class my_module: public sc_module
{
public:
...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSendReq);
    sensitive << port.getSendReqTrfEventFinder();
    dont_initialize();
  ...
}
  ...

  void port_canSendReq();
...
}//end of my_module

void my_module::port_canSendReq(void){
  ...
  port.getReqTrf()->setReqMode(tlmReqUntilUnreq);
  port.sendReqTrf();
  ...
}
```

Dynamic Sensitivity

```
wait(port.getReceiveSendReqtTrfEvent);
port.getReqTrf()->setReqMode(tlmReqUntilGrant);
port.sendReqTrf();
```

UnreqTrf (Initiator)

- Attributes
- General Usage
- Static Sensitivity
- Dynamic Sensitivity

Attributes

There are no attributes.

General Usage

```
if (port.canSendUnreqTrf()){
    port.sendUnreqTrf();
}
```

Static Sensitivity

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSendUnreq);
      sensitive << port.getSendUnreqTrfEventFinder();
      dont_initialize();
    ...
  }
...
void port_canSendUnreq();
...
}//end of my_module

void my_module::port_canSendUnreq(void){
  ...
  port.sendUnreqTrf();
  ...
}
```

Dynamic Sensitivity

```
wait(port.getReceiveSendUnreqtTrfEvent);
port.getUnreqTrf();
port.sendUnreqTrf();
```

GrantTrf (Initiator)

- Attributes
- General Usage
- Static Sensitivity
- Dynamic Sensitivity

Attributes

There are no attributes.

General Usage

```
if (port.getGrantTrf()){
  //execute code
}
```

Static Sensitivity

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_doWhenGranted);
      sensitive << port.getReceiveGrantTrfEventFinder();
      dont_initialize();
    ...
  }
...

void port_doWhenGranted();
...

}//end of my_module

void my_module::port_doWhenGranted(void){

  //execute code

}
```

Dynamic Sensitivity

```
wait(port.getReceiveGrantTrfEvent);
```

AddrTrf (Initiator and Target)

- Attributes
- General Usage in Initiator
- Static Sensitivity in Initiator

- Dynamic Sensitivity in Initiator
- Cross-Referencing Transfers in Initiator

Attributes

Attribute	Type	Value	Mapping to Signal
Address	unsigned int	32 bit value	HADDR
Type	tlmTTransactionType	tlmIdle, tlmWriteAtAddress tlmReadAtAddress	HTRANS[1] = 0 (HWRITE = 1) & (HTRANS[1] = 1) (HWRITE = 0) & (HTRANS[1] = 1)
Kind	tlmTKind	tlmOpcode tlmData	HPROT[0] = 0 HPROT[0] = 1
AccessSize	unsigned int	8 16 32 64	HSIZE = 0x0 HSIZE = 0x1 HSIZE = 0x2 HSIZE = 0x3
Group	tlmTGroup	tlmSingle tlmBurstStart tlmBurstCont tlmBurstIdle	(HTRANS = NONSEQ(0x2)) & (HBURST = SINGLE(0)) (HTRANS = NONSEQ(0x2)) & (HBURST != SINGLE(0)) (HTRANS = SEQ(0x3) & (HBURST != SINGLE(0)) HTRANS = BUSY(0x1)
BurstLength	unsigned int	0x4 0x8 0x10 0x3FF	(HBURST = WRAP4(0x2)) | (HBURST = INCR4(0x3)) (HBURST = WRAP8(0x4)) | (HBURST = INCR8(0x5)) (HBURST = WRAP16(0x6)) | (HBURST = INCR16(0x7)) HBURST = INCR (0x1)
BurstWrap	tlmTBurstWrap	tlmIncremental tlmWrapBurstSize	HBURST = INCR (0x1, 0x3, 0x5, 0x7) HBURST = WRAP (0x2, 0x4, 0x6)
Cacheable	unsigned int	false true	HPROT[3] = 0 HPROT[3] = 1
Bufferable	unsigned int	false true	HPROT[2] = 0 HPROT[2] = 1
ProtectionType	tlmTProtectionType	tlmUser tlmPrivileged	HPROT[1] = 0 HPROT[1] = 1

General Usage in Initiator

```
if ( port.getAddrTrf() ){
  ...
  port.AddrTrf->setAddress( 0x1000);
  port.AddrTrf->setType(tlmWriteAtAddress);
  port.AddrTrf->setAccessSize(32);
  port.AddrTrf->setGroup(tlmSingle);
  port.AddrTrf->setBurstLength(0x3FF);
  port.AddrTrf->setBurstWrap(tlmIncremental);
  port.AddrTrf->setCacheable(false);
  port.AddrTrf->setBufferable(true);
  port.AddrTrf->setProtectionType(tlmPrivileged);
  port.sendAddrTrf();
  ...
}
```

Static Sensitivity in Initiator

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSendAddr);
      sensitive << port.getSendAddrTrfEventFinder();
      dont_initialize();
    ...

  }
  ...

  void port_canSendAddr();
  ...

}//end of my_module

void my_module::port_canSendAddr(void){
  //set attributes
  port.getAddrTrf()->setAddress( 0x1000);
  port.AddrTrf->setType(tlmWriteAtAddress);
  port.AddrTrf->setAccessSize(32);
  port.AddrTrf->setGroup(tlmSingle);
  port.AddrTrf->setBurstLength(0x3FF);
  port.AddrTrf->setBurstWrap(tlmIncremental);
  port.AddrTrf->setCacheable(false);
  port.AddrTrf->setBufferable(true);
  port.AddrTrf->setProtectionType(tlmPrivileged);
  //send trf
  port.sendAddrTrf();
}
```

Dynamic Sensitivity in Initiator

```
wait( port.getSendAddrTrfEvent() );

//set attributes
port.getAddrTrf()->setAddress( 0x1000);
port.AddrTrf->setType(tlmWriteAtAddress);
port.AddrTrf->setAccessSize(32);
...
//send trf
port.sendAddrTrf();
```

Cross-Referencing Transfers in Initiator

There are no cross-references available.

AddrTrf (Target)

- Attributes
- General Usage in Target
- Static Sensitivity in Target
- Dynamic Sensitivity in Target
- Cross-Referencing Transfers in Target

Attributes

Attribute	Type	Value	Mapping to Signal
masterID	unsigned int	4 bit value	HMASTERID

General Usage in Target

```
if ( port.getAddrTrf() ){
  ...
  unsigned int v_address = port.AddrTrf->getAddress();
  tlmTTransactionType v_type = port.AddrTrf->getType();
  unsigned int accessSize v_size = port.AddrTrf->getAccessSize();
  tlmTGroup v_group = port.AddrTrf->getGroup();
  unsigned int v_burstLength = port.AddrTrf-getBurstLength();
  tlmTBurstWrap v_burstWrap = port.AddrTrf->getBurstWrap();
  bool v_cacheable = port.AddrTrf->getCacheable();
  bool v_bufferable = port.AddrTrf->getBufferable();
  tlmTProtectionType v_pType = port.AddrTrf->getProtectionType();
  unsigned int v_masterID = port.AddrTrf->getMasterID();
  ...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_receiveAddr);
      sensitive << port.getReceiveAddrTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_receiveAddr();
  ...

}//end of my_module

void my_module::port_receiveAddr(void){

  //get attributes
  unsigned int v_address = port.getAddrTrf()->getAddress();
  tlmTTransactionType v_type = port.AddrTrf->getType();
  unsigned int accessSize v_size = port.AddrTrf->getAccessSize();
  ...

}
```

Dynamic Sensitivity in Target

```
wait( port.getReceiveAddrTrfEvent() );

//get attributes
unsigned int v_address = port.AddrTrf->getAddress();
tlmTTransactionType v_type = port.AddrTrf->getType();
...
```

Cross-Referencing Transfers in Target

```
port.getAddrTrf();
value = port.AddrTrf->getLockTrf()->getLock();
```

LockTrf (Target)

- Attributes
- General Usage in Target
- Static Sensitivity in Target
- Dynamic Sensitivity in Target
- Cross-Referencing Transfers in Target

Attributes

Attribute	Type	Value	Mapping to Signal
Lock	unsigned int	true false	HMASTLOCK

General Usage in Target

```
if ( port.getLockTrf() ){
  ...
  bool int v_locked = port.LockTrf()->getLock();
  ...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_receiveLock);
      sensitive << port.getReceiveLockTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_receiveLock();
  ...

}//end of my_module

void my_module::port_receiveLock(void){

  bool lock = port.getLockTrf()->getLock();

}
```

Dynamic Sensitivity in Target

```
wait( port.getReceiveLockTrfEvent() );
bool lock = port.getLockTrf()->getLock();
```

Cross-Referencing Transfers in Target

There are no cross-references available.

LockTrf (Initiator)

- Attributes
- General Usage in Initiator
- Static Sensitivity in Initiator
- Dynamic Sensitivity in Initiator
- Cross-Referencing Transfers in Initiator

Attributes

Attribute	Type	Value	Mapping to Signal
Lock	unsigned int	boolean	HLOCK

NOTE ***LockTrf*** must be sent one cycle before the corresponding ***AddrTrf***.

General Usage in Initiator

```
if ( port.getLockTrf() ){
   ...
   port.LockTrf->setLock( true);
   port.sendLockTrf();
   ...
}
```

Static Sensitivity in Initiator

```
class my_module: public sc_module
{
public:
   ...
   // Constructor
   SC_CTOR(my_module){
      ...
      SC_METHOD(port_canSendLock);
         sensitive << port.getSendLockTrfEventFinder();
         dont_initialize();
      ...
   }
   ...

   void port_canSendLock();
   ...

}//end of my_module

void my_module::port_canSendLock(void){

   port.getLockTrf()->setLock(true);
   port.sendLockTrf();

}
```

Dynamic Sensitivity in Initiator

```
wait( port.getSendLockTrfEvent() );
port.getLockTrf()->setLock(true);
port.sendLockTrf();
```

Cross-Referencing Transfers in Initiator

There are no cross-references available.

WriteDataTrf (Initiator and Target)

- Attributes
- General Usage in Initiator
- Static Sensitivity in Initiator
- Dynamic Sensitivity in Initiator
- Cross-Referencing Transfers in Initiator
- General Usage in Target
- Static Sensitivity in Target
- Dynamic Sensitivity in Target
- Cross-Referencing Transfers in Target

Attributes

Attribute	Type	Value	Mapping to Signal
WriteData	unsigned int (data_width <=32) int 64 (data_width > 32)	8 -16 - 32 bit value 64 bit value	HWDATA

General Usage in Initiator

```
if ( port.getWriteDataTrf() ){
   ...
   port.WriteDataTrf->setWriteData(0xAABBCCDD);
   port.sendWriteDataTrf();
   ...
}
```

Static Sensitivity in Initiator

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSendWriteData);
      sensitive << port.getSendWriteDataTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_canSendWriteData();
  ...

}//end of my_module

void my_module::port_canSendWriteData(void){

  port.getWriteDataTrf()->setWriteData(0xAABBCCDD);
  port.sendWriteDataTrf();

}
```

Dynamic Sensitivity in Initiator

```
wait( port.getSendWriteDataTrfEvent() );
port.getWriteDataTrf()->setWriteData(0xAABBCCDD);
port.sendWriteDataTrf();
```

Cross-Referencing Transfers in Initiator

```
port.getWriteDataTrf();
value = port.WriteDataTrf->getAddrTrf()->getAddrTrfAttribute();
value = port.WriteDataTrf->getLockTrf()->getLock();
```

General Usage in Target

```
if ( port.getWriteDataTrf() ){
  ...
  unsigned int v_data = port.WriteDataTrf->getWriteData();
  ...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_getsWriteData);
      sensitive << port.getReceiveWriteDataTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_getsWriteData();
  ...

}//end of my_module

void my_module::port_getsWriteData(void){

  unsigned int v_data = port.getWriteDataTrf()->getWriteData();

}
```

Dynamic Sensitivity in Target

```
wait( port.getReceiveWriteDataTrfEvent() );
unsigned int v_data = port.getWriteDataTrf()->getWriteData();
```

Cross-Referencing Transfers in Target

```
port.getWriteDataTrf();
value = port.WriteDataTrf->getAddrTrf()->getAddrTrfAttribute();
value = port.WriteDataTrf->getLockTrf()->getLock();
```

ReadDataTrf (Initiator and Target)

- Attributes
- General Usage in Target
- Static Sensitivity in Target
- Dynamic Sensitivity in Target
- Cross-Referencing Transfers in Target
- General Usage in Initiator
- Static Sensitivity in Initiator
- Dynamic Sensitivity in Initiator
- Cross-Referencing Transfers in Initiator

Attributes

Attribute	Type	Value	Mapping to Signal
ReadData	unsigned int (data_width <=32) int 64 (data_width > 32)	8 -16 - 32 value 64 bit value	HRDATA

General Usage in Target

```
if ( port.getReadDataTrf() ){
  ...
  port.ReadDataTrf->setReadData(0xAABBCCDD);
  port.sendReadDataTrf(); //or   port.sendDelayedReadDataTrf(int n = delay)
  ...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSendReadData);
      sensitive << port.canSendReadDataTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_canSendReadData();
  ...

}//end of my_module

void my_module::port_canSendReadData(void){

  port.getReadDataTrf()->setReadData(0xAABBCCDD);
  port.sendReadDataTrf();//or   port.sendDelayedReadDataTrf(int n = delay)

}
```

Dynamic Sensitivity in Target

```
wait( port.getSendReadDataTrfEvent() );
port.getReadDataTrf()->setReadData(0xAABBCCDD);
port.sendReadDataTrf(); //or port.sendDelayedReadDataTrf(int n = delay)
```

Cross-Referencing Transfers in Target

```
port.getReadDataTrf();
value = port.ReadDataTrf->getAddrTrf()->getAddrTrfAttribute();
value = port.ReadDataTrf->getLockTrf()->getLock();
```

General Usage in Initiator

```
if ( port.getReadDataTrf() ){
  ...
  unsigned int v_data = port.ReadDataTrf->getReadData();
  ...
}
```

Static Sensitivity in Initiator

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_getsReadData);
      sensitive << port.getReceiveReadDataTrfEventFinder();
      dont_initialize();
    ...
  }
...

  void port_getsReadData();
  ...

}//end of my_module

  void my_module::port_getsReadData(void){

unsigned int v_data = port.getReadDataTrf() ->getReadData();

}
```

Dynamic Sensitivity in Initiator

```
wait( port.getReceiveReadDataTrfEvent() );
unsigned int v_data = port.getReadDataTrf()->getReadData();
```

Cross-Referencing Transfers in Initiator

```
port.getReadDataTrf();
value = port.ReadDataTrf->getAddrTrf()->getAddrTrfAttribute();
value = port.ReadDataTrf->getLockTrf()->getLock();
value = port.ReadDataTrf->getEotTrf()->getStatus();
```

EotTrf (Initiator and Target)

Attributes

Attribute	Type	Value	Mapping to Signal
status	tlmTStatus	tlmOk tlmError tlmSplit tlmRetry	(HREADY = 1) & (HRESP = 0x0) (HREADY = 0) & (HRESP = 0x1) (HREADY = 0) & (HRESP = 0x3) (HREADY = 0) & (HRESP = 0x2)

General Usage in Target

```
if ( port.getEotTrf() ){
  ...
  port.EotTrf->setStatus(tlmOk);
  port.sendEotTrf(); // or port.sendDelayedEotTrf(int n = delay);
...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSendEot);
      sensitive << port.canSendEotTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_canSendEot();

...

}//end of my_module

void my_module::port_canSendEot(void){

  port.getEotTrf()->setStatus(tlmSplit);
  port.sendEotTrf(); // or port.sendDelayedEotTrf(int n = delay);

}
```

Dynamic Sensitivity in Target

```
wait( port.getSendEotTrfEvent() );
port.getEotTrf()->setStatus(tlmOk);
port.sendEotTrf(); // or port.sendDelayedEotTrf(int n = delay);
```

Cross-Referencing Transfers in Target

```
port.getEotTrf();
value = port.EotTrf->getAddrTrf()->getAddrTrfAttribute();
value = port.EotTrf->getLockTrf()->getLock();
```

General Usage in Initiator

```
if ( port.getEotTrf() ){
  ...
  tlmTStatus v_status = port.EotTrf->getStatus();
  ...
}
```

Static Sensitivity in Initiator

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){

    ...
    SC_METHOD(port_getsEot);
      sensitive << port.getReceiveEotTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_getsEot();
  ...

}//end of my_module

void my_module::port_getsEot(void){

  tlmTStatus v_status = port.getEotTrf()->getStatus();

}
```

Dynamic Sensitivity in Initiator

```
wait( port.getReceiveEotTrfEvent() );
tlmTStatus v_status = port.getEotTrf()->getStatus();
```

Cross-Referencing Transfers in Initiator

```
port.getEotTrf();
value = port.EotTrf->getAddrTrf()->getAddrTrfAttribute();
value = port.EotTrf->getLockTrf()->getLock();
value = port.EotTrf->getWriteDataTrf->getWriteData();
value = port.EotTrf->getReadDataTrf->getReadData();
```

SplitResumeTrf (Target)

- Attributes
- General Usage in Target
- Static Sensitivity in Target
- Dynamic Sensitivity in Target
- Cross-Referencing Transfers in Target

Attributes

Attribute	Type	Value	Mapping to Signal
splitResume	unsigned int	0x1 to 0xF	HSPLIT

General Usage in Target

```
if ( port.getSplitResumeTrf() ){
  ...
  port.SplitResumeTrf->setSplitResume(0x2);
  port.sendSplitResumeTrf();
  ...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSplitResume);
      sensitive << port.canSendSplitResumeTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_canSendSplitResume();
  ...

}//end of my_module

void my_module::port_canSendSplitResume(void){

  port.getSplitResumeTrf()->setSplitResume(0x2);
  port.sendSplitResumeTrf();

}
```

Dynamic Sensitivity in Target

```
wait( port.getSendSplitResumeTrfEvent() );
port.getSplitResumeTrf()->setSplitResume(0x2);
port.sendSplitResumeTrf();
```

Cross-Referencing Transfers in Target

There are no cross references.

CancelTrf (Initiator)

Attributes

There are no attributes.

NOTE ***Initiator*** must send ***CancelTrf*** only when receiving split/retry response.

General Usage in Initiator

```
if ( port.getCancelTrf() ){
  ...
  port.sendCancelTrf();
  ...
}
```

Static Sensitivity in Initiator

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSendCancelTrf);
      sensitive << port.getSendCancelTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_canSendCancelTrf();
  ...

}//end of my_module

void my_module::port_canSendCancelTrf(void){
  port.getCancelTrf();
  port.sendCancelTrf();
}
```

Dynamic Sensitivity in Initiator

```
wait( port.getSendCancelTrfEvent() );
port.getCancelTrf();
port.sendCancelTrf();
```

Cross-Referencing Transfers in Initiator

There are no cross references.

AHBLite Transfer Attributes and API Guide

AddrTrf (Initiator and Target)

Attributes

Attribute	Type	Value	Mapping to Signal
Address	unsigned int	32 bit value	HADDR
Type	tlmTTransactionType	tlmIdle, tlmWriteAtAddress tlmReadAtAddress	HTRANS[1] = 0 (HWRITE = 1) & (HTRANS[1] = 1) (HWRITE = 0) & (HTRANS[1] = 1)
Kind	tlmTKind	tlmOpcode tlmData	HPROT[0] = 0 HPROT[0] = 1
AccessSize	unsigned int	8 16 32 64	HSIZE = 0x0 HSIZE = 0x1 HSIZE = 0x2 HSIZE = 0x3

Group	tlmTGroup	tlmSingle tlmBurstStart tlmBurstCont tlmBurstIdle	(HTRANS = NONSEQ(0x2)) & (HBURST = SINGLE(0)) (HTRANS = NONSEQ(0x2)) & (HBURST != SINGLE(0)) (HTRANS = SEQ(0x3) & (HBURST != SINGLE(0)) HTRANS = BUSY(0x1)
BurstLength	unsigned int	0x4 0x8 0xF 0x3FF	(HBURST = WRAP4(0x2)) | (HBURST = INCR4(0x3)) (HBURST = WRAP8(0x4)) | (HBURST = INCR8(0x5)) (HBURST = WRAP16(0x6)) | (HBURST = INCR16(0x7)) HBURST = INCR (0x1)
BurstWrap	tlmTBurstWrap	tlmIncremental tlmWrapBurstSize	HBURST = INCR (0x1, 0x3, 0x5, 0x7) HBURST = WRAP (0x2, 0x4, 0x6)
Cacheable	unsigned int	false true	HPROT[3] = 0 HPROT[3] = 1
Bufferable	unsigned int	false true	HPROT[2] = 0 HPROT[2] = 1
ProtectionType	tlmTProtectionType	tlmUser tlmPrivileged	HPROT[1] = 0 HPROT[1] = 1

General Usage in Initiator

```
if ( port.getAddrTrf() ){
  ...
  port.AddrTrf->setAddress( 0x1000);
  port.AddrTrf->setType(tlmWriteAtAddress);
  port.AddrTrf->setAccessSize(32);
  port.AddrTrf->setGroup(tlmSingle);
  port.AddrTrf->setBurstLength(0x3FF);
  port.AddrTrf->setBurstWrap(tlmIncremental);
  port.AddrTrf->setCacheable(false);
  port.AddrTrf->setBufferable(true);
  port.AddrTrf->setProtectionType(tlmPrivileged);
  port.sendAddrTrf();
  ...
}
```

Static Sensitivity in Initiator

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSendAddr);
      sensitive << port.getSendAddrTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_canSendAddr();
  ...

}//end of my_module

void my_module::port_canSendAddr(void){
  //set attributes
  port.getAddrTrf()->setAddress( 0x1000);
  port.AddrTrf->setType(tlmWriteAtAddress);
  port.AddrTrf->setAccessSize(32);
  port.AddrTrf->setGroup(tlmSingle);
  port.AddrTrf->setBurstLength(0x3FF);
  port.AddrTrf->setBurstWrap(tlmIncremental);
  port.AddrTrf->setCacheable(false);
  port.AddrTrf->setBufferable(true);
  port.AddrTrf->setProtectionType(tlmPrivileged);
  //send trf
  port.sendAddrTrf();
}
```

Dynamic Sensitivity in Initiator

```
wait( port.getSendAddrTrfEvent() );
//set attributes
port.getAddrTrf()->setAddress( 0x1000);
port.AddrTrf->setType(tlmWriteAtAddress);
port.AddrTrf->setAccessSize(32);
...
//send trf
port.sendAddrTrf();
```

Cross-Referencing Transfers in Initiator

There are no cross references available.

General Usage in Target

```
if ( port.getAddrTrf() ){
  ...
  unsigned int v_address = port.AddrTrf->getAddress();
  tlmTTransactionType v_type = port.AddrTrf->getType();
  unsigned int accessSize v_size = port.AddrTrf->getAccessSize();
  tlmTGroup v_group = port.AddrTrf->getGroup();
  unsigned int v_burstLength = port.AddrTrf->getBurstLength();
  tlmTBurstWrap = port.AddrTrf->getBurstWrap();
  bool v_cacheable = port.AddrTrf->getCacheable();
  bool v_bufferable = port.AddrTrf->getBufferable();
  tlmTProtectionType = port.AddrTrf->getProtectionType();
  ...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
  ...
    SC_METHOD(port_receiveAddr);
      sensitive << port.getReceiveAddrTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_receiveAddr();
  ...

}//end of my_module

void my_module::port_receiveAddr(void){

  //get attributes
  unsigned int v_address = port.getAddrTrf()->getAddress();
  tlmTTransactionType v_type = port.AddrTrf->getType();
  unsigned int accessSize v_size = port.AddrTrf->getAccessSize();
  ...
}
```

Dynamic Sensitivity in Target

```
wait( port.getReceiveAddrTrfEvent() );

//get attributes
unsigned int v_address = port.AddrTrf->getAddress();
tlmTTransactionType v_type = port.AddrTrf->getType();
...
```

Cross-Referencing Transfers in Target

```
port.getAddrTrf();
value = port.AddrTrf->getLockTrf()->getLock();
```

LockTrf (Initiator and Target)

- Attributes
- General Usage in Target
- Static Sensitivity in Target
- Dynamic Sensitivity in Target
- Cross-Referencing Transfers in Target
- General Usage in Initiator
- Static Sensitivity in Initiator
- Dynamic Sensitivity in Initiator

- Cross-Referencing Transfers in Initiator

Attributes

Attribute	Type	Value	Mapping to Signal
Lock	unsigned int	boolean	HMASTLOCK

General Usage in Target

```
if ( port.getLockTrf() ){
  ...
  bool int v_locked = port.LockTrf()->getLock();
  ...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_receiveLock);
      sensitive << port.getReceiveLockTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_receiveLock();
  ...

}//end of my_module

void my_module::port_receiveLock(void){

  bool lock = port.getLockTrf()->getLock();

}
```

Dynamic Sensitivity in Target

```
wait( port.getReceiveLockTrfEvent() );
bool lock = port.getLockTrf()->getLock();
```

Cross-Referencing Transfers in Target

There are no cross references available.

General Usage in Initiator

```
if ( port.getLockTrf() ){
  ...
  port.LockTrf->setLock( true);
  port.sendLockTrf();
  ...
}
```

Static Sensitivity in Initiator

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSendLock);
      sensitive << port.getSendLockTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_canSendLock();
  ...

}//end of my_module

void my_module::port_canSendLock(void){

  port.getLockTrf()->setLock(true);
  port.sendLockTrf();

}
```

Dynamic Sensitivity in Initiator

```
wait( port.getSendLockTrfEvent() );
port.getLockTrf()->setLock(true);
port.sendLockTrf();
```

Cross-Referencing Transfers in Initiator

There are no cross references available.

WriteDataTrf (Initiator and Target)

- Attributes
- General Usage in Initiator
- Static Sensitivity in Initiator
- Dynamic Sensitivity in Initiator
- Cross-Referencing Transfers in Initiator
- General Usage in Target
- Static Sensitivity in Target
- Dynamic Sensitivity in Target
- Cross-Referencing Transfers in Target

Attributes

Atribute	Type	Value	Mapping to Signal
WriteData	unsigned int (data_width <=32) int64 (data_width > 32)	8 -16 -32 bit value 64 bit value	HWDATA

General Usage in Initiator

```
if ( port.getWriteDataTrf() ){
  ...
  port.WriteDataTrf->setWriteData(0xAABBCCDD);
  port.sendWriteDataTrf();
  ...
}
```

Static Sensitivity in Initiator

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSendWriteData);
      sensitive << port.getSendWriteDataTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_canSendWriteData();
  ...

}//end of my_module

void my_module::port_canSendWriteData(void){

  port.getWriteDataTrf()->setWriteData(0xAABBCCDD);
  port.sendWriteDataTrf();

}
```

Dynamic Sensitivity in Initiator

```
wait( port.getSendWriteDataTrfEvent() );
port.getWriteDataTrf()->setWriteData(0xAABBCCDD);
port.sendWriteDataTrf();
```

Cross-Referencing Transfers in Initiator

```
port.getWriteDataTrf();
value = port.WriteDataTrf->getAddrTrf()->getAddrTrfAttribute();
value = port.WriteDataTrf->getLockTrf()->getLock();
```

General Usage in Target

```
if ( port.getWriteDataTrf() ){
  ...
  unsigned int v_data = port.WriteDataTrf->getWriteData();
  ...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_getsWriteData);
      sensitive << port.getReceiveWriteDataTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_getsWriteData();
  ...

}//end of my_module

void my_module::port_getsWriteData(void){

  unsigned int v_data = port.getWriteDataTrf()->getWriteData();

}
```

Dynamic Sensitivity in Target

```
wait( port.getReceiveWriteDataTrfEvent() );
unsigned int v_data = port.getWriteDataTrf()->getWriteData();
```

Cross-Referencing Transfers in Target

```
port.getWriteDataTrf();
value = port.WriteDataTrf->getAddrTrf()->getAddrTrfAttribute();
value = port.WriteDataTrf->getLockTrf()->getLock();
```

ReadDataTrf (Initiator and Target)

- Attributes
- General Usage in Target
- Static Sensitivity in Target
- Dynamic Sensitivity in Target
- Cross-Referencing Transfers in Target
- General Usage in Initiator

- Static Sensitivity in Initiator
- Dynamic Sensitivity in Initiator
- Cross-Referencing Transfers in Initiator

Attributes

Attribute	Type	Value	Mapping to Signal
ReadData	unsigned int (data_width <=32) int64 (data_width > 32)	8 -16 - 32 bit value 64 bit value	HRDATA

General Usage in Target

```
if ( port.getReadDataTrf() ){
  ...
  port.ReadDataTrf->setReadData(0xAABBCCDD);
  port.sendReadDataTrf(); //or   port.sendDelayedReadDataTrf(int n = delay)
  ...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSendReadData);
      sensitive << port.canSendReadDataTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_canSendReadData();
  ...

}//end of my_module

void my_module::port_canSendReadData(void){

  port.getReadDataTrf()->setReadData(0xAABBCCDD);
port.sendReadDataTrf(); //or
             port.sendDelayedReadDataTrf(int n = delay)

}
```

Dynamic Sensitivity in Target

```
wait( port.getSendReadDataTrfEvent() );
port.getReadDataTrf()->setReadData(0xAABBCCDD);
port.sendReadDataTrf(); //or
              port.sendDelayedReadDataTrf(int n = delay)
```

Cross-Referencing Transfers in Target

```
port.getReadDataTrf();
value = port.ReadDataTrf->getAddrTrf()->getAddrTrfAttribute();
value = port.ReadDataTrf->getLockTrf()->getLock();
```

General Usage in Initiator

```
if ( port.getReadDataTrf() ){
  ...
  unsigned int v_data = port.ReadDataTrf->getReadData();
  ...
}
```

Static Sensitivity in Initiator

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_getsReadData);
      sensitive << port.getReceiveReadDataTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_getsReadData();
  ...

}//end of my_module

void my_module::port_getsReadData(void){
  unsigned int v_data = port.getReadDataTrf()->getReadData();

}
```

Dynamic Sensitivity in Initiator

```
wait( port.getReceiveReadDataTrfEvent() );
unsigned int v_data = port.getReadDataTrf()->getReadData();
```

Cross-Referencing Transfers in Initiator

```
port.getReadDataTrf();
value = port.ReadDataTrf->getAddrTrf()->getAddrTrfAttribute();
value = port.ReadDataTrf->getLockTrf()->getLock();
value = port.ReadDataTrf->getEotTrf()->getStatus();
```

EotTrf (Initiator and Target)

Attributes

Attribute	Type	Value	Mapping to Signal
status	tlmTStatus	tlmOk tlmError	(HREADY = 1) & (HRESP = 0x0) (HREADY = 0) & (HRESP = 0x1)

General Usage in Target

```
if ( port.getEotTrf() ){
  ...
  port.EotTrf->setStatus(tlmOk);
  port.sendEotTrf(); // or
                port.sendDelayedEotTrf(int n = delay);
  ...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSendEot);
      sensitive << port.canSendEotTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_canSendEot();
  ...

}//end of my_module

  void my_module::port_canSendEot(void){

  port.getEotTrf()->setStatus(tlmOk);
  port.sendEotTrf(); // or
                 port.sendDelayedEotTrf(int n = delay);

}
```

Dynamic Sensitivity in Target

```
wait( port.getSendEotTrfEvent() );
port.getEotTrf()->setStatus(tlmOk);
port.sendEotTrf(); // or port.sendDelayedEotTrf(int n = delay);
```

Cross-Referencing Transfers in Target

```
port.getEotTrf();
value = port.EotTrf->getAddrTrf()->getAddrTrfAttribute();
value = port.EotTrf->getLockTrf()->getLock();
```

General Usage in Initiator

```
if ( port.getEotTrf() ){
  ...
  tlmTStatus v_status = port.EotTrf->getStatus();
  ...
}
```

Static Sensitivity in Initiator

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_getsEot);
      sensitive << port.getReceiveEotTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_getsEot();
  ...

}//end of my_module

void my_module::port_getsEot(void){

  tlmTStatus v_status = port.getEotTrf()->getStatus();

}
```

Dynamic Sensitivity in Initiator

```
wait( port.getReceiveEotTrfEvent() );
tlmTStatus v_status = port.getEotTrf()->getStatus();
```

Cross-Referencing Transfers in Initiator

```
port.getEotTrf();
value = port.EotTrf->getAddrTrf()->getAddrTrfAttribute();
value = port.EotTrf->getLockTrf()->getLock();
value = port.EotTrf->getWriteDataTrf->getWriteData();
value = port.EotTrf->getReadDataTrf->getReadData();
```

APB Transfer Attributes and API Guide

- AddrTrf (Target)
- WriteDataTrf (Target)
- ReadDataTrf (Target)

AddrTrf (Target)

- Attributes
- General Usage in Target
- Static Sensitivity in Target
- Dynamic Sensitivity in Target
- Cross-Referencing Transfers in Target

Attributes

Attribute	Type	Value	Mapping to Signal
address	unsigned int	32 bit value	PADDR
type	tlmTTransactionType	tlmWriteAtAddress tlmReadAtAddress	PWRITE = 1 PWRITE = 0

General Usage in Target

```
if ( port.getAddrTrf() ){
  ...
  unsigned int v_address = port.AddrTrf->getAddress();
  tlmTTransactionType v_type = port.AddrTrf->getType();
  ...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_receiveAddr);
      sensitive << port.getReceiveAddrTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_receiveAddr();
  ...

}//end of my_module

void my_module::port_receiveAddr(void){

  //get attributes
  unsigned int v_address = port.getAddrTrf()->getAddress();
  tlmTTransactionType v_type = port.AddrTrf->getType();
  ...

}
```

Dynamic Sensitivity in Target

```
wait( port.getReceiveAddrTrfEvent() );

//get attributes
unsigned int v_address = port.AddrTrf->getAddress();
tlmTTransactionType v_type = port.AddrTrf->getType();
...
```

Cross-Referencing Transfers in Target

```
port.getAddrTrf();
value = port.AddrTrf->getWriteDataTrf()->getWriteData();
```

WriteDataTrf (Target)

- Attributes
- General Usage in Target
- Static Sensitivity in Target
- Dynamic Sensitivity in Target
- Cross-Referencing Transfers in Target

Attributes

Attribute	Type	Value	Mapping to Signal
WriteData	unsigned int	32 bit value	PWDATA

General Usage in Target

```
if ( port.getWriteDataTrf() ){
  ...
  unsigned int v_data = port.WriteDataTrf->getWriteData();
  ...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_getsWriteData);
      sensitive << port.getReceiveWriteDataTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_getsWriteData();
  ...

}//end of my_module

void my_module::port_getsWriteData(void){

  unsigned int v_data = port.getWriteDataTrf()->getWriteData();

}
```

Dynamic Sensitivity in Target

```
wait( port.getReceiveWriteDataTrfEvent() );
unsigned int v_data = port.getWriteDataTrf()->getWriteData();
```

Cross-Referencing Transfers in Target

```
port.getWriteDataTrf();
value = port.WriteDataTrf->getAddrTrf()->getType();
value = port.WriteDataTrf->getAddrTrf()->getAddress();
```

ReadDataTrf (Target)

- Attributes
- General Usage in Target
- Static Sensitivity in Target
- Dynamic Sensitivity in Target
- Cross-Referencing Transfers in Target

Attributes

Attribute	Type	Value	Mapping to Signal
ReadData	unsigned int	32 bit value	PRDATA

General Usage in Target

```
if ( port.getReadDataTrf() ){
  ...
  port.ReadDataTrf->setReadData(0xAABBCCDD);
  port.sendReadDataTrf();
  ...
}
```

Static Sensitivity in Target

```
class my_module: public sc_module
{
public:
  ...
  // Constructor
  SC_CTOR(my_module){
    ...
    SC_METHOD(port_canSendReadData);
      sensitive << port.canSendReadDataTrfEventFinder();
      dont_initialize();
    ...
  }
  ...

  void port_canSendReadData();
...

}//end of my_module

void my_module::port_canSendReadData(void){

  port.getReadDataTrf()->setReadData(0xAABBCCDD);
  port.sendReadDataTrf();

}
```

Dynamic Sensitivity in Target

```
wait( port.getSendReadDataTrfEvent() );
port.getReadDataTrf()->setReadData(0xAABBCCDD);
port.sendReadDataTrf();
```

Cross-Referencing Transfers in Target

```
port.getReadDataTrf();
value = port.ReadDataTrf->getAddrTrf()->getType();
value = port.ReadDataTrf->getAddrTrf()->getAddress();
```