AMBA-PV Extensions to TLM 2.0

Developer Guide

ARM®

ARM DUI 0455G (ID060613)

AMBA-PV Extensions to TLM 2.0
Developer Guide

Release Information

Change History

Date	Issue	Confidentiality	Change
February 2009	A	Non-Confidential	New document based on AMBA-PV header files.
April 2009	B	Non-Confidential	Update to fix defects.
September 2009	C	Non-Confidential	Split AMBA-PV documentation into Developer Guide and Reference Manual. Added documentation for basic transactions, example systems and the creation of AMBA-PV compliant models.
March 2010	D	Non-Confidential	Added documentation for the AMBA-PV protocol checker. Added description of AMBA-PV extension attributes and methods.
November 2011	E	Non-Confidential	Documented changes to support the AMBA4 buses AXI4, ACE and DVM.
May 2012	F	Non-Confidential	Update for Fast Models v7.1.
December 2012	G	Non-Confidential	Update for Fast Models v8.0.

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents
AMBA-PV Extensions to TLM 2.0 Developer Guide

Preface

This preface introduces the *AMBA-PV Extensions to TLM 2.0 Developer Guide*. It contains the following sections:

About this book

This document describes the classes and interfaces included in the *AMBA-PV Extensions to TLM 2.0* (AMBA-PV). These classes and interfaces provide a *Programmer's View* (PV) of the AMBA® 4 buses.

Intended audience

This document is written for experienced hardware and software developers to aid the development of TLM 2.0 compatible models that communicate over AMBA 4 buses.

You must be familiar with:

- the basic concepts of C++ such as classes and inheritance
- SystemC and TLM 2.0 standards.

Using this book

This document is organized into the following chapters:

Chapter 1 *Introduction*

Read this for an introduction to AMBA-PV.

Chapter 2 *AMBA-PV Extension*

Read this for a detailed description of the AMBA-PV extension.

Chapter 3 *AMBA-PV Classes*

Read this for an overview of the AMBA-PV classes and interfaces.

Chapter 4 *Example Systems*

Read this for a description of the example systems provided with AMBA-PV.

Chapter 5 *How to Create AMBA-PV Compliant Models*

Read this for guidelines on creating AMBA-PV compliant models.

Chapter 6 *AMBA-PV Protocol Checker*

Read this for a description of the AMBA-PV protocol checker and the checks it performs.

Glossary

The *ARM Glossary* is a list of terms used in ARM documentation, together with definitions for those terms. The *ARM Glossary* does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See *ARM Glossary*, `http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html`.

Typographical conventions

The typographical conventions are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Additional reading

This section lists related publications by ARM® and by third parties.

See Infocenter, http://infocenter.arm.com for access to ARM documentation.

ARM publications

The book contains information that is specific to this product. The following publications provide reference information about the ARM architecture:

- *AMBA APB Protocol Specification* (ARM IHI 0024)
- *AMBA AXI Protocol Specification* (ARM IHI 0022)
- *AMBA AHB-Lite Protocol Specification* (ARM IHI 0033)
- *AMBA Specification* (ARM IHI 0011)
- *ARM Architecture Reference Manuals*, http://infocenter.arm.com/help/topic/com.arm.doc.set.architecture/index.html.

The following publications provide information about related ARM products and toolkits:

- *AMBA-PV Extensions to TLM 2.0 Reference Manual* (ARM DUI 0522).

Other publications

This section lists relevant documents published by third parties.

For further information on the Accellera Systems Initiative, see Accellera, http://www.accellera.org.

The following publications provide reference information about SystemC and TLM 2.0 standards:

- IEEE Std 1666-2005, *SystemC Language Reference Manual*, 31 March 2006
- *OSCI TLM-2.0 Language Reference Manual*, July 2009.

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to `errata@arm.com`. Give:

- the title
- the number, ARM DUI 0455G
- the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the AMBA-PV extensions. It contains the following section:

1.1 About the AMBA-PV extensions

The *AMBA-PV Extensions to TLM 2.0* (AMBA-PV) provides a mapping of AMBA buses on top of TLM 2.0:

- Dedicated to *Programmer's View* (PV), it focuses on high-level, functionally accurate, transaction modeling. Low-level signals such as, for example, channel handshake, are not important at that level.
- Standard for modeling of AMBA ACE, AXI, AHB and APB buses with TLM 2.0.
- Targeted at *Loosely-Timed* (LT) coding style of TLM 2.0, it includes blocking transport, *Direct Memory Interface* (DMI), and debug interfaces.
- Interoperable, it permits models using the mapped AMBA buses to work in an Accellera-compliant SystemC environment.

AMBA-PV classes and interfaces are layered on top of the TLM 2.0 library. AMBA-PV specializes TLM 2.0 classes and interfaces to handle AMBA buses control information such as secure, non-secure and privileged. In addition, AMBA-PV provides a framework that minimizes the effort required to write TLM 2.0 models that communicate over the AMBA buses.

AMBA buses add the following specific features to the TLM 2.0 *Generic Payload* (GP):

- addressing options support
- protection unit support
- cache support
- atomic accesses support
- quality of service (QoS) support
- multiple region support
- coherency support
- barrier transactions
- distributed virtual memory (DVM) support.

AMBA-PV extensions to the TLM 2.0 *Base Protocol* (BP) cover the following:

- definition of AMBA-PV extension and traits classes
- specialization of TLM 2.0 sockets and interfaces
- use of TLM 2.0 `b_transport()` blocking transport interface only.

In addition, AMBA-PV defines classes and interfaces for the modeling of side-band signals such as, for example, interrupts.

Chapter 2
AMBA-PV Extension

This chapter describes the AMBA-PV extension. It contains the following sections:

2.1 Introduction

AMBA-PV defines an extension, class `amba_pv_extension`, to the TLM 2.0 GP, class `tlm_generic_payload`. This extension class targets AMBA buses modeling, using an LT coding style, and features attributes for the modeling of:

- burst length, from 1 to 256 data transfers per burst
- burst transfer size of 8-1024 bits
- wrapping, incrementing, and non-incrementing burst types
- atomic operations, using exclusive or locked accesses

  Note

  It is recommended that locked accesses are only used to support legacy devices, because of their impact on the interconnect performance and their unavailability in AXI4 and ACE.

  The AMBA-PV bus decoder model does not currently support locked accesses.

- system-level caching and buffering control
- secure and privileged accesses
- quality of service (QoS) indication
- multiple regions
- cache coherency transactions (ACE-Lite)
- bi-directional cache coherency transactions (ACE)
- distributed virtual memory (DVM) transactions.

This extension does not model any of the following:

- separate address/control and data phases
- separate read and write data channels
- ability to issue multiple outstanding addresses
- out-of-order transaction completion
- optional extensions that cover signaling for low-power operation
- split transactions
- undefined-length bursts
- user-defined signals.

  Note

  Undefined-length bursts are specific to the AHB bus. They can be modeled as incrementing bursts of defined length, providing the master knows the total transfer length. AHB bus specifies a 1KB address boundary that bursts must not cross. This de-facto limits the length of an undefined-length burst.

It additionally supports unaligned burst start addresses and unaligned write data transfers using byte strobes.

AMBA-PV defines a new trait class `amba_pv_protocol_types` that features:

- support for most of the TLM 2.0 BP rules
- word length equals burst size
- no part-words
- byte enables on write transactions only
- byte enable length is a multiple of the burst size
- simulated endianness equals host endianness.

This class is used for the `TYPES` template parameter with TLM 2.0 classes and interfaces.

If using `amba_pv_protocol_types` with TLM 2.0 classes and interfaces, the following additional rules apply to the TLM 2.0 GP attributes:

- the data length attribute must be greater than or equal to the burst size times the burst length
- the streaming width attribute must be equal to the burst size for a fixed burst
- the byte enable pointer attribute must be `NULL` on read transactions
- if non zero, the byte enable length attribute shall be a multiple of the burst size on write transactions
- if the address attribute is not aligned on the burst size, only the address of the first burst beat must be unaligned, the subsequent beats addresses being aligned.

Note

This does not enforce any requirements on slaves for read transactions, and this must be represented with appropriate byte enables for write transactions.

The AMBA-PV extension must be used with AMBA-PV sockets, that is, sockets parameterized with the `amba_pv_protocol_types` traits class. This follows the rules set out in the section *Define a new protocol traits class containing a typedef for tlm_generic_payload* of the *OSCI TLM-2.0 Language Reference Manual*, July 2009. The AMBA-PV extension is a mandatory extension for the modeling of AMBA buses. For more information, see the section *Non-ignorable and mandatory extensions* in the same document.

2.2 Attributes and methods

The AMBA-PV extension classes contain a set of private attributes and a set of public access functions to get and set the values of these attributes.

This section aims at describing these attributes and functions.

2.2.1 Class definition

```
namespace amba_pv {
enum amba_pv_domain_t {
    AMBA_PV_NON_SHAREABLE   = 0x0,
    AMBA_PV_INNER_SHAREABLE = 0x1,
    AMBA_PV_OUTER_SHAREABLE = 0x2,
    AMBA_PV_SYSTEM          = 0x3
};

std::string amba_pv_domain_string(amba_pv_domain_t);

enum amba_pv_bar_t {
    AMBA_PV_RESPECT_BARRIER         = 0x0,
    AMBA_PV_MEMORY_BARRIER          = 0x1,
    AMBA_PV_IGNORE_BARRIER          = 0x2,
    AMBA_PV_SYNCHRONISATION_BARRIER = 0x3
};

std::string amba_pv_bar_string(amba_pv_bar_t);

enum amba_pv_snoop_t {
    AMBA_PV_READ_NO_SNOOP          = 0x0,
    AMBA_PV_READ_ONCE              = 0x0,
    AMBA_PV_READ_CLEAN             = 0x2,
    AMBA_PV_READ_NOT_SHARED_DIRTY  = 0x3,
    AMBA_PV_READ_SHARED            = 0x1,
    AMBA_PV_READ_UNIQUE            = 0x7,
    AMBA_PV_CLEAN_UNIQUE           = 0xB,
    AMBA_PV_CLEAN_SHARED           = 0x8,
    AMBA_PV_CLEAN_INVALID          = 0x9,
    AMBA_PV_MAKE_UNIQUE            = 0xC,
    AMBA_PV_MAKE_INVALID           = 0xD,
    AMBA_PV_WRITE_NO_SNOOP         = 0x0,
    AMBA_PV_WRITE_UNIQUE           = 0x0,
    AMBA_PV_WRITE_LINE_UNIQUE      = 0x1,
    AMBA_PV_WRITE_BACK             = 0x3,
    AMBA_PV_WRITE_CLEAN            = 0x2,
    AMBA_PV_EVICT                  = 0x4,
    AMBA_PV_BARRIER                = 0x0,
    AMBA_PV_DVM_COMPLETE           = 0xE,
    AMBA_PV_DVM_MESSAGE            = 0xF
};

std::string amba_pv_snoop_read_string(amba_pv_snoop_t,amba_pv_domain_t,amba_pv_bar_t);

std::string amba_pv_snoop_write_string(amba_pv_snoop_t,amba_pv_domain_t,amba_pv_bar_t);

class amba_pv_control {
    public:
        amba_pv_control();
        void set_id(unsigned int);
        unsigned int get_id() const;
        void set_privileged(bool = true);
        bool is_privileged() const;
        void set_non_secure(bool = true);
```

```
        bool is_non_secure() const;
        void set_instruction(bool = true);
        bool is_instruction() const;
        void set_exclusive(bool = true);
        bool is_exclusive() const;
        void set_locked(bool = true);
        bool is_locked() const;
        void set_bufferable(bool = true);
        bool is_bufferable() const;
        void set_cacheable(bool = true);
        bool is_cacheable() const;
        void set_read_allocate(bool = true);
        bool is_read_allocate() const;
        void set_write_allocate(bool = true);
        bool is_write_allocate() const;
        void set_modifiable(bool = true);
        bool is_modifiable() const;
        void set_read_other_allocate(bool = true);
        bool is_read_other_allocate() const;
        void set_write_other_allocate(bool = true);
        bool is_write_other_allocate() const;
        void set_qos(unsigned int);
        unsigned int get_qos() const;
        void set_region(unsigned int);
        unsigned int get_region() const;
        void set_snoop(amba_pv_snoop_t);
        amba_pv_snoop_t get_snoop() const;
        void set_domain(amba_pv_domain_t);
        amba_pv_domain_t get_domain() const;
        void set_bar(amba_pv_bar_t);
        amba_pv_bar_t get_bar() const;
};

enum amba_pv_resp_t {
    AMBA_PV_OKAY    = 0x0,
    AMBA_PV_EXOKAY  = 0x1,
    AMBA_PV_SLVERR  = 0x2,
    AMBA_PV_DECERR  = 0x3,
};

std::string amba_pv_resp_string(amba_pv_resp_t);

amba_pv_resp_t amba_pv_resp_from_tlm(tlm::tlm_response_status);

tlm::tlm_response_status amba_pv_resp_to_tlm(amba_pv_resp_t);

class amba_pv_response {
    public:
        amba_pv_response();
        amba_pv_response(amba_pv_resp_t);
        void set_resp(amba_pv_resp_t);
        amba_pv_resp_t get_resp() const;
        bool is_okay() const;
        void set_okay();
        bool is_exokay() const;
        void set_exokay();
        bool is_slverr() const;
        void set_slverr();
        bool is_decerr() const;
        void set_decerr();
        bool is_pass_dirty() const;
        void set_pass_dirty(bool=true);
        bool is_shared() const;
```

```
        void set_shared(bool=true);
        bool is_snoop_data_transfer() const;
        void set_snoop_data_transfer(bool=true);
        bool is_snoop_error() const;
        void set_snoop_error(bool=true);
        bool is_snoop_was_unique() const;
        void set_snoop_was_unique(bool=true);
        void reset();
};

enum amba_pv_dvm_message_t {
    AMBA_PV_TLB_INVALIDATE                        = 0x0,
    AMBA_PV_BRANCH_PREDICTOR_INVALIDATE           = 0x1,
    AMBA_PV_PHYSICAL_INSTRUCTION_CACHE_INVALIDATE = 0x2,
    AMBA_PV_VIRTUAL_INSTRUCTION_CACHE_INVALIDATE  = 0x3,
    AMBA_PV_SYNC                                  = 0x4,
    AMBA_PV_HINT                                  = 0x6
};

std::string amba_pv_dvm_message_string(amba_pv_dvm_message_t);

enum amba_pv_dvm_os_t {
    AMBA_PV_HYPERVISOR_OR_GUEST = 0x0,
    AMBA_PV_GUEST               = 0x2,
    AMBA_PV_HYPERVISOR          = 0x3
};

std::string amba_pv_dvm_os_string(amba_pv_dvm_os_t);

enum amba_pv_dvm_security_t {
    AMBA_PV_SECURE_AND_NON_SECURE = 0x0,
    AMBA_PV_SECURE_ONLY           = 0x2,
    AMBA_PV_NON_SECURE_ONLY       = 0x3
};

std::string amba_pv_dvm_security_string(amba_pv_dvm_security_t);

class amba_pv_dvm {
    public:
        amba_pv_dvm();
        void set_dvm_transaction(unsigned int);
        unsigned int get_dvm_transaction() const;
        void set_dvm_additional_address(sc_dt::uint64);
        bool is_dvm_additional_address_set() const;
        sc_dt::uint64 get_dvm_additional_address() const;
        void set_dvm_vmid(unsigned int);
        bool is_dvm_vmid_set() const;
        unsigned int get_dvm_vmid() const;
        void set_dvm_asid(unsigned int);
        bool is_dvm_asid_set() const;
        unsigned int get_dvm_asid() const;
        void set_dvm_virtual_index(unsigned int);
        bool is_dvm_virtual_index_set() const;
        unsigned int get_dvm_virtual_index() const;
        void set_dvm_completion(bool /* completion */ = true);
        bool is_dvm_completion_set() const;
        void set_dvm_message_type(amba_pv_dvm_message_t);
        amba_pv_dvm_message_t get_dvm_message_type() const;
        void set_dvm_os(amba_pv_dvm_os_t);
        amba_pv_dvm_os_t get_dvm_os() const;
        void set_dvm_security(amba_pv_dvm_security_t);
        amba_pv_dvm_security_t get_dvm_security() const;
        void reset();
```

```
};

enum amba_pv_burst_t {
    AMBA_PV_FIXED = 0,
    AMBA_PV_INCR,
    AMBA_PV_WRAP
};

std::string amba_pv_burst_string(amba_pv_burst_t);

class amba_pv_extension:
    public tlm::tlm_extension<amba_pv_extension>,
    public amba_pv_control
    public amba_pv_dvm {
    public:
        amba_pv_extension();
        amba_pv_extension(size_t, const amba_pv_control *);
        amba_pv_extension(size_t,
                          size_t,
                          const amba_pv_control *,
                          amba_pv_burst_t);
        virtual tlm::tlm_extension_base * clone() const;
        virtual void copy_from(tlm::tlm_extension_base const &);
        void set_length(unsigned int);
        unsigned int get_length() const;
        void set_size(unsigned int);
        unsigned int get_size() const;
        void set_burst(amba_pv_burst_t);
        amba_pv_burst_t get_burst() const;
        void set_resp(amba_pv_resp_t);
        amba_pv_resp_t get_resp() const;
        bool is_okay() const;
        void set_okay();
        bool is_exokay() const;
        void set_exokay();
        bool is_slverr() const;
        void set_slverr();
        bool is_decerr() const;
        void set_decerr();
        bool is_pass_dirty() const;
        void set_pass_dirty(bool);
        bool is_shared() const;
        void set_shared(bool);
        bool is_snoop_data_transfer() const;
        void set_snoop_data_transfer(bool=true);
        bool is_snoop_error() const;
        void set_snoop_error(bool=true);
        bool is_snoop_was_unique() const;
        void set_snoop_was_unique(bool=true);
        void set_response_array_ptr(amba_pv_response*);
        amba_pv_response* get_response_array_ptr();
        void set_response_array_complete(bool=true);
        bool is_response_array_complete();
        void reset();
        void reset(unsigned int,
                          const amba_pv_control *);
        void reset(unsigned int,
                          unsigned int,
                          const amba_pv_control *,
                          amba_pv_burst_t);
};

sc_dt::uint64 amba_pv_address(const sc_dt::uint64 &,
```

```
                                    unsigned int,
                                    unsigned int,
                                    amba_pv_burst_t,
                                    unsigned int);
}
```

The amba_pv_control base class includes attributes that relate to system-level caches, protection units, atomic accesses, QoS, multiple regions, cache coherency, barrier transactions and DVM. The amba_pv_control class is also used as an argument to the user layer interface methods. See *User layer* on page 3-4.

2.2.2 Constructors, copying and addressing

- The default constructors must set the AMBA-PV extension attributes to their default values, as defined in the following subsections.
- The constructor `amba_pv_extension(size_t, const amba_pv_control *)` must set the burst size attribute value to the value passed as argument, and must set the attributes values of the `amba_pv_control` base class to the values of the attributes of the `amba_pv_control` object whose address is passed as argument, if not `NULL`.
- The constructor `amba_pv_extension(size_t, size_t, const amba_pv_control *, amba_pv_burst_t)` must set the burst size attribute value to the value passed as argument, must set the burst length attribute value to the value passed as argument, must set the burst type attribute value to the value passed as argument, and must set the attributes values of the `amba_pv_control` base class to the values of the attributes of the `amba_pv_control` object whose address is passed as argument, if not `NULL`
- The virtual method `clone()` must create a copy of the AMBA-PV extension object, including all its attributes.
- The virtual method `copy_from()` must modify the current AMBA-PV extension object by copying the attributes of another AMBA-PV extension object.
- The global function `amba_pv_address()` must compute the address of a transfer or beat within a burst given the transaction address, burst length, burst size, burst type, and beat number.

2.2.3 Default values and modifiability of attributes

- The value of every AMBA-PV extension attribute has to be set by the master prior to passing the transaction object through an interface method call.

- Table 2-1 lists default values and modifiability of the AMBA-PV extension attributes:

Table 2-1 Default values and modifiability of attributes

Attribute	Default value	Modifiable by interconnect	Modifiable by slave
Burst length	1	No	No
Burst size	8	No	No
Burst type	AMBA_PV_INCR	No	No
ID	0	Yes	No
Privileged	false	No	No
Non-secure	false	No	No
Instruction	false	No	No
Exclusive	false	Yes [a]	No
Locked	false	No	No
Bufferable	false	No	No
Modifiable/Cacheable[b]	false	No	No
Read allocate	false	No	No
Write allocate	false	No	No
Read other allocate	false	No	No
Write other allocate	false	No	No
QoS	0	Yes	No
Region	0	No	No
Domain	AMBA_PV_NON_SHAREABLE	No	No
Snoop	AMBA_PV_READ_NO_SNOOP[c]	No	No
Bar	AMBA_PV_RESPECT_BARRIER	No	No
Response	AMBA_PV_OKAY	Yes	Yes
PassDirty	false	Yes	Yes
IsShared	false	Yes	Yes
DataTransfer[d]	false	Yes	Yes
Error[d]	false	Yes	Yes
WasUnique[d]	false	Yes	Yes
ResponseArray	null	No	No
ResponseArray complete	false	Yes	Yes

a. As in the case of an exclusive monitor that flattens the exclusive access before passing it downstream.

b. The Modifiable attribute is identical to the Cacheable attribute but has been renamed in AXI4 to better describe the required functionality.
c. `AMBA_PV_WRITE_NO_SNOOP` and `AMBA_PV_READ_NO_SNOOP` have the same encoding representation.
d. These attributes are only valid responses to upstream snoops, typically from interconnect to master.

- If an AMBA-PV extension object is re-used, the modifiability rules in Table 2-1 on page 2-9 cease to apply at the end of the lifetime of the corresponding transaction instance. The rules re-apply if the AMBA-PV extension object is re-used for a new transaction.
- After adding the AMBA-PV extension to a transaction object and passing that transaction object as an argument to an interface method call (`b_transport()`, `get_direct_mem_ptr()`, or `transport_dbg()`), the master must not modify any of the AMBA-PV extension attributes during the lifetime of the transaction.
- An interconnect can modify the ID attribute, but only before passing the corresponding transaction as an argument to an interface method call (`b_transport()`, `get_direct_mem_ptr()`, or `transport_dbg()`) on the forward path. When the interconnect has passed a pointer to the AMBA-PV extension to a downstream model, it is not permitted to modify the ID of that extension object again during the entire lifetime of the corresponding transaction.
- As a consequence of the above rule, the ID attribute is valid immediately on entering any of the method calls `b_transport()`, `get_direct_mem_ptr()`, or `transport_dbg()`. Following the return from any of those calls, the ID attribute has the value set by the interconnect furthest downstream.
- The interconnect and slave can modify the response attribute at any time between having first received the corresponding transaction object and the time at which they pass a response upstream by returning control from the `b_transport()`, `get_direct_mem_ptr()`, or `transport_dbg()` methods.
- The master can assume it is seeing the value of the AMBA-PV extension response attribute only after it has received a response for the corresponding transaction.
- If the AMBA-PV extension is used for the direct memory or debug transport interfaces, the modifiability rules given here must apply to the appropriate attributes of the AMBA-PV extension, namely the ID, privileged, non-secure, and instruction attributes.

2.2.4 Burst length attribute

- The method `set_length()` must set the burst length attribute to the value passed as argument. The method `get_length()` must return the current value of the burst length attribute.
- The burst length attribute specifies the number of data transfers that occur within this burst. It must have a value between 1 and 256 for defined-length burst. Additional restrictions apply depending on the value of the burst type attribute (See *Extension checks* on page 6-5).
- The default value of the burst length attribute must be 1, for single transfer.
- The burst length attribute is specific to the AXI, ACE and AHB buses. It is ignored for transactions modeling transfers on the APB bus.
- The maximum burst length value for AXI3 and AHB buses is 16, and the maximum value for AXI4 and ACE buses is 256.

2.2.5 Burst size attribute

- The method `set_size()` must set the burst size attribute to the value passed as argument. The method `get_size()` must return the current value of the burst size attribute.
- The burst size attribute specifies the maximum number of data bytes to transfer in each beat, or data transfer, within a burst. It must have a value of 1, 2, 4, 8, 16, 32, 64 or 128.
- The value of the burst size attribute must be less than or equal to `BUSWIDTH` / 8, where `BUSWIDTH` is the template parameter of the socket classes from AMBA-PV (or classes derived from these) and expressed in bits.
- The default value of the burst size attribute must be 8, for 64 bits wide transfer.
- The burst size attribute is specific to the AXI, ACE and AHB buses. It is ignored for transactions modeling transfers on the APB bus.

2.2.6 Burst type attribute

- The method `set_burst()` must set the burst type attribute to the value passed as argument. The method `get_burst()` must return the current value of the burst type attribute.
- A transaction with a burst type attribute value of `AMBA_PV_WRAP` must have an aligned address.
- The default value of the burst type attribute must be `AMBA_PV_INCR`, for incrementing burst.
- The burst type attribute is specific to the AXI, ACE and AHB buses. It is ignored for transactions modeling transfers on the APB bus.

2.2.7 ID attribute

- The method `set_id()` must set the ID attribute to the value passed as argument. The method `get_id()` must return the current value of the ID attribute.
- The ID attribute is mainly used for exclusive accesses. The ID attribute must be set by the master originating the transaction. The interconnect must modify the ID attribute to ensure its uniqueness across all its masters before passing the transaction to the addressed slave.
- The default value of the ID attribute must be 0.
- The ID attribute is specific to the AXI, ACE and AHB buses. It is ignored for transactions modeling transfers on the APB bus.

2.2.8 Privileged attribute

- The method `set_privileged()` must set the privileged attribute to the value passed as argument. The method `is_privileged()` must return the current value of the privileged attribute.
- The privileged attribute enables masters to indicate their processing mode. A privileged transaction typically has a greater level of access within the system.
- The default value of the privileged attribute must be `false`.
- The privileged attribute is specific to the AXI, ACE and AHB buses. It is ignored for transactions modeling transfers on the APB bus.

2.2.9 Non-secure attribute

- The method `set_non_secure()` must set the non-secure attribute to the value passed as argument. The method `is_non_secure()` must return the current value of the non-secure attribute.
- The non-secure attribute enables differentiating between secure and non-secure transactions.
- The default value of the non-secure attribute must be `false`.
- The non-secure attribute is specific to the AXI and ACE buses. It is ignored for transactions modeling transfers on the AHB and APB buses.

2.2.10 Exclusive attribute

- The method `set_exclusive()` must set the exclusive attribute to the value passed as argument. The method `is_exclusive()` must return the current value of the exclusive attribute.
- The exclusive attribute selects exclusive access, and the response attribute (see Table 2-3 on page 2-19) indicates the success or failure of the exclusive access.
- The AMBA-PV package provides with an exclusive monitor model (See *Exclusive monitor* on page 3-8) that supports exclusive access and that can be added before your slave. It removes the requirement for your slave to model additional logic to support exclusive access.
- It is recommended that masters do not use the direct memory interface for exclusive accesses.
- The address of an exclusive access must be aligned to the total number of bytes in the transaction as determined by the value of the burst size attribute multiplied by the value of the burst length attribute.
- The number of bytes to be transferred in an exclusive access must be a power of 2 and less than or equal to 128.
- It is recommended that every exclusive write has an earlier outstanding exclusive read with the same value for the ID attribute.
- It is recommended that the value of the address, burst size, and burst length attributes of an exclusive write with a given value for the ID attribute is the same as the value of the address, burst size, and burst length attributes of the preceding exclusive read with the same value for the ID attribute.
- An `AMBA_PV_EXOKAY` value for the response attribute can only be given to an exclusive access.
- The exclusive attribute must not have the value `true` together with the locked attribute.
- The default value of the exclusive attribute must be `false`.
- The exclusive attribute is specific to the AXI and ACE buses. It is ignored for transactions modeling transfers on the AHB and APB buses.

2.2.11 Locked attribute

- The method `set_locked()` must set the locked attribute to the value passed as argument. The method `is_locked()` must return the current value of the locked attribute.

- Locked transactions, that is transactions for which the associated locked attribute has the value `true`, require that the interconnect prevents any other transactions occurring while the locked sequence is in progress and can thus have an impact on the interconnect performance.
- It is recommended that locked accesses are only used to support legacy devices. Locked transactions are currently not supported by the AMBA-PV bus decoder.
- The locked attribute must not have the value `true` together with the exclusive attribute.
- The default value of the locked attribute must be `false`.
- The locked attribute is specific to the AXI3 and AHB buses. It is ignored for transactions modeling transfers on the APB, AXI4 and ACE buses.

2.2.12 Bufferable attribute

- The method `set_bufferable()` must set the bufferable attribute to the value passed as argument. The method `is_bufferable()` must return the current value of the bufferable attribute.
- The bufferable attribute specifies whether or not the associated transaction is bufferable.
- A bufferable transaction can be delayed in reaching its final destination. This is usually only relevant to writes.
- The default value of the bufferable attribute must be `false`.
- The bufferable attribute is specific to the AXI and AHB buses. It is ignored for transactions modeling transfers on the APB bus.

2.2.13 Modifiable/Cacheable attribute

- The methods `set_modifiable()` and `set_cacheable()` must set the modifiable attribute to the value passed as argument. The methods `is_modifiable()` and `is_cacheable()` must return the current value of the modifiable attribute.
- The modifiable attribute specifies whether the associated transaction is modifiable.
- For write transactions, a number of different writes can be merged together. For read transactions, a location can be pre-fetched or can be fetched only once for multiple reads. To determine if a transaction must be cached, use this attributes with the read allocate (see *Read allocate attribute*) and write allocate (see *Write allocate attribute* on page 2-14) attributes.
- The default value of the modifiable attribute must be `false`.
- The modifiable attribute is specific to the AXI and AHB buses. It is ignored for transactions modeling transfers on the APB bus.
- The cacheable attribute used by the AXI3 and AHB buses has been renamed the modifiable attribute for AXI4 and ACE to better describe the required function of the attribute. The actual functionality is unchanged.

2.2.14 Read allocate attribute

- The method `set_read_allocate()` must set the read allocate attribute to the value passed as argument. The method `is_read_allocate()` must return the current value of the read allocate attribute.

- The read allocate attribute specifies whether or not this transaction must be allocated if it is a read and it misses in the cache.
- The value of this attribute must not be set to `true` if the value of the modifiable attribute is set to `false`.
- The default value of the read allocate attribute must be `false`.
- The read allocate attribute is specific to the AXI and ACE buses. It is ignored for transactions modeling transfers on the AHB and APB buses.

2.2.15 Write allocate attribute

- The method `set_write_allocate()` must set the write allocate attribute to the value passed as argument. The method `is_write_allocate()` must return the current value of the write allocate attribute.
- The write allocate attribute specifies whether or not this transaction must be allocated if it is a write and it misses in the cache.
- The value of this attribute must not be set to `true` if the value of the modifiable attribute is set to `false`.
- The default value of the write allocate attribute must be `false`.
- The write allocate attribute is specific to the AXI and ACE buses. It is ignored for transactions modeling transfers on the AHB and APB buses.

2.2.16 Read other allocate attribute

- The read other allocate attribute indicates that the location could have been previously allocated in the cache because of a write transaction or because of the actions of another master.
- The value of this attribute must not be set to true if the value of the modifiable attribute is set to `false`.
- The method `set_read_other_allocate()` sets the read other allocate attribute to the value passed as argument. The method `is_read_other_allocate()` returns the current value of the read other allocate attribute.
- The default value of the read other allocate attribute is `false`.
- The read other allocate attribute is specific to the AXI4 and ACE buses. It is ignored for transactions modeling transfers on the AHB and APB buses.
- To maintain compatibility with AXI3, this attribute may also be accessed using the write allocate attribute methods `set_write_allocate()` and `is_write_allocate()`.

2.2.17 Write other allocate attribute

- The write other allocate attribute indicates that the location could have been previously allocated in the cache because of a read transaction or because of the actions of another master.
- The method `set_write_other_allocate()` sets the write other allocate attribute to the value passed as argument. The method `is_write_other_allocate()` returns the current value of the write other allocate attribute.

- The value of this attribute must not be set to true if the value of the modifiable attribute is set to `false`.
- The default value of the write other allocate attribute is `false`.
- The write other allocate attribute is specific to the AXI4 and ACE buses. It is ignored for transactions modeling transfers on the AHB and APB buses.
- To maintain compatability with AXI3, this attribute may also be accessed using the read allocate attribute methods `set_read_allocate()` and `is_read_allocate()`.

2.2.18 Quality of Service (QoS) attribute

- The QoS attribute may be used to support Quality of Service schemes. The bus protocol does not specify the exact use of the QoS identifier but recommend that it is used as a priority indicator.
- The method `set_qos()` sets the QoS attribute to the value passed as argument. The method `get_qos()` returns the current value of the QoS attribute.
- The default value of the QoS attribute is `0`, which indicates that the interface is not participating in any `QoS` scheme.
- The QoS attribute is specific to the AXI4 and ACE buses. It is ignored for transactions modeling transfers on the AXI3, AHB and APB buses.
- For AXI4 and ACE the QoS indicator attribute value must be between 0 and 15 inclusive.

2.2.19 Region attribute

- The region attribute may be used to support multiple region interfaces and is used to uniquely identify a region.
- The method `set_region()` sets the region attribute to the value passed as argument. The method `get_region()` returns the current value of the region attribute.
- The default value of the region attribute is `0`.
- The region attribute is specific to the AXI4 and ACE buses. It is ignored for transactions modeling transfers on the AXI3, AHB and APB buses.
- For AXI4 and ACE the region indicator attribute value must be between 0 and 15 inclusive.

2.2.20 Domain attribute

- The domain attribute indicates the shareability domain for a transaction.
- The method `set_domain()` sets the domain attribute to the value passed as argument. The method `get_domain()` returns the current value of the domain attribute.
- The default value of the domain attribute is `AMBA_PV_NON_SHAREABLE`.
- The domain attribute is specific to ACE buses. It is ignored for transactions modeling transfers on the AXI, AHB and APB buses.
- The encoding of the domain attribute value exactly matches the encoding used on the ACE channels AWDOMAIN and ARDOMAIN.

2.2.21 Snoop attribute

- The snoop attribute specifies the transaction type for shareable transactions.
- The method `set_snoop()` sets the snoop attribute to the value passed as argument. The method `get_snoop()` returns the current value of the snoop attribute.
- The default value of the snoop attribute is encoded as `0` which for read transactions represents `AMBA_PV_READ_NO_SNOOP` and for write transactions `AMBA_PV_WRITE_NO_SNOOP`.
- The meaning of a given snoop attribute value encoding is dependant on the domain and bar attribute values and whether the transaction is a read or a write.
- The snoop attribute is specific to ACE buses. It is ignored for transactions modeling transfers on the AXI, AHB and APB buses.
- The encoding of the snoop attribute value exactly matches the encoding used on the ACE channels AWSNOOP and ARSNOOP.

2.2.22 Bar attribute

- The method `set_bar()` sets the bar attribute to the value passed as argument. The method `get_bar()` returns the current value of the bar attribute.
- The bar attribute indicates barrier information for the transaction.
- The default value of the domain attribute is `AMBA_PV_RESPECT_BARRIER`.
- The bar attribute is specific to ACE buses. It is ignored for transactions modeling transfers on the AXI, AHB and APB buses.
- The encoding of the bar attribute value exactly matches the encoding used on the ACE channels AWBAR and ARBAR.

2.2.23 DVM messages

- To provide a Programmer's View model of Distributed Virtual Memory (DVM) transactions, the AMBA-PV extension class contains a set of private attributes and a set of public access methods for DVM messages.
- A given transaction only represents a DVM message if the snoop attribute is set to AMBA_PV_DVM_MESSAGE.
- DVM messages are specific to ACE and ACE-Lite buses. They are ignored for transactions modeling transfers on the AXI, AHB and APB buses.

DVM default values

Table 2-2 lists the default values for the AMBA-PV extension attributes for DVM:

Table 2-2 DVM default values

Attribute	Default value	Default set status
VMID	`0`	`false`
ASID	`0`	`false`
Virtual Index	`0`	`false`
Completion	`false`	-

Table 2-2 DVM default values (continued)

Attribute	Default value	Default set status
Message type	AMBA_PV_TLB_INVALIDATE	-
Operating system	AMBA_PV_HYPERVISOR_OR_GUEST	-
Security	AMBA_PV_SECURE_AND_NON_SECURE	-
Additional address	`0`	`false`
DVM transaction	`0`	-

DVM VMID attribute

- The VMID attribute defines the Virtual Machine Identifier for some DVM operations.
- The method `is_dvm_vmid_set()` returns true if the VMID attribute has been set. If the VMID attribute has not been set then the VMID attribute value should not be used.
- The method `get_dvm_vmid()` returns the current value of the VMID attribute. The method `set_dvm_vmid()` sets the value of the VMID attribute.
- By default the VMID attribute is not set and the default value of the VMID attribute is `0`.

DVM ASID attribute

- The ASID attribute defines the Address Space Identifier for some DVM operations.
- The method `is_dvm_asid_set()` returns true if the ASID attribute has been set. If the ASID attribute has not been set then the ASID attribute value should not be used.
- The method `get_dvm_asid()` returns the current value of the ASID attribute. The method `set_dvm_asid()` sets the value of the ASID attribute.
- By default the ASID attribute is not set and the default value of the ASID attribute is `0`.

DVM Virtual Index attribute

- The Virtual Index attribute can be used as part of the physical address by physical instruction cache invalidate DVM messages.
- The method `is_dvm_virtual_index_set()` returns true if the Virtual Index attribute has been set. If the Virtual Index attribute has not been set then the Virtual Index attribute value should not be used.
- The method `get_dvm_virtual_index()` returns the current value of the Virtual_index attribute. The method `set_dvm_virtual_index()` sets the value of the Virtual Index attribute.
- By default the Virtual Index attribute is not set and the default value of the Virtual Index attribute is `0`.

DVM Completion attribute

- The Completion attribute identifies whether completion is required for DVM Sync messages.
- The method `is_dvm_completion_set()` returns true if the Completion attribute has been set. The method `set_dvm_completion()` sets the value of the completion attribute.

- By default the Completion attribute has the value `false`.

DVM Message type attribute

- The Message type attribute specifies the required DVM operation.
- The method `get_dvm_message_type()` returns the current value of the Message type attribute. The method `set_dvm_message_type()` sets the value of the message type attribute.
- By default the Message type attribute has the value `AMBA_PV_TLB_INVALIDATE`.

DVM Operating system attribute

- The Operating system attribute specifies the operating system that the DVM operation applies to.
- The method `get_dvm_os()` returns the current value of the Operating system attribute. The method `set_dvm_os()` sets the value of the operating system attribute.
- By default the Operating system attribute has the value `AMBA_PV_HYPERVISOR_OR_GUEST`.

DVM Security attribute

- The Security attribute specifies how the DVM operation applies to the secure and non-secure worlds.
- The method `get_dvm_security()` returns the current value of the security attribute. The method `set_dvm_security()` sets the value of the security attribute.
- By default the security attribute has the value `AMBA_PV_SECURE_AND_NON_SECURE`.

DVM Additional address attribute

- The Additional address attribute defines the additional address required by some DVM operations.
- The method `is_dvm_additional_address_set()` returns true if the Additional address attribute has been set. If the Additional address attribute has not been set then the Additional address attribute value should not be used.
- The method `get_dvm_additional_address()` returns the current value of the Additional address attribute. The method `set_dvm_additional_address()` sets the value of the Additional address attribute.
- By default the Additional address attribute is not set and the default value of the Additional address attribute is `0`.

DVM transaction encoding

- For ACE buses the DVM attributes are packed and encoded into the least significant 32-bits of the address channel.
- The method `get_dvm_transaction()` returns the current value of the VMID, ASID, Virtual Index, Completion, Message type, Operating system and Security attributes as they would be packed and encoded on the address channel.
- The method `set_dvm_transaction()` sets the value of the VMID, ASID, Virtual Index, Completion, Message type, Operating system and Security attributes using a single 32-bit value encoded as the attributes would be packed and encoded on the address channel.

2.2.24 Response attribute

- The method `set_resp()` must set the response attribute to the value passed as argument. The method `get_resp()` must return the current value of the response attribute.
- The method `is_okay()` must return `true` if and only if the current value of the response attribute is `AMBA_PV_OKAY`. The method `set_okay()` must set the value of the response attribute to `AMBA_PV_OKAY`.
- The method `is_exokay()` must return `true` if and only if the current value of the response attribute is `AMBA_PV_EXOKAY`. The method `set_exokay()` must set the value of the response attribute to `AMBA_PV_EXOKAY`.
- The method `is_slverr()` must return `true` if and only if the current value of the response attribute is `AMBA_PV_SLVERR`. The method `set_slverr()` must set the value of the response attribute to `AMBA_PV_SLVERR`.
- The method `is_decerr()` must return `true` if and only if the current value of the response attribute is `AMBA_PV_DECERR`. The method `set_decerr()` must set the value of the response attribute to `AMBA_PV_DECERR`.
- Table 2-3 lists the four response values defined by the AMBA-PV extension.

Table 2-3 AMBA-PV response interpretation

Response	Interpretation
`AMBA_PV_OKAY`	Indicates that a normal access has been successful. It also indicates an exclusive access failure.
`AMBA_PV_EXOKAY`	Indicates that either the read or write portion of an exclusive access has been successful.
`AMBA_PV_SLVERR`	Indicates that the access has reached the slave successfully, but the slave returned an error condition to the originating master.
`AMBA_PV_DECERR`	Indicates that there is no slave at the transaction address. This is typically generated by an interconnect component.

- The response attribute must be set to `AMBA_PV_OKAY` by the master, and might be overwritten by the slave or the interconnect.
- If the slave is able to execute the transaction, it must set the response attribute to `AMBA_PV_OKAY`. If not, the slave must set the response attribute to `AMBA_PV_SLVERR`.
- If the interconnect is able to pass the transaction downstream to the addressed slave, it must not overwrite the response attribute. If not, the interconnect must set the response attribute to `AMBA_PV_DECERR`.
- The default value of the response attribute must be `AMBA_PV_OKAY`.
- The slave or interconnect is responsible for setting the response attribute before returning control from the `b_transport()` method of the TLM 2.0 blocking transport interface.
- It is recommended that the master always checks the value of the response attribute after the completion of the transaction.
- The global function `amba_pv_resp_string()` must return the response value passed as argument as a text string.

- The global function `amba_pv_resp_from_tlm()` must translate the TLM 2.0 response status value passed as argument into an AMBA-PV response value. The global function `amba_pv_resp_to_tlm()` must translate the AMBA-PV response value passed as argument into a TLM 2.0 response status value. Those translations must be performed according to Table 2-4.

Table 2-4 Translation between AMBA-PV response and TLM 2.0 response status

AMBA-PV response	TLM 2.0 response status
AMBA_PV_OKAY	TLM_OK_RESPONSE
AMBA_PV_EXOKAY	TLM_OK_RESPONSE [a]
AMBA_PV_SLVERR	TLM_GENERIC_ERROR_RESPONSE TLM_COMMAND_ERROR_RESPONSE TLM_BURST_ERROR_RESPONSE TLM_BYTE_ENABLE_ERROR_RESPONSE
AMBA_PV_DECERR	TLM_INCOMPLETE_RESPONSE TLM_ADDRESS_ERROR_RESPONSE

a. The exclusive attribute of the associated transaction must have a value of `true`.

2.2.25 ACE response attributes PassDirty and IsShared

- On ACE and ACE-Lite buses the additional response attributes PassDirty and IsShared are supported.
- When true the PassDirty attribute indicates that before the snoop process, the cache line was held in a Dirty state and the responsibility for writing the cache line back to memory is being passed to the initiating master or interconnect.
- The method `is_pass_dirty()` returns the current value of the response PassDirty signal. The method `set_pass_dirty()` sets the value of the PassDirty attribute.
- The default value of the PassDirty attribute is `false`.
- When true the IsShared attribute indicates that the snooped cache retains a copy of the cache line after the snoop process has completed.
- The method `is_shared()` returns the current value of the response IsShared attribute. The method `set_shared()` sets the value of the IsShared attribute.
- The default value of the IsShared attribute is `false`.

2.2.26 ACE snoop response attributes DataTransfer, Error and WasUnique

- On ACE buses additional snoop response attributes DataTransfer, Error and WasUnique are supported.
- When `true` the DataTransfer attribute indicates that the snoop response includes a transfer of data.
- The method `is_snoop_data_transfer()` returns the current value of the DataTransfer attribute. The method `set_snoop_data_transfer()` sets the value of the DataTransfer attribute.
- The default value of the DataTransfer attribute is `false`.

- When `true` the Error attribute indicates that the snooped cache line is in error.
- The method `is_snoop_error()` returns the current value of the Error attribute. The method `set_snoop_error()` sets the value of the Error attribute.
- The default value of the Error attribute is `false`.
- When `true` the WasUnique attribute indicates that the snooped cache line was held in a Unique state before the snoop process.
- The method `is_snoop_was_unique()` returns the current value of the snoop response WasUnique attribute. The method `set_snoop_was_unique()` sets the value of the WasUnique attribute.
- The default value of the WasUnique attribute is `false`.

2.2.27 Response array attribute

- The response array provides an alternative path for slaves to return response status; with a separate response status for each beat of a burst transaction.
- The method `get_response_array_ptr()` returns a pointer to the response array or null if the master has not set an array response pointer. The method `set_response_array_ptr()` sets a pointer to a response array.
- The method `set_response_array_complete()` is used by the slave to set the response array completion flag that when true indicates that the elements of the response array have been set with response data. The method `is_response_array_complete()` returns the status of the response array completion flag.
- If a response array is going to be made available it is the responsibility of the master to set the response array pointer. The size of the response array must be at least as large as the burst length attribute.
- A slave can choose to use the response attribute to report response status with a single response for the entire transaction even if a response array has been made available. But a slave can also optionally check for a response array and if an array pointer is available set the response status in the response array instead of using the response attribute. The slave must not set elements of the response array beyond the value of the burst length attribute.
- If a slave uses the response array it must set the response array completion flag to `true`.
- The master reads response status from the response attribute unless it has both set an array response pointer and the slave has set the response array completion status to `true`.

Response array element attributes

Table 2-5 lists the AMBA-PV response array element attributes:

Table 2-5 AMBA-PV response array element attributes

Attribute	Default value	Set method(s)	Get method(s)
Response	AMBA_PV_OK	set_resp(), set_okay(), set_exokay(), set_slverr(), set_decerr()	get_resp(), is_okay(), is_exokay(), is_slverr(), is_decerr()
PassDirty	false	set_pass_dirty()	is_pass_dirty()
IsShared	false	set_is_shared()	is_shared()
DataTransfer	false	set_snoop_data_transfer()	is_snoop_data_transfer()
Error	false	set_snoop_error()	is_snoop_error()
WasUnique	false	set_snoop_was_unique()	is_snoop_was_unique()

The attributes listed in Table 2-5 have the same semantics and accessors as the equivalent response attributes documented in *Response attribute* on page 2-19, *ACE response attributes PassDirty and IsShared* on page 2-20 and *ACE snoop response attributes DataTransfer, Error and WasUnique* on page 2-20.

2.2.28 Data organization

- In general, the organization of the AMBA-PV data array is in “bus order”, independent of the organization of local storage within the master or the slave.
- The contents of the data and byte enable arrays must be interpreted using the burst size attribute of the AMBA-PV extension. The size of a transferred word, or beat, within a transaction, is defined by the burst size attribute. The data array must not contain part-word, even when the transaction address is unaligned.
- The word boundaries within the data and byte enable arrays must be address-aligned, that is, they must fall on addresses that are integer multiples of the burst size. The data length attribute must be greater than or equal to the burst size times the burst length.
- The local address of a word or beat within the data array is given by the amba_pv_address() function as follows:

  ```
  amba_pv_address(address, burst_length, burst_size, burst_type, N);
  ```

  where N denotes the beat number as in 1-16.

2.2.29 Direct memory interface

- In the case of the AMBA-PV protocol, the AMBA-PV extension attributes that further indicate the address of the DMI access requested are the ID, privileged, non-secure, and instruction attributes.
- The ID attribute must be set by the master to further indicate the address of the DMI access requested.
- The privileged, non-secure, and instruction attributes must be set by the master to further indicate the address of the DMI access requested, and must not be modified by any interconnect or slave component.

- The slave can service DMI requests differently depending on the value of other AMBA-PV extension attributes. It is recommended that all AMBA-PV extension attributes are set by the master before requesting DMI access.

2.2.30 Debug transport interface

- In the case of the AMBA-PV protocol, the AMBA-PV extension attributes that further indicate the address of the debug access are the ID, privileged, non-secure, and instruction attributes.
- The ID attribute must be set by the master to further indicate the address of the debug access.
- The privileged, non-secure, and instruction attribute must be set by the master to further indicate the address of the debug access, and must not be modified by any interconnect or slave component.
- The slave or interconnect components can ignore all other attributes of the AMBA-PV extension.

2.3 AMBA signal mapping

Table 2-6, Table 2-7, Table 2-8 on page 2-25, Table 2-9 on page 2-25 and Table 2-10 on page 2-25 list the relationships between the AMBA hardware signals and the private attributes of the AMBA-PV extension and the TLM 2.0 Generic Payload:

Table 2-6 Address channels

Signal	Description	Variable
AxID	ID	`amba_pv_control::m_id`
AxADDR	Address	`tlm_generic_payload::m_address`
AxADDR	DVM message attributes	`amba_pv_extension::m_dvm_transaction`
AxLEN	Burst length	`amba_pv_extension::m_length`
AxSIZE	Burst size	`amba_pv_extension::m_size`
AxBURST	Burst type	`amba_pv_extension::m_burst`
AxLOCK	Lock type	`amba_pv_control::m_exclusive` `amba_pv_control::m_locked`

AxCACHE	Cache type	`amba_pv_control::m_bufferable`
		`amba_pv_control::m_modifiable`
		`amba_pv_control::m_axcache_allocate_bit2`
		`amba_pv_control::m_axcache_allocate_bit3`

AxPROT	Protection type	`amba_pv_control::m_privileged`
		`amba_pv_control::m_non_secure`
		`amba_pv_control::m_instruction`

AxQOS	Quality of service type	`amba_pv_control::m_qos`
AxREGION	Region type	`amba_pv_control::m_region`
AxDOMAIN	Domain type	`amba_pv_control::m_domain`
AxSNOOP	Snoop type	`amba_pv_control::m_snoop`
AxBAR	Barrier type	`amba_pv_control::m_bar`

Table 2-7 Write data and response channels

Signal	Description	Variable
WID, **BID**	ID	`amba_pv_control::m_id`
WDATA	Write data	`tlm_generic_payload::m_data` `tlm_generic_payload::m_length`
WSTRB	Write strobes	`tlm_generic_payload::m_byte_enable` `tlm_generic_payload::m_byte_enable_length`
BRESP	Write response	`tlm_generic_payload::m_response_status` `amba_pv_extension::m_response`

Table 2-8 Read data channels

Signal	Description	Variable
RID	ID	`amba_pv_extension::m_id`
RDATA	Read data	`tlm_generic_payload::m_data` `tlm_generic_payload::m_length`
RRESP	Read response	`tlm_generic_payload::m_response_status` `amba_pv_extension::m_response`

Table 2-9 Snoop data channels

Signal	Description	Variable
CDDATA	Snoop data	`tlm_generic_payload::m_data` `tlm_generic_payload::m_length`
CRRESP	Snoop response	`tlm_generic_payload::m_response_status` `amba_pv_extension::m_response`

Table 2-10 Unmapped signals

Signal	Description	Variable
xVALID	Address/data/response valid	Not applicable at PV level
xREADY	Address/data/response ready	Not applicable at PV level
xLAST	Read/write last	Not applicable at PV level
xACK	Read/Write acknowledge	Not applicable at PV level
xUSER	User defined signals	Use is not recommended

Note

The `tlm_generic_payload::m_length` attribute must be greater than or equal to `amba_pv_addressing::m_size` times `amba_pv_addressing::m_length`.

Note

For fixed bursts, the `tlm_generic_payload::m_streaming_width` attribute holds the same information as the `amba_pv_addressing::m_size` attribute.

2.4 Mapping for AMBA buses

The control signals mapping for AXI, ACE and AHB buses is listed in Table 2-11. The APB bus does not use these control signals.

Table 2-11 Signals mapping for `amba_pv_control`

amba_pv_control	ACE, ACE-Lite	AXI4	AXI3	AHB
`bool is_privileged() const;` `void set_privileged(bool = true);`	**AxPROT[0]**	**AxPROT[0]**	**AxPROT[0]**	**HPROT[1]**
`bool is_instruction() const;` `void set_instruction(bool = true);`	**AxPROT[2]**	**AxPROT[2]**	**AxPROT[2]**	**HPROT[0]**
`bool is_non_secure() const;` `void set_non_secure(bool = true);`	**AxPROT[1]**	**AxPROT[1]**	**AxPROT[1]**	-
`bool is_locked() const;` `void set_locked(bool = true);`	-	-	**AxLOCK = 2**	**HLOCK**
`bool is_exclusive() const;` `void set_exclusive(bool = true);`	AxLOCK	AxLOCK	**AxLOCK = 1**	-
`void set_bufferable(bool = true);` `bool is_bufferable() const;`	**AxCACHE[0]**	**AxCACHE[0]**	**AxCACHE[0]**	**HPROT[2]**
`void set_cacheable(bool = true);` `bool is_cacheable() const;`	-	-	**AxCACHE[1]**	**HPROT[3]**
`void set_modifiable(bool = true);` `bool is_modifiable() const;`	**AxCACHE[1]**	**AxCACHE[1]**	-	-
`void set_read_allocate(bool = true);` `bool is_read_allocate() const;`	**AxCACHE[2]**	**AxCACHE[2]**	**AxCACHE[2]**	-
`void set_write_allocate(bool = true);` `bool is_write_allocate() const;`	**AxCACHE[3]**	**AxCACHE[3]**	**AxCACHE[3]**	-
`void set_other_read_allocate(bool = true);` `bool is_other_read_allocate() const;`	**AxCACHE[3]**	**AxCACHE[3]**	-	-
`void set_other_write_allocate(bool = true);` `bool is_other_write_allocate() const;`	**AxCACHE[2]**	**AxCACHE[2]**	-	-
`void set_qos(unsigned int);` `unsigned int get_qos() const;`	**AxQOS[3:0]**	**AxQOS[3:0]**	-	-
`void set_region(unsigned int);` `unsigned int get_region() const;`	**AxREGION[3:0]**	**AxREGION[3:0]**	-	-
`void set_domain(amba_pv_domain_t);` `amba_pv_domain_t get_domain() const;`	**AxDOMAIN[1:0]**	-	-	-
`void set_snoop(amba_pv_snoop_t);` `amba_pv_snoop_t get_snoop() const;`	**AxSNOOP[3:0]**	-	-	-
`void set_bar(amba_pv_bar_t);` `amba_pv_bar_t get_bar() const;`	**AxBAR[1:0]**	-	-	-

The response mapping for AXI, ACE, AHB and APB buses is listed in Table 2-12.

Table 2-12 Response mapping for `amba_pv_resp_t`

amba_pv_resp_t	AXI xRESP	AHB HRESP	APB PSLVERR
`AMBA_PV_OKAY`	OKAY	OKAY	LOW
`AMBA_PV_EXOKAY`	EXOKAY	-	-
`AMBA_PV_SLVERR`	SLVERR	ERROR	HIGH
`AMBA_PV_DECERR`	DECERR	ERROR	HIGH

Note

APB peripherals are not required to support the **PSLVERR** signal. If a peripheral does not support this signal then the corresponding appropriate response is `AMBA_PV_OKAY`.

The additional response bit mappings for the ACE bus are listed in Table 2-13.

Table 2-13 Mapping for additional ACE response bits

amba_pv_extension and amba_pv_response	ACE	ACE-Lite
`bool is_pass_dirty() const;` `void set_pass_dirty(bool = true);`	**RRESP[2]**, **CRRESP[2]**	**RRESP[2]**
`bool is_shared() const;` `void set_shared(bool = true);`	**RRESP[3]**, **CRRESP[2]**	**RRESP[3]**
`bool is_snoop_data_transfer() const;` `void set_snoop_data_transfer(bool = true);`	CRRESP[0]	-
`bool is_snoop_error() const;` `void set_snoop_error(bool = true);`	CRRESP[1]	-
`bool is_snoop_was_unique() const;` `void set_snoop_was_unique(bool = true);`	CRRESP[4]	-

2.5 Basic transactions

This section gives examples of basic AMBA-PV transactions. Each example shows the data organization and the attributes usage. The examples are provided in:

- *Fixed burst example*
- *Incremental burst example*
- *Wrapped burst example* on page 2-29
- *Unaligned burst example* on page 2-29.

2.5.1 Fixed burst example

Figure 2-1 shows an example of a fixed read burst of four transfers. Each row in the figure represents a transfer.

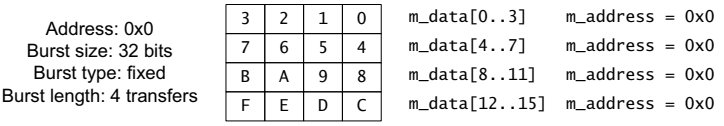

Figure 2-1 Fixed read burst

Note

The data organization is the same whether this burst happens on 32-bit or on 64-bit buses.

The attributes of the TLM 2.0 GP are as follows:

```
m_command = TLM_READ_COMMAND;
m_address = 0x0;
m_data_length = 16;
m_streaming_width = 4;
```

The attributes of the AMBA-PV extension are as follows:

```
m_burst = AMBA_PV_FIXED;
m_length = 4;
m_size = 4;
```

Note

This transaction is specific to the AMBA 3 AXI protocol.

2.5.2 Incremental burst example

Figure 2-2 shows an example of an incremental write burst of four transfers. Each row in the figure represents a transfer.

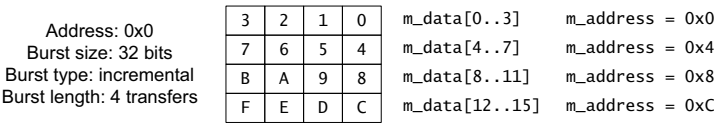

Figure 2-2 Incremental write burst

Note

The data organization is the same whether this burst happens on 32-bit or on 64-bit buses.

The attributes of the TLM 2.0 GP are as follows:

```
m_command = TLM_WRITE_COMMAND;
m_address = 0x0;
m_data_length = 16;
m_streaming_width = 16;
```

The attributes of the AMBA-PV extension are as follows:

```
m_burst = AMBA_PV_INCR;
m_length = 4;
m_size = 4;
```

2.5.3 Wrapped burst example

Figure 2-3 shows an example of a wrapped burst of four transfers. Each row in the figure represents a transfer.

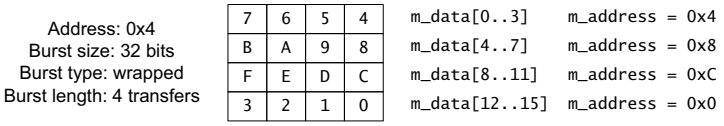

Figure 2-3 Wrapped burst

Note

The data organization is the same whether this burst happens on 32-bit or 64-bit buses.

The attributes of the TLM 2.0 GP are as follows:

```
m_command = TLM_WRITE_COMMAND;
m_address = 0x4;
m_data_length = 16;
m_streaming_width = 16;
```

The attributes of the AMBA-PV extension are as follows:

```
m_burst = AMBA_PV_WRAP;
m_length = 4;
m_size = 4;
```

2.5.4 Unaligned burst example

Figure 2-4 shows an example of an unaligned incremental write burst of four transfers. Each row in the figure represents a transfer. The shaded cells indicate bytes that are not transferred, based on the address and byte enable attributes.

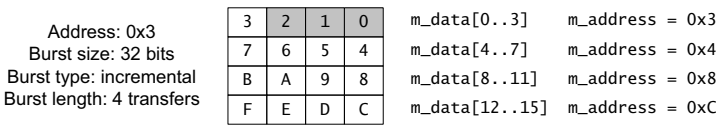

Figure 2-4 Unaligned write burst

Note

The data organization is the same whether this burst happens on 32-bit or 64-bit buses.

The attributes of the TLM 2.0 GP are as follows:

```
m_command = TLM_WRITE_COMMAND;
m_address = 0x3;
m_data_length = 16;
m_byte_enable_length = 16;
m_byte_enable_ptr = {0x00, 0x00, 0x00, 0xFF...};
m_streaming_width = 16;
```

The attributes of the AMBA-PV extension are as follows:

```
m_burst = AMBA_PV_INCR;
m_length = 4;
m_size = 4;
```

Note

This transaction is specific to the AMBA 3 AXI bus.

Chapter 3
AMBA-PV Classes

This chapter provides an overview of the AMBA-PV class hierarchy and briefly describes each major class. It contains the following sections:

3.1 Class description

This section describes the relationships between the AMBA-PV and TLM 2.0 classes and interfaces. It contains the following subsections:

- *AMBA-PV extension*
- *Core interfaces* on page 3-3
- *User layer* on page 3-4
- *Sockets* on page 3-5
- *Bridges* on page 3-6
- *Memory* on page 3-7
- *Exclusive monitor* on page 3-8
- *Bus decoder* on page 3-9
- *Protocol checker* on page 3-9
- *Signaling* on page 3-10
- *User and transport layers* on page 3-11.

Note

All AMBA-PV classes and interfaces use the `amba_pv` namespace.

3.1.1 AMBA-PV extension

The AMBA-PV extension, `amba_pv_extension` class, is shown in Figure 3-1 on page 3-3. It extends the `tlm_extension` class and provides support for AMBA 4 buses specific addressing options and additional control information.

The additional control information provided by the AMBA 4 buses is modeled by the `amba_pv_control` class. It is also used by the user interface methods. See *User layer* on page 3-4.

The additional transaction information required by DVM operations is modeled by the `amba_pv_dvm` class.

The `amba_pv_attributes` class provides support for additional user-defined attributes in the form of additional named attributes (namely a map). To use this class, you must define the `AMBA_PV_INCLUDE_ATTRIBUTES` macro at compile time.

Note

The `amba_pv_attributes` class might impact simulation performance.

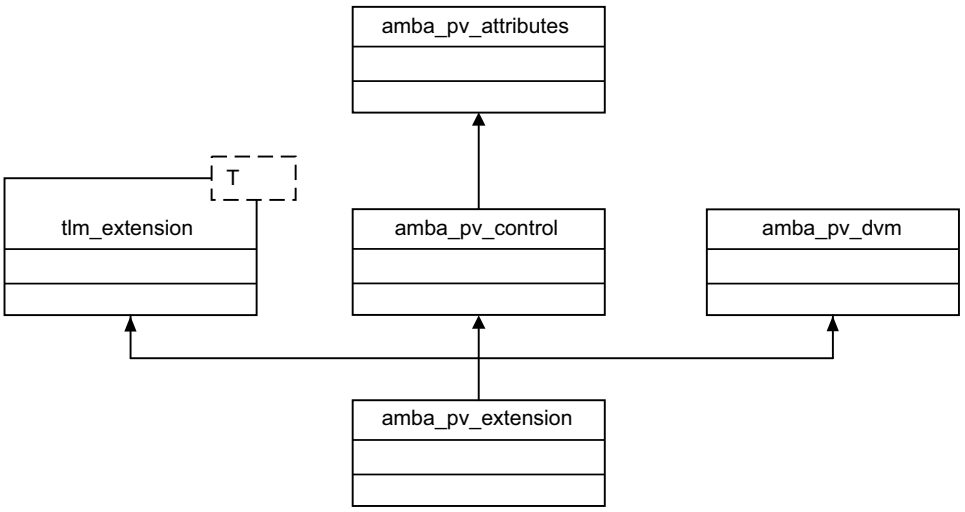

Figure 3-1 Extension hierarchy

3.1.2 Core interfaces

The AMBA-PV core interfaces shown in Figure 3-2 on page 3-4 comprise:

`amba_pv_fw_transport_if`

tagged variant of `tlm_fw_transport_if`, must be implemented by AMBA-PV slave modules

`amba_pv_bw_transport_if`

tagged variant of `tlm_bw_transport_if`, must be implemented by AMBA-PV master modules.

`amba_pv_bw_snoop_if`

tagged variant of `tlm_fw_transport_if`.

`amba_pv_bw_transport_and_snoop_if`

tagged variant of `tlm_fw_transport_if` and `tlm_bw_transport_if`, must be implemented by AMBA-PV ACE master modules. This class is a simple composite of the `amba_pv_bw_transport_if` and `amba_pv_bw_snoop_if`.

The core interfaces are part of the transport layer. See *User and transport layers* on page 3-11 for more information.

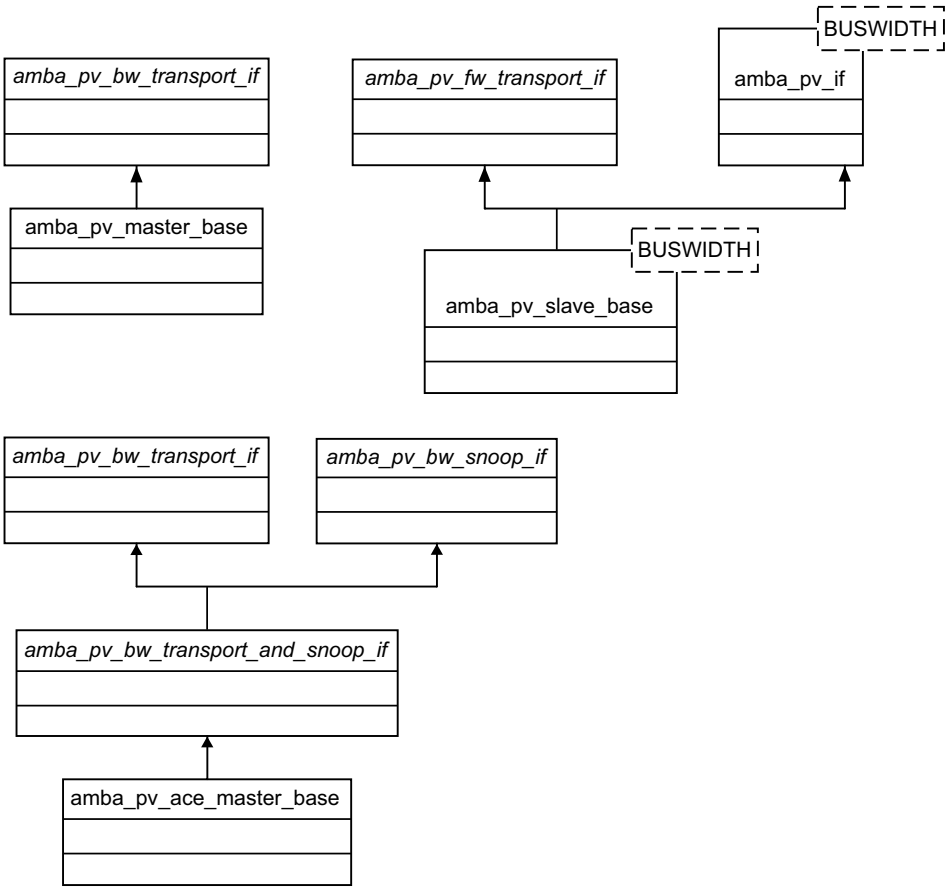

Figure 3-2 Core interfaces and user layer

3.1.3 User layer

The user layer shown in Figure 3-2 comprises the following:

`amba_pv_if<>`

user-layer transaction interface providing `read()`, `write()`, `burst_read()`, `burst_write()`, `debug_read()`, `debug_write()`, `get_direct_mem_ptr()` convenience methods

`amba_pv_master_base`

base class for AMBA-PV master modules, to be bound to `amba_pv_master_socket<>`, provides default implementations of `invalidate_direct_mem_ptr()`.

`amba_pv_slave_base<>`

base class for AMBA-PV slave modules, to be bound to `amba_pv_slave_socket<>`, provides with conversion of `b_transport()` and `transport_dbg()` into user-layer methods, and default implementations of `transport_dbg()` and `get_direct_mem_ptr()`.

`amba_pv_ace_master_base`

base class for AMBA-PV ACE master modules, to be bound to `amba_pv_ace_master_socket<>`, provides default implementations of `invalidate_direct_mem_ptr()`, `b_snoop()` and `snoop_dbg()`.

See *User and transport layers* on page 3-11 for more information on how the user layer builds on top of the transport layer.

3.1.4 Sockets

The `amba_pv_master_socket<>` class shown in Figure 3-3 provides:

- socket identification/tagging
- implementation for the `amba_pv_if` user-layer interface.

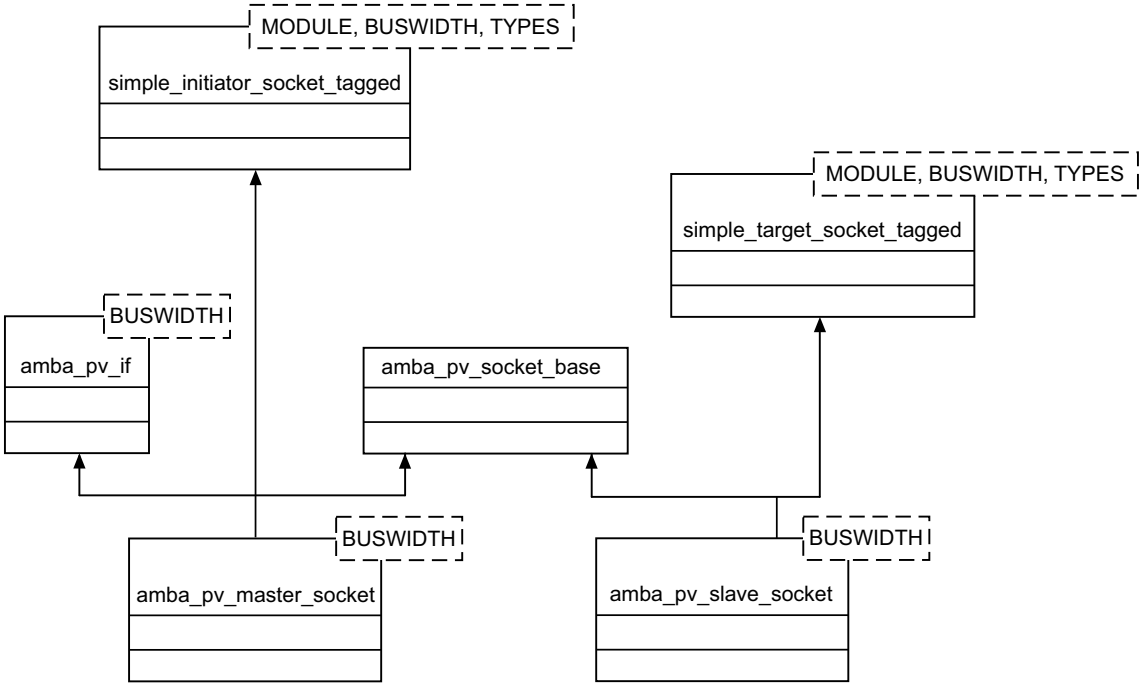

Figure 3-3 Sockets

The `amba_pv_slave_socket<>` class shown in Figure 3-3 features socket identification/tagging.

3.1.5 ACE sockets

The `amba_pv_ace_master_socket<>` class shown in Figure 3-4 on page 3-6 provides:

- all the function of `amba_pv_master_socket<>`
- includes an `amba_pv_snoop_socket<>` as a private data member.

The `amba_pv_ace_slave_socket<>` class shown in Figure 3-4 on page 3-6provides:

- all the function of `amba_pv_slave_socket<>`
- includes an `amba_pv_master_socket<>` as a private data member.

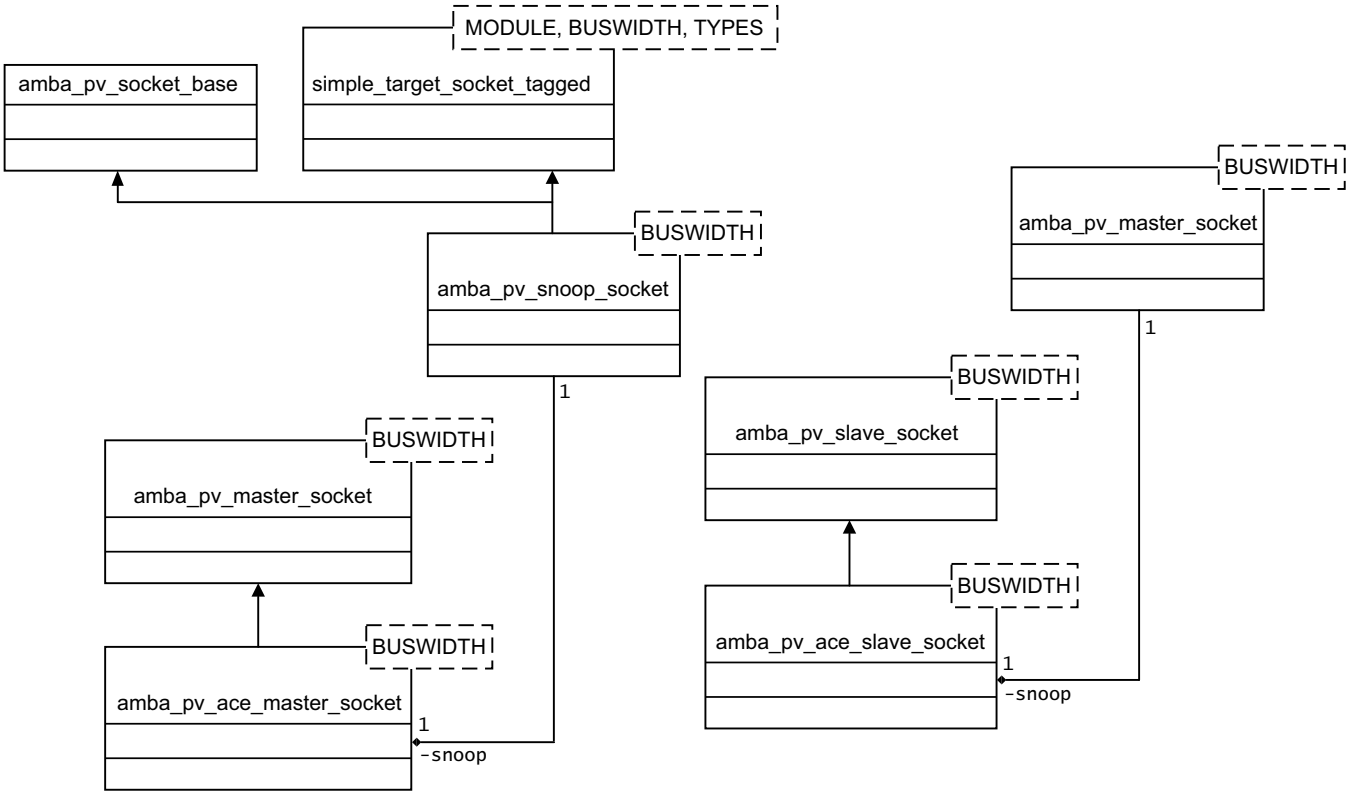

Figure 3-4 ACE sockets

3.1.6 Bridges

The `amba_pv_to_tlm_bridge<>` and `amba_pv_from_tlm_bridge<>` classes shown in Figure 3-5 on page 3-7 bridge between TLM 2.0 BP and AMBA-PV.

If bridging from TLM 2.0 BP to AMBA-PV, the following rules are checked:

- the address attribute must be aligned to the bus width for burst transactions and to the data length for single transactions
- the data length attribute must be a multiple of the bus width for burst transactions
- the streaming width attribute must be equal to the bus width for fixed burst transactions
- the byte enable pointer attribute must be `NULL` on read transactions
- the byte enable length attribute must be equal to the data length for single write transactions and a multiple of the bus width for burst write transactions, if non zero.

If bridging from AMBA-PV to TLM 2.0 BP, wrapping bursts are translated into sequential (incremental) bursts.

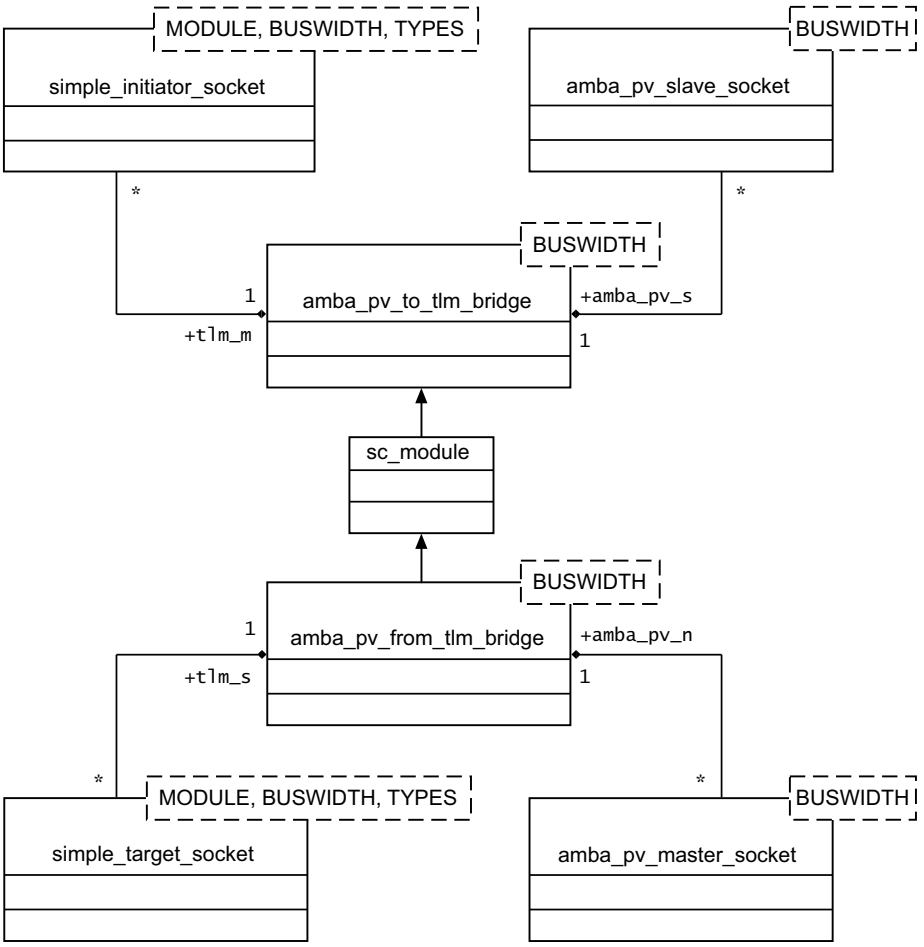

Figure 3-5 AMBA-PV - TLM bridges

3.1.7 Memory

Memories can be represented by either a simple model or an advanced model, as shown in Figure 3-6 on page 3-8. The advanced model, class `amba_pv_memory<>`, supports optimized heap usage, save and restore.

Figure 3-6 Memory

3.1.8 Exclusive monitor

The `amba_pv_exclusive_monitor<>` class shown in Figure 3-7 provides exclusive access support and can be added before any AMBA-PV slave.

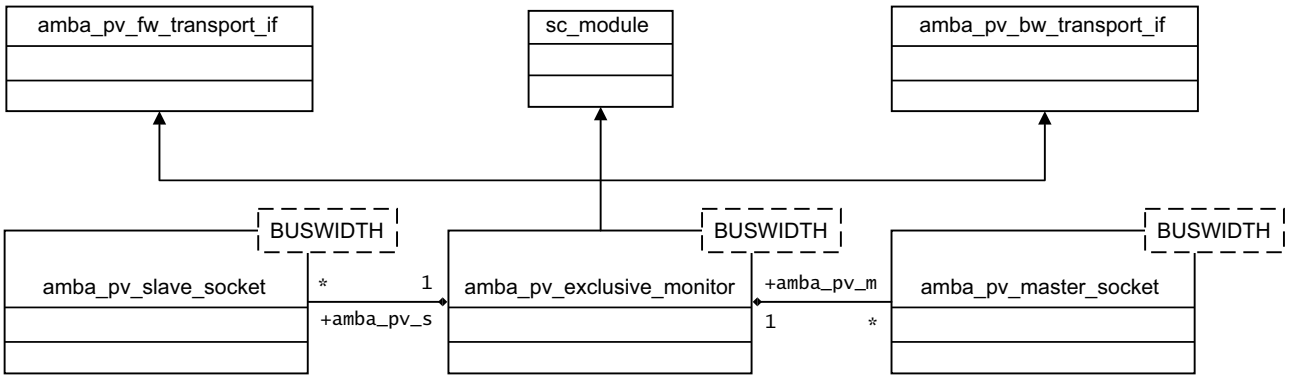

Figure 3-7 Monitor

3.1.9 Bus decoder

The amba_pv_decoder<> class shown in Figure 3-8 routes transactions through to the appropriate slave depending on the transaction address. It can load its address map from a stream or file.

Note

The amba_pv_decoder<> class does not currently support locked transactions. Any locked transaction are handled as if not locked.

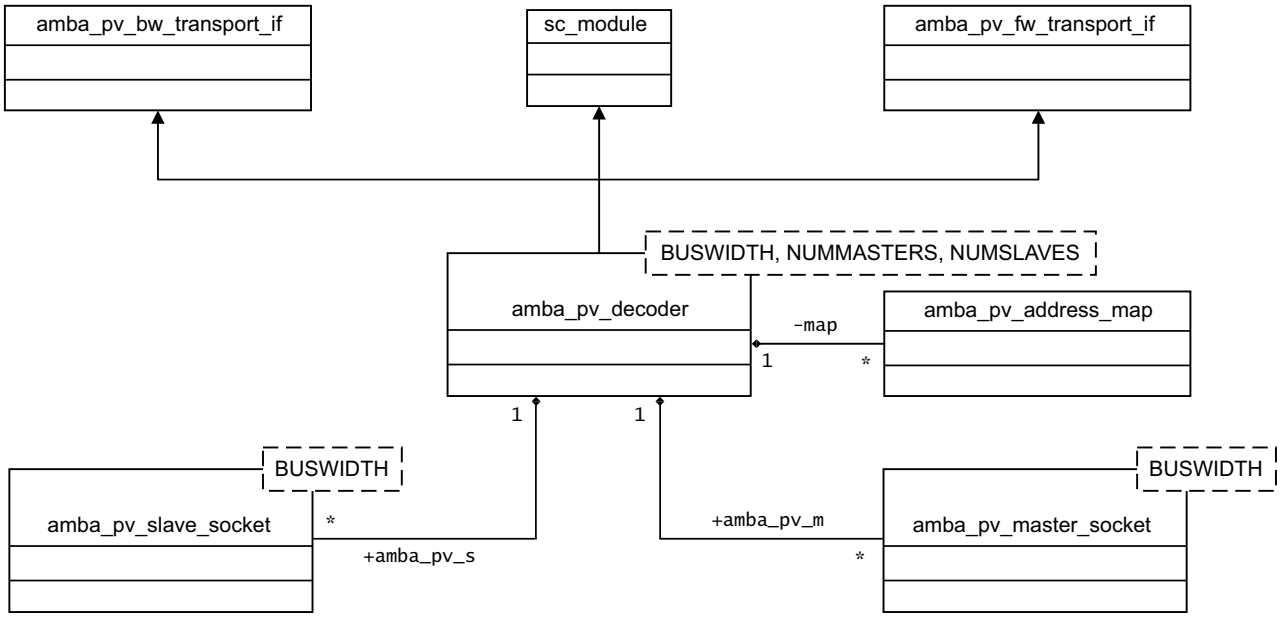

Figure 3-8 Bus decoder

3.1.10 Protocol checker

The amba_pv_protocol_checker<> class shown in Figure 3-9 on page 3-10 is used for confirming that a model complies with AMBA buses protocols.

The transactions that passes through are checked against the AMBA buses protocols. Errors are reported using the SystemC reporting mechanism.

Note

The AMBA-PV protocol checker does not perform any TLM 2.0 BP checks.

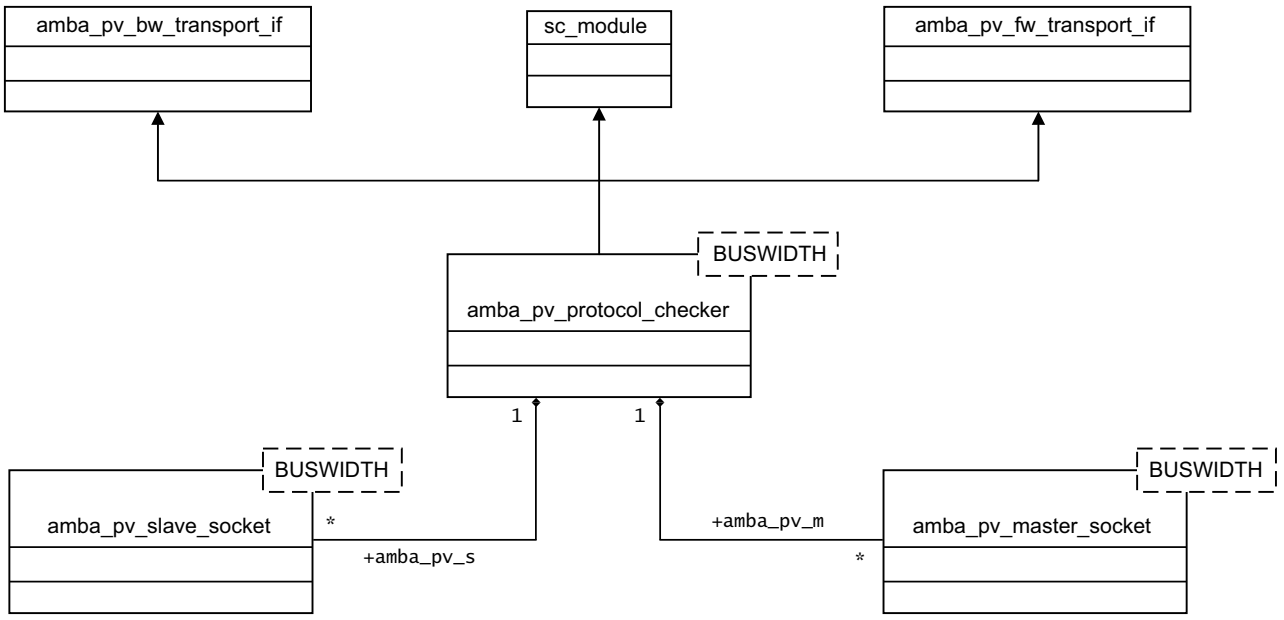

Figure 3-9 Protocol checker

See Chapter 6 *AMBA-PV Protocol Checker* for more information on the protocol checker and the checks it performs.

3.1.11 Signaling

The Signal API shown in Figure 3-10 on page 3-11 defines classes and interfaces for the modeling of side-band signals such as, for example, interrupts. There are two variants of these classes and interfaces:

- the Signal one that permits components to indicate a signal state change to other components and uses the `signal_` prefix
- the SignalState one that permits the other components to passively query the current state of the signal and uses the `signal_state_` prefix.

The Signal API features immediate propagation of the signal state (no update phase or time elapse) and does not require intermediate storage of the signal state in a channel.

The Signal classes and interfaces features a `STATE` template parameter.

Note

These Signal classes and interfaces are provided as part of AMBA-PV as an alternative to using SystemC `sc_signal<>` for side-band signal modeling at PV-level. The SystemC `sc_signal<>` is implemented as a primitive channel using the request/update mechanism. This introduces extra processes, resulting in extra delta cycles in the simulation, and prevents immediate propagation of the signal state.

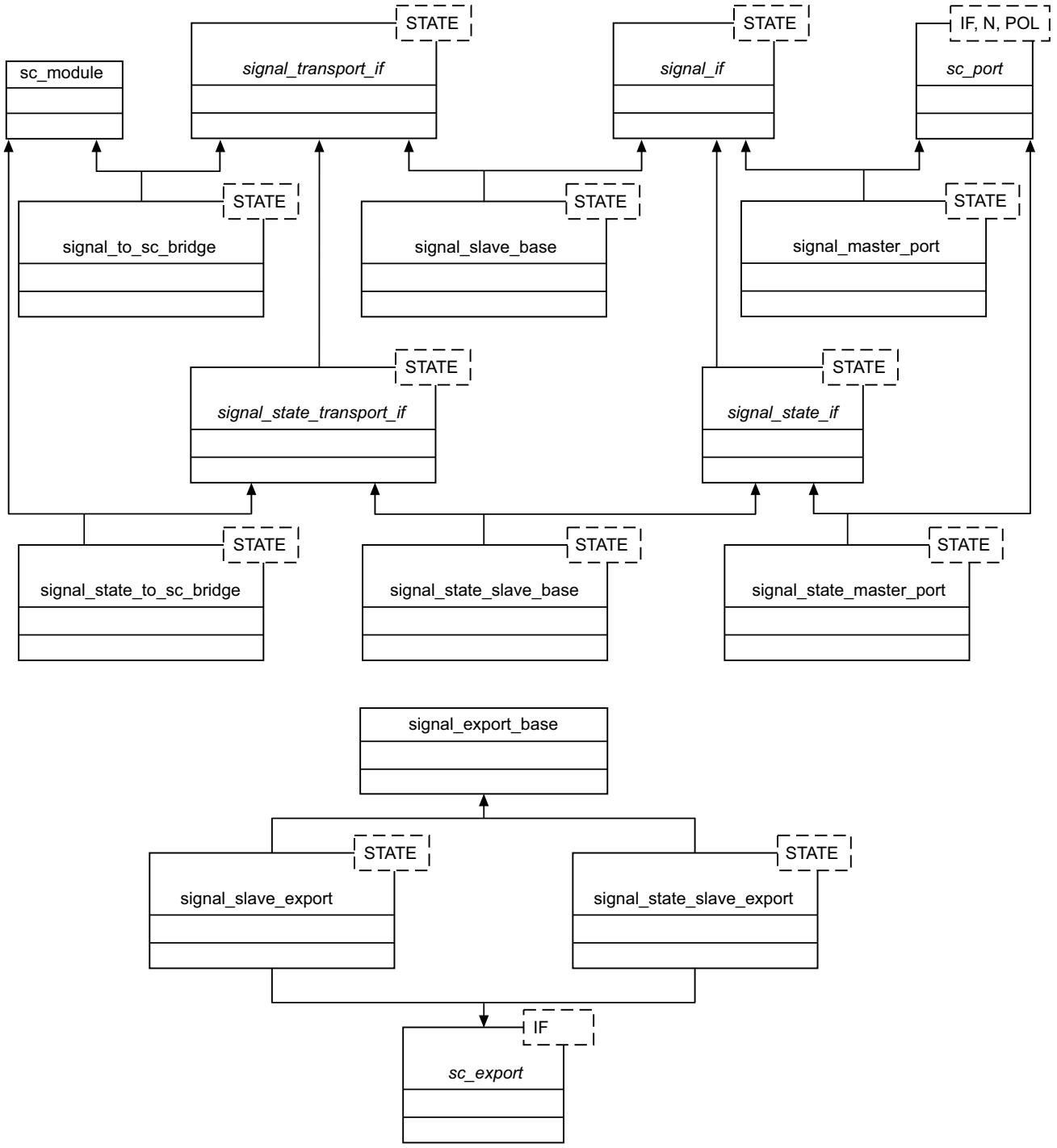

Figure 3-10 Signaling

3.1.12 User and transport layers

The AMBA-PV user and transport layers manage interactions between the master and slave.

Forward calls from master to slave

These calls go from the user layer through the transport layer and back to user layer as shown in Figure 3-11 on page 3-12.

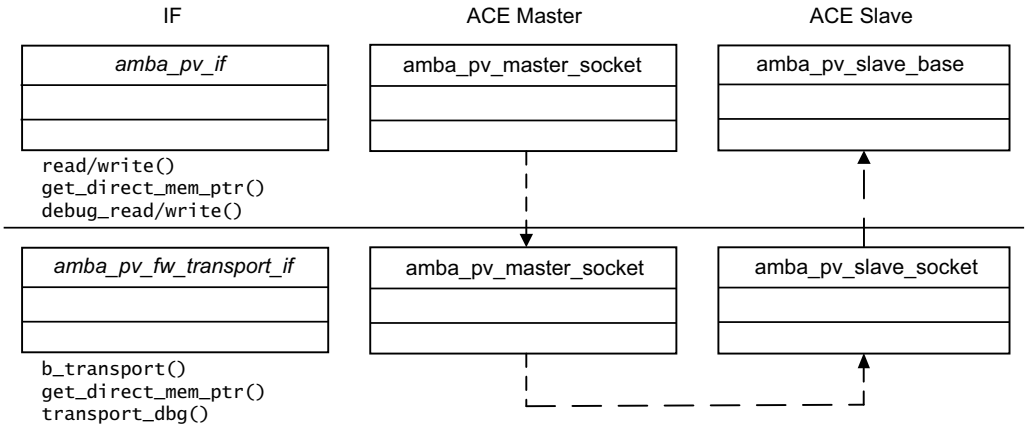

Figure 3-11 Master to slave calls

The `amba_pv_if<>` interface is implemented by the master socket. Class `amba_pv_slave_base<>` inherits from this interface. The interface defines the following member functions:

- `read()`
- `read_burst()`
- `write()`
- `write_burst()`
- `get_direct_mem_ptr()`
- `debug_read()`
- `debug_write()`.

The `amba_pv_fw_transport_if` interface is an AMBA-PV core interface. Class `amba_pv_slave_base<>` also inherits from this interface. The interface defines the following member functions:

- `b_transport()`
- `get_direct_mem_ptr()`
- `transport_dbg()`.

Backward calls from slave to master

These calls go from the user layer through the transport layer and back to user layer as shown in Figure 3-12 on page 3-13.

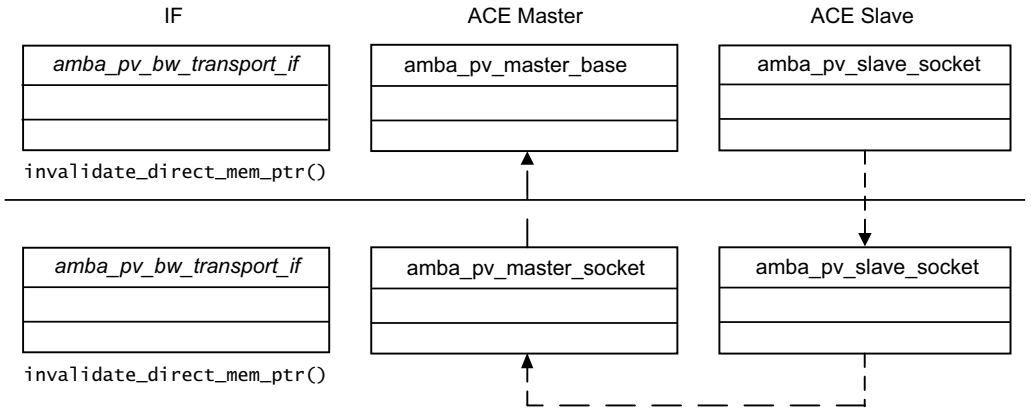

Figure 3-12 Slave to master calls

The `amba_pv_bw_transport_if` interface is an AMBA-PV core interface. It defines the `invalidate_direct_mem_ptr()` member function to invalidate pointers that were previously established for a DMI region in the slave and features tagging through its socket identification parameter.

Using the ACE sockets

The forward calls from ACE masters to ACE slaves follow a similar flow as for the non-ACE sockets. See Figure 3-13.

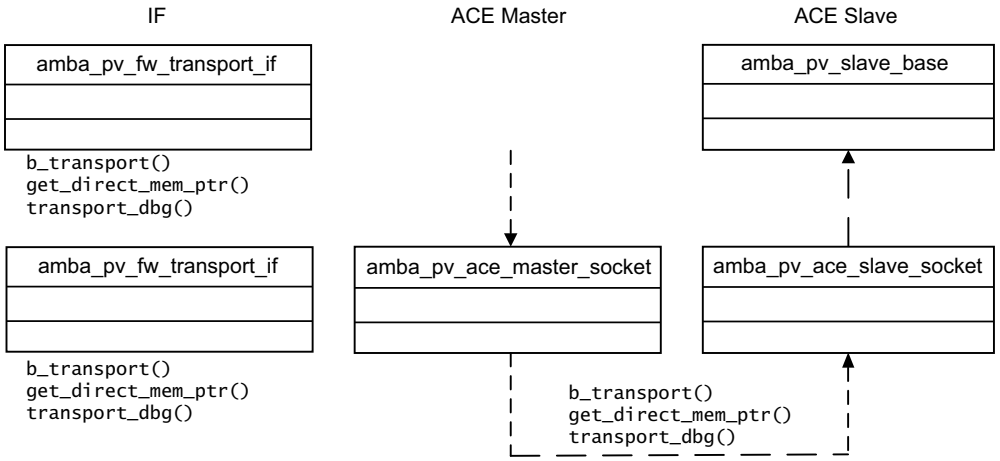

Figure 3-13 ACE master to slave calls

The user layer is not useful for modeling ACE transactions because the extra response attributes required by ACE are not available in `amba_pv_control`.

Note

This is so that source level compatibility with previous versions of AMBA-PV can be maintained.

Backward calls from ACE slaves to ACE masters are shown in Figure 3-14 on page 3-14.

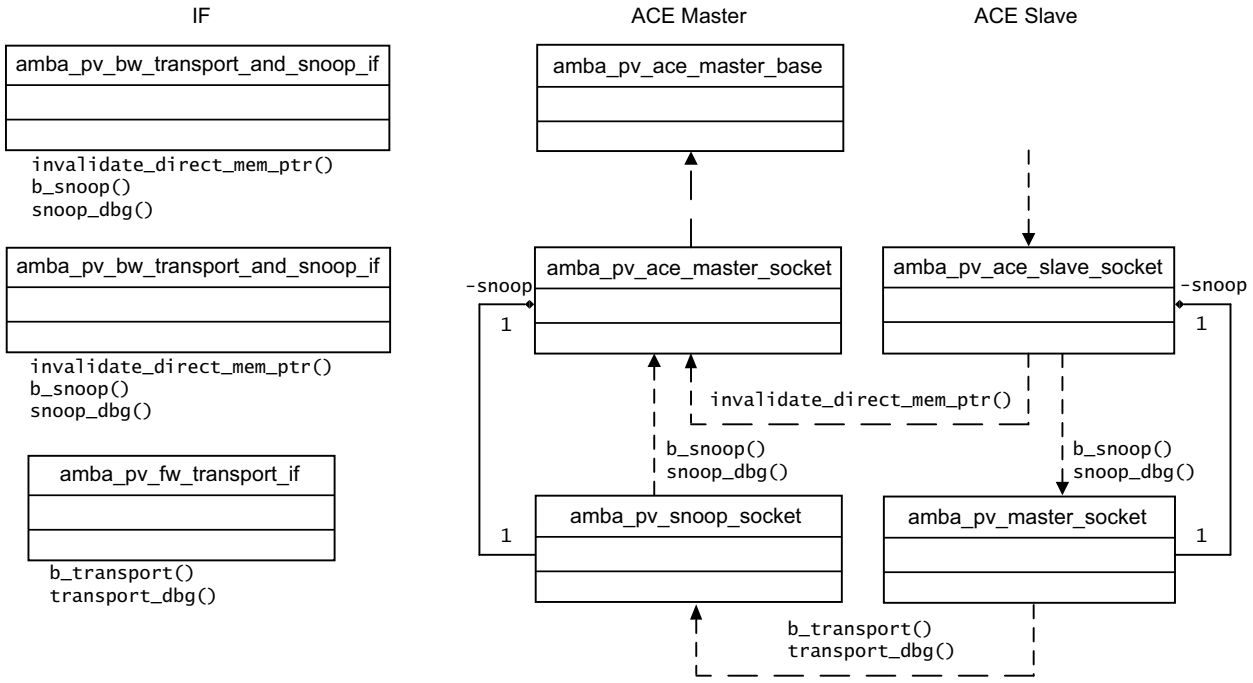

Figure 3-14 ACE slave to master calls

The `amba_pv_bw_transport_and_snoop_if` interface is an AMBA-PV core interface. Class `amba_pv_ace_master_base<>` also inherits from this interface. The interface defines the following member functions:

`invalidate_direct_mem_ptr()`

Invalidate pointers that were previously returned via `get_direct_mem_ptr()`.

`b_snoop()`

Equivalent function to the forward method `b_transport()` but used for transactions in the upstream slave to master direction.

`snoop_dbg()`

Equivalent function to the forward method `transport_dbg()` but used for transactions in the upstream slave to master direction.

3.2 Class summary

This section provides a summary of the AMBA-PV classes and interfaces.

Note

For the full list of classes and interfaces, see the AMBA-PV header files. The top-level file is `amba_pv.h` which contains includes for the other header files.

All AMBA-PV classes and interfaces use the `amba_pv` namespace.

The following classes and interfaces are defined in AMBA-PV:

`amba_pv_attributes`
: This supports additional user-defined attributes.

`amba_pv_ace_master_base`
: This is the base class for AMBA-PV ACE master modules.

`amba_pv_ace_master_socket<>`
: This is the socket to be instantiated on the master side for full ACE modeling.

`amba_pv_ace_slave_socket<>`
: This is the socket to be instantiated on the slave side for full ACE modeling.

`amba_pv_bw_snoop_if`
: This is a tagged variant of the `tlm_bw_transport_if` interface, to be implemented by AMBA-PV ACE master modules.

`amba_pv_bw_transport_and_snoop_if`
: This is a simple combination of the interfaces `amba_pv_bw_snoop_if` and `amba_pv_bw_transport_if`.

`amba_pv_bw_transport_if`
: This is a tagged variant of the `tlm_bw_transport_if` interface, to be implemented by AMBA-PV master modules.

`amba_pv_control`
: This supports additional control information that is part of the AMBA buses.

`amba_pv_dvm`
: This provides additional transaction information for DVM operations.

`amba_pv_extension`
: This is the AMBA-PV extension type.

`amba_pv_fw_transport_if`
: This is a tagged variant of the `tlm_fw_transport_if` interface, to be implemented by AMBA-PV slave modules.

`amba_pv_if<>`
: This is the user-layer transaction interface.

`amba_pv_master_base`
: This is the base class for AMBA-PV master modules.

`amba_pv_master_socket<>`

This is the socket to be instantiated on the master side. This socket is also automatically instantiated on the slave side when an `amba_pv_ace_slave_socket<>` is instantiated.

`amba_pv_slave_base<>`

This is the base class for AMBA-PV slave modules.

`amba_pv_slave_socket<>`

This is the socket to be instantiated on the slave side.

`amba_pv_snoop_socket<>`

This socket is automatically instantiated on the master side when an amba_pv_ace_master_socket<> is instantiated.

The templated AMBA-PV classes and interfaces have a `BUSWIDTH` parameter.

An AMBA-PV bus master invokes methods on its `amba_pv_master_socket` to generate burst read and write requests on the AMBA-PV bus and check the returned responses.

An AMBA-PV bus slave implements `read()` and `write()` methods to process requests and return the associated responses.

The TLM 2.0 `b_transport()` blocking interface is the basic mechanism that implements this master-slave interaction. In addition, AMBA-PV uses the extension mechanism to extend TLM 2.0 and provide maximum interoperability.

3.2.1 Additional classes for virtual platforms

The following classes and interfaces are defined to model virtual platform components:

`amba_pv_address_map`

This defines the structures related to address maps.

`amba_pv_decoder<>`

This is the bus decoder model.

`amba_pv_exclusive_monitor<>`

This supports AMBA 3 exclusive accesses.

`amba_pv_from_tlm_bridge<>`

This is the bridge module for interface between TLM 2.0 BP and AMBA-PV. This provides interoperability at subsystem boundaries. The component uses the TLM 2.0 extension mechanism.

`amba_pv_memory<>`

This is the advanced memory model that features optimized heap usage, save, and restore.

`amba_pv_memory_base<>`

This is the base class for memory models.

`amba_pv_protocol_checker<>`

This is the protocol checker that is used for conforming that a platform or model complies with the AMBA-PV protocol.

`amba_pv_simple_memory<>`

This is the simple memory model.

`amba_pv_simple_probe<>`

This is the simple probe component that dumps the contents of transactions.

`amba_pv_to_tlm_bridge<>`

This is the bridge module for interface between TLM 2.0 BP and AMBA-PV. This provides interoperability at subsystem boundaries. The component uses the TLM 2.0 extension mechanism.

These templated classes and interfaces have a `BUSWIDTH` parameter.

3.2.2 Additional classes for side-band signals

The following classes and interfaces are defined to model side-band signals. There are variants with or without `get_state()` access function to passively query the current state of the signal:

`signal_export_base<>`

This is the Signal export base class.

`signal_from_sc_bridge<>`

This is the generic bridge module from `sc_signal<>` to Signal.

`signal_if<>`

This is the user-layer interface for Signal.

`signal_master_port<>`

This is the port to be instantiated on the Signal master side.

`signal_request<>`

This is the Signal request type.

`signal_response<>`

This is the Signal response type.

`signal_slave_export<>`

This is the export to be instantiated on the Signal slave side.

`signal_slave_base<>`

This is the base class for Signal slave modules.

`signal_state_if<>`

This is the user-layer interface for SignalState.

`signal_state_nonblocking_transport_if<>`

This is the core non-blocking transport interface for SignalState.

`signal_state_to_sc_bridge<>`

This is a generic bridge module from SignalState to `sc_signal<>`.

`signal_state_from_sc_bridge<>`

This is the generic bridge module from `sc_signal<>` to SignalState.

`signal_state_master_port<>`

This is the port to be instantiated on the SignalState master side.

`signal_state_slave_base<>`

This is the base class for SignalState slave modules.

`signal_state_slave_export<>`

This is the export to be instantiated on the SignalState slave side.

`signal_to_sc_bridge<>`

This is the generic bridge module between Signal and `sc_signal<>`.

`signal_nonblocking_transport_if<>`

This is the core non-blocking transport interface for the Signal.

The templated Signal classes and interfaces have a `STATE` parameter.

Chapter 4
Example Systems

This chapter describes the procedure to build and run the example systems provided with AMBA-PV and located in `$MAXCORE_HOME/AMBA-PV`. It contains the following sections:

4.1 Configuring the examples

This section describes how to configure the AMBA-PV examples. The examples are installed with AMBA-PV and located in `$MAXCORE_HOME/AMBA-PV`.

The examples use SystemC and TLM headers and libraries and require the following environment variables to be set:

Table 4-1 Environment variables

Environment variable	Description
`SYSTEMC_HOME`	Points to the directory SystemC 2.2.0 is installed into
`TLM_HOME`	Points to the directory TLM 2.0 is installed into

Note

The `SYSTEMC_HOME` and `TLM_HOME` environment variables are set when AMBA-PV is installed. If you require a different copy of SystemC or TLM, modify the variables accordingly before building the examples.

SystemC and TLM headers and libraries are installed in `$MAXCORE_HOME/OSCI`, which contains releases of the SystemC and TLM packages and patch files. The patch files document the required changes to the SystemC and TLM packages available from Accellera, `http://www.accellera.org`. The SystemC and TLM packages are link compatible with the Accellera download version.

Note

The AMBA-PV examples rely on a certain directory structure for libraries and header files. This is different from the packages that can be obtained from Accellera because AMBA-PV supports a different range of compilers than the Accellera packages. If the original OSCI packages are used in conjunction with the AMBA-PV examples, a set of patch files must be applied to the Accellera package that adjusts the directory names. To re-build the packages follow the instructions from the `README.txt` file available in the `$MAXCORE_HOME/OSCI/source` directory.

On Linux hosts, the `make` command in each example directory generates an executable that consists of the example name followed by `.x` (for example, `dma.x`, or `bridge.x`).

On Microsoft Windows hosts, Microsoft Visual Studio project files are provided for the 2005 and 2008 versions (for example, `dma_VC2005.vcproj`, or `bridge_VC2008.vcproj`, or `bridge_VC2010.vcxproj`).

4.2 Bridge example

This example, shown in Figure 4-1, illustrates bridging to and from the TLM BP using the `amba_pv_to_tlm_bridge<>` and `amba_pv_from_tlm_bridge<>` classes. It is based on the exclusive example (see *Exclusive example* on page 4-9) and features:

- a simple memory, class `amba_pv_simple_memory<>`
- an exclusive access monitor, class `amba_pv_exclusive_monitor<>`
- two masters competing for access to this memory, the first performs exclusive accesses while the second performs regular accesses
- an `amba_pv_to_tlm_bridge<>` - `amba_pv_from_tlm_bridge<>` bridges chain inserted between the masters and the memory
- a bus decoder, class `amba_pv_bus_decoder<>`, routing transactions from the masters to the exclusive access monitor.

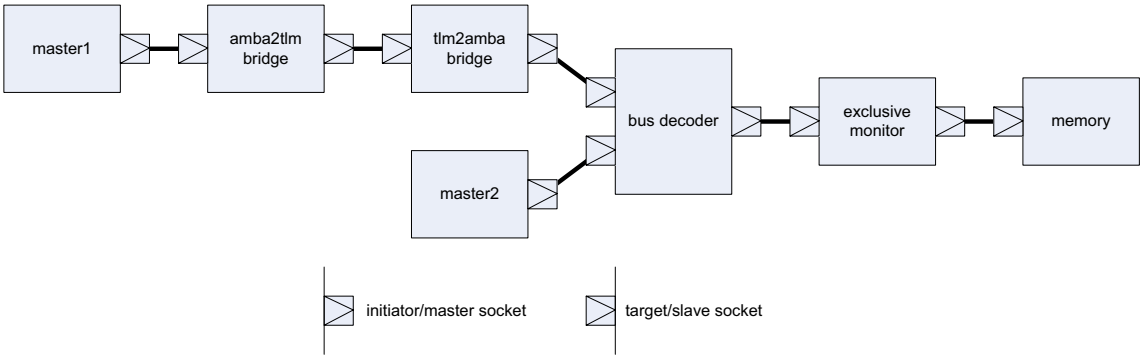

Figure 4-1 Bridge example system

The example is located in `$MAXCORE_HOME/AMBA-PV/examples/bridge_example`.

4.2.1 Building and running the example

To build the debug version of this example, perform the following:

- under Linux, enter the following at the command line:

  ```
  make DEBUG=y clean all
  ```

- under Microsoft Windows, open `bridge_[VC2005|VC2008].vcproj` or `bridge_VC2010.vcxproj` with Microsoft Visual Studio and build the `bridge` project, with the `Debug` configuration active.

To build the release version of this example, perform the following:

- under Linux, enter the following at the command line:

  ```
  make DEBUG=n clean all
  ```

- under Microsoft Windows, open `bridge_[VC2005|VC2008].vcproj` or `bridge_VC2010.vcxproj` with Microsoft Visual Studio and build the `bridge` project, with the `Release` configuration active.

Note

Under Linux, the `make clean` command is optional. If used, the example is completely rebuilt.

To run this example, enter the following at the command line:

- under Linux:
  ```
  ./bridge.x
  ```
- under Microsoft Windows:
  ```
  bridge.exe
  ```

4.3 Debug example

This example, shown in Figure 4-2, illustrates the use of AMBA-PV debug transfers between a master and a slave.

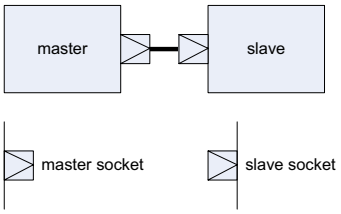

Figure 4-2 Debug example system

The example is located in `$MAXCORE_HOME/AMBA-PV/examples/dbg_example`.

4.3.1 Building and running the example

To build the debug version of this example, perform the following:

- under Linux:

  ```
  make DEBUG=y clean all
  ```

- under Microsoft Windows, open `dbg_[VC2005|VC2008].vcproj` or `dbg_VC2010.vcxproj` with Microsoft Visual Studio and build the `dbg` project, with the `Debug` configuration active.

To build the release version of this example, perform the following:

- under Linux, enter the following at the command line:

  ```
  make DEBUG=n clean all
  ```

- under Microsoft Windows, open `dbg_[VC2005|VC2008].vcproj` or `dbg_VC2010.vcxproj` with Microsoft Visual Studio and build the `dbg` project, with the `Release` configuration active.

Note

Under Linux, the `make clean` command is optional. If used, the example is completely rebuilt.

To run this example, enter the following at the command line:

- under Linux:

  ```
  ./dbg.x
  ```

- under Microsoft Windows:

  ```
  dbg.exe
  ```

4.4 DMA example

This example, shown in Figure 4-3, illustrates the use of AMBA-PV burst transfers and the Signal API in a system comprising a simple DMA model programmed to perform transfers between 2 memories. Additionally, it illustrates the use of DMI for simulation performances optimization.

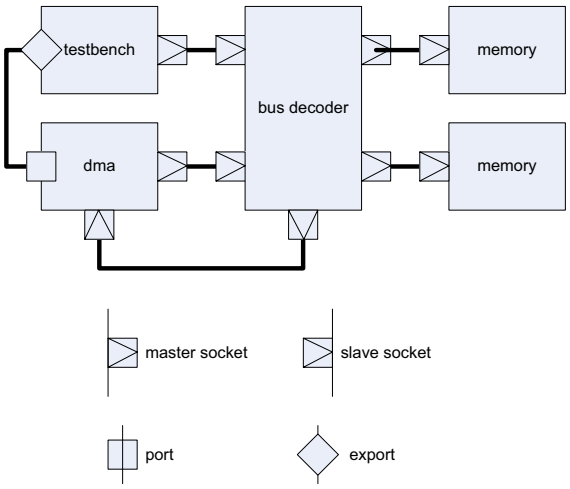

Figure 4-3 DMA example system

This example comprises the following components:

- a simple test bench to program the DMA transfers
- an AMBA-PV bus decoder, class `amba_pv_decoder<>`, to route transactions between the system components
- a simple DMA model, implementing a producer-consumer scheme and capable of using DMI for memory transfers
- two AMBA-PV memories, class `amba_pv_memory<>`.

The example is located in `$MAXCORE_HOME/AMBA-PV/examples/dma_example`.

4.4.1 Building and running the example

To build the debug version of this example, perform the following:

- under Linux, enter the following at the command line:

  ```
  make DEBUG=y clean all
  ```

- under Microsoft Windows, open `dma_[VC2005|VC2008].vcproj` or `dma_VC2010.vcxproj` with Microsoft Visual Studio and build the `dma` project, with the `Debug` configuration active.

To build the release version of this example, perform the following:

- under Linux, enter the following at the command line:

  ```
  make DEBUG=n clean all
  ```

- under Microsoft Windows, open `dma_[VC2005|VC2008].vcproj` or `dma_VC2010.vcxproj` with Microsoft Visual Studio and build the `dma` project, with the `Release` configuration active.

Note

Under Linux, the `make clean` command is optional. If used, the example is completely rebuilt.

To run this example, enter the following at the command line:

- under Linux:

  ```
  ./dma.x
  ```

- under Microsoft Windows:

  ```
  dma.exe
  ```

To run this example over a giving number of transfers, enter the following at the command line:

- under Linux:

  ```
  ./dma.x 400000
  ```

- under Microsoft Windows:

  ```
  dma.exe 400000
  ```

Where 40000 specifies the number of transfers to run.

Simulation statistics are displayed as follows:

```
tb module created - 400000 runs
dma module created

Simulation starts...
Simulation ends

--- Simulation statistics: ----------------------------------------------
Total transactions executed : 4400000
Total KBytes transferred    : 210938
Total simulation time       : 18446744.000000 sec.
Real simulation time        : 10.200000 sec.
Transactions per sec.       : 431372.557
KBytes transferred per sec. : 20680.147
-------------------------------------------------------------------------
```

To run this example with DMI enabled, enter the following at the command line:

- under Linux:

  ```
  ./dma.x --dmi 400000
  ```

- under Microsoft Windows:

  ```
  dma.exe --dmi 400000
  ```

Simulation statistics are displayed as follows:

```
tb module created - 400000 runs
dma module created

Simulation starts...
Simulation ends

--- Simulation statistics: ----------------------------------------------
Total transactions executed : 4400000
Total KBytes transferred    : 210938
Total simulation time       : 18446744.000000 sec.
Real simulation time        : 2.180000 sec.
```

```
Transactions per sec.       : 2018348.562
KBytes transferred per sec. : 96760.318
------------------------------------------------------------------------
```

Note

These figures are given here as examples. They do not constitute any reference in terms of timing. They can vary according to the host configuration on which the example is running.

For information, those figures were obtained on a RedHat Enterprise 4, Intel 32bits, Linux 2.6.9 host.

4.5 Exclusive example

This example, shown in Figure 4-4, illustrates the use of specific AMBA protocol control information with exclusive access to a simple memory through an exclusive access monitor.

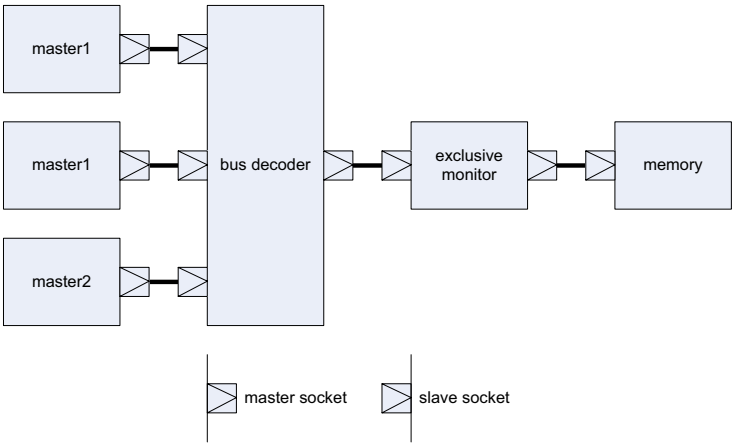

Figure 4-4 Exclusive example system

This example comprises the following components:

- a simple memory, class `amba_pv_simple_memory<>`
- an exclusive access monitor, class `amba_pv_exclusive_monitor<>`
- three masters competing for access to this memory, the first two perform exclusive accesses while the third performs regular accesses
- a bus decoder, class `amba_pv_decoder<>`, to route transactions from the masters to the exclusive access monitor.

This example also features a `PROBE` version which includes an intermediate probe component, class `amba_pv_simple_probe<>`, to print the contents of transactions exchanged between the masters and the exclusive monitor.

The example is located in `$MAXCORE_HOME/AMBA-PV/examples/exclusive_example`.

4.5.1 Building and running the example

To build the debug version of this example, perform the following:

- under Linux, enter the following at the command line:

  ```
  make DEBUG=y clean all
  ```

- under Microsoft Windows, open `exclusive_[VC2005|VC2008].vcproj` or `exclusive_VC2010.vcxproj` with Microsoft Visual Studio and build the `exclusive` project, with the `Debug` configuration active.

To build the release version of this example, perform the following:

- under Linux, enter the following at the command line:

  ```
  make DEBUG=n clean all
  ```

- under Microsoft Windows, open `exclusive_[VC2005|VC2008].vcproj` or `exclusive_VC2010.vcxproj` with Microsoft Visual Studio and build the `exclusive` project, with the `Release` configuration active.

To build the PROBE version of this example, perform the following:

- under Linux, enter the following at the command line:

  ```
  make DEBUG=n clean probe
  ```

- under Microsoft Windows, open exclusive_[VC2005|VC2008].vcproj or exclusive_VC2010.vcxproj with Microsoft Visual Studio and build the exclusive project, with the Probe configuration active.

Note

Under Linux, the make clean command is optional. If used, the example is completely rebuilt.

To run this example, enter the following at the command line:

- under Linux:

  ```
  ./exclusive.x
  ```

- under Microsoft Windows:

  ```
  exclusive.exe
  ```

Chapter 5
How to Create AMBA-PV Compliant Models

This chapter contains a set of guidelines for the creation of AMBA-PV-compliant models of masters, slaves and interconnects components. It contains the following sections:

5.1 How to create an AMBA-PV master

- Derive the master class from class `amba_pv_master_base` (in addition to `sc_module`).
- Instantiate one master socket of class `amba_pv_master_socket` for each connection to an AMBA bus. Specify a distinct identifier for each socket.
- Implement the method `invalidate_direct_mem_ptr()`.

  Note

  A master does not need to implement this method explicitly if it does not support DMI.

- Set every attribute of each `amba_pv_control` object before passing it as an argument to `read()`, `write()`, `burst_read()`, `burst_write()`, `get_direct_mem_ptr()`, `debug_read()`, or `debug_write()`.
- On completion of the transaction, check the returned response value.

5.2 How to create an AMBA-PV slave

- Derive the slave class from class `amba_pv_slave_base` (in addition to `sc_module`).

  Note

  A memory slave can derive from class `amba_pv_memory_base` instead.

- Instantiate one slave socket of class `amba_pv_slave_socket` for each connection to an AMBA bus. Specify a distinct identifier for each socket.
- Implement the methods `read()`, `write()`, `get_direct_mem_ptr()`, `debug_read()`, and `debug_write()`.

  Note

  A slave does not need to implement any other method than `read()` and `write()` if it does not support DMI or debug transactions.

- In the implementations of the `read()` and `write()` methods, inspect and act on the parameters, and on the attributes of the AMBA-PV extension (`amba_pv_control` object). Instead of implementing the requested functionality, a slave might choose to return an `AMBA_PV_SLVERR` error response. Return an `AMBA_PV_OKAY` response to indicate the success of the transfer.
- In the implementation of `get_direct_mem_ptr()`, either return `false`, or inspect and act on the parameters, and on the attributes of the AMBA-PV extension (`amba_pv_control` object), and set the return value and all the attributes of the DMI descriptor (class `tlm_dmi`) appropriately.
- In the implementation of `debug_read()` and `debug_write()`, either return 0, or inspect and act on the parameters, and on the attributes of the AMBA-PV extension (`amba_pv_control` object). Return the number of bytes read/written.

5.3 How to create an AMBA-PV interconnect

- Derive the interconnect class from classes `amba_pv_fw_transport_if` and `amba_pv_bw_transport_if` (in addition to `sc_module`).
- Instantiate one master or slave socket of class `amba_pv_master_socket` or `amba_pv_slave_socket`, respectively, for each connection to an AMBA bus. Specify a distinct identifier for each socket.

  Note

  The interconnect can alternatively use the class `amba_pv_socket_array` for master and slave sockets.

- Implement the method `invalidate_direct_mem_ptr()` for master sockets, and the methods `b_transport()`, `get_direct_mem_ptr()`, and `transport_dbg()` for slave sockets.

  Note

  Each master/slave socket is identified by its `socket_id`, first parameter of those methods.

- Pass on incoming method calls as appropriate on both the forward and backward paths.

  Note

  The interconnect does not need to implement the `get_direct_mem_ptr()` method explicitly if it does not support DMI. Similarly, the interconnect does not need to implement the `transport_dbg()` method explicitly if it does not support debug.

- In the implementation of `b_transport()`, the only AMBA-PV extension attributes modifiable by an interconnect component are the ID and the response attributes.
- In the implementation of `get_direct_mem_ptr()` and `transport_dbg()`, the only AMBA-PV extension attribute modifiable by a bus decoder component is the ID attribute.
- Do not modify any other attributes. A component needing to modify any other AMBA-PV extension attributes must construct a new extension object, and thereby become a master in its own right.
- Decode the generic payload address attribute on the forward path and modify the address attribute if necessary according to the location of the slave in the address map. This applies to transport, DMI, and debug interfaces.

  Note

  The interconnect can use the class `amba_pv_address_map` for representing the address map.

- In the implementation of `get_direct_mem_ptr()`, do not modify the DMI descriptor (`tlm_dmi`) attributes on the forward path. Do modify the DMI start address and end address, and DMI access attributes appropriately on the return path.
- In the implementation of `invalidate_direct_mem_ptr()`, modify the address range arguments before passing the call along the backward path.

5.4 How to create an AMBA-PV ACE master

- Derive the master class from class `amba_pv_ace_master_base` (in addition to `sc_module`).
- Instantiate one master socket of class `amba_pv_ace_master_socket` for each connection to an AMBA bus. Specify a distinct identifier for each socket.
- Implement the method `invalidate_direct_mem_ptr()`.

  Note

  An ACE master does not need to implement this method explicitly if it does not support DMI.

- Implement the methods `b_snoop()` and `snoop_dbg()`.

  Note

  An ACE master does not need to implement the method `snoop_dbg()` if it does not support debug transactions.

- Create and set an `amba_pv_extension` object. Set a pointer to this extension object in an `amba_pv_transaction` object before passing the `amba_pv_transaction` object as an argument to `b_transport()` or `transport_dbg()`.
- On completion of the transaction, check the returned response status.

5.5 How to create an AMBA-PV ACE slave

- Derive the slave class from class `amba_pv_slave_base` (in addition to `sc_module`).
- Instantiate one slave socket of class `amba_pv_ace_slave_socket` for each connection to an AMBA ACE bus. Specify a distinct identifier for each socket.
- Implement the methods `b_transport()`, `get_direct_mem_ptr()` and `transport_dbg()`.

  Note

  A slave does not need to implement any other method than `b_transport()` if it does not support DMI or debug transactions.

- In the implementations of the `b_transport()` method obtain a pointer to the `amba_pv_extension` object using `get_extension()`. Inspect and act upon the attributes in the extension object. The transaction response should be set in the extension object. Rather than implementing the requested functionality, a slave may choose to return an `AMBA_PV_SLVERR` error response. Setting an `AMBA_PV_OKAY` response indicates the success of the transfer.
- In the implementation of `get_direct_mem_ptr()`, either return `false`, or inspect and act on the parameters, and on the attributes of the AMBA-PV extension, and set the return value and all the attributes of the DMI descriptor (`class tlm_dmi`) appropriately.
- In the implementation of `transport_dbg()`, either return `0`, or obtain a pointer to the AMBA-PV extension. Inspect and act on the parameters, and on the attributes of the AMBA-PV extension. Return the number of bytes read/written.

Chapter 6
AMBA-PV Protocol Checker

This chapter describes the AMBA-PV protocol checker and the checks it performs. It contains the following sections:

6.1 Introduction

You can use the AMBA-PV protocol checker with any model that is designed to implement the AMBA-PV protocol. The behavior of the model you test is checked against the protocol by a set of checks in the protocol checker. You can instantiate the protocol checker, class `amba_pv_protocol_checker`, between any pair of AMBA-PV master and slave sockets.

The transactions that pass through are checked against the AMBA-PV protocol. You can instantiate the protocol checker, class `amba_pv_ace_protocol_checker`, between any pair of AMBA-PV ACE master and slave sockets. Errors are reported using the SystemC reporting mechanism. All errors are reported with a message type of "`amba_pv_protocol_checker`" and with a severity of `SC_ERROR`. Recommendations are reported with a severity of `SC_WARNING`. Their reporting can be disabled.

The AMBA-PV protocol checker tests your model against the AMBA AXI3 protocol by default. You can configure the protocol checker to specifically test your model against one of the ACE, AXI4, AHB or APB protocols.

Note

The AMBA-PV protocol checker does not perform any TLM 2.0 BP checks.

6.1.1 Disabling recommended checks

Table 6-1 lists the method for configuring recommended checks from the protocol checker:

Table 6-1 Reporting of protocol recommendations method

Name	Description	Allowed value	Default value
`recommend_on()`	Enable or disable reporting of protocol recommendations.	`true` or `false`	`true`, enabled

If `recommend_on(false)` is called to disable reporting of protocol recommendations, the following warning is issued:

```
Warning: amba_pv_protocol_checker: All AMBA-PV recommended rules have been disabled by
recommend_on()
```

6.1.2 Selecting AMBA protocol checks

Table 6-2 lists the method for configuring the AMBA protocol checks from the protocol checker:

Table 6-2 Selecting AMBA protocol checks method

Name	Description	Allowed value	Default value
`check_protocol()`	Select the AMBA protocol checks to perform.	`AMBA_PV_APB`, `AMBA_PV_AHB`, `AMBA_PV_AXI`[a], `AMBA_PV_AXI3`, `AMBA_PV_AXI4_LITE`, `AMBA_PV_AXI4`, `AMBA_PV_ACE_LITE`, `AMBA_PV_ACE`	`AMBA_PV_AXI3`

a. AMBA_PV_AXI is the same as AMBA_PV_AXI3. Use of AMBA_PV_AXI is deprecated.

The protocol checker tests your model against the selected AMBA protocol.

If `check_protocol` is called to select checking against a protocol other than AXI3, the following warning is issued:

`Warning: amba_pv_protocol_checker: PROTOCOL-NAME protocol rules have been selected by check_protocol()`

where *PROTOCOL-NAME* is the selected protocol.

If `check_protocol(AMBA_PV_APB)` is called to select checking against the APB protocol, the following warning is issued:

```
Warning: amba_pv_protocol_checker: APB protocol rules have been selected by
check_protocol()
```

6.2 Checks description

This section describes the checks performed by the protocol checker and indicates the area of the *AMBA AXI and ACE Protocol Specification*, the *AMBA 3 APB Protocol Specification*, or the *AMBA 3 AHB-Lite Protocol Specification* that they apply to. It contains the following subsections:

6.2.1 Architecture checks

Table 6-3 lists the architecture checks performed by the protocol checker:

Table 6-3 Architecture checks

Bus type(s)	Description of check	AMBA3 APB Protocol Specification	AMBA3 AHB-Lite Protocol Specification	AMBA4 AXI and ACE Protocol Specification
APB	The data bus can be up to 32 bits wide.	Section 4.1 “AMBA 3 APB signals”.	-	-
AHB	Recommended that the minimum data bus width is 32 bits.	-	Section 6.1 “Data bus width”.	-
AHB	The data bus can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.	-	Section 6.1 “Data bus width”.	-
AXI4-Lite	The data bus can be 32 or 64 bits wide.	-	-	Section B1.1 “Definition of AXI4-Lite”.
AXI3, AXI4, ACE-Lite	The data bus can be 32, 64, 128, 256, 512, or 1024 bits wide.	-	-	Section A1.3.1 “Channel definition”.

6.2.2 Extension checks

Table 6-4 lists the extension checks performed by the protocol checker:

Table 6-4 Extension checks

Bus type(s)	Description of check	AMBA3 APB Protocol Specification	AMBA3 AHB-Lite Protocol Specification	AMBA4 AXI and ACE Protocol Specification
"*All*"	The `amba_pv_extension` pointer cannot be NULL.	-	-	-
"*All*"	The size of any transfer must not exceed the bus width of the sockets in the transaction.	-	Section 3.4 "Transfer size".	Section A3.4.1 "Burst size".
APB, AXI4-Lite	The size of any transfer must equal the bus width of the sockets in the transaction.	Section 4.1 "AMBA 3 APB signals".	-	Section B1.1.1 "AXI4 signals not supported in AXI4-Lite".
AHB, AXI3, AXI4, ACE-Lite	The size of any transfer must be 1, 2, 4, 8, 16, 32, 64 or 128 bytes.	-	Section 3.4 "Transfer size".	Section A3.4.1 "Burst size".
APB, AXI4-Lite	All transactions are single transfers.	Section 4.1 "AMBA 3 APB signals".	-	Section B1.1.1 "AXI4 signals not supported in AXI4-Lite".
AHB	A transaction of burst type WRAP must have a length of 4, 8 or 16.	-	Section 3.5 "Burst operation".	-
AHB	A burst must have a type INCR or WRAP.	-	Section 3.5 "Burst operation".	-
AXI3, AXI4, ACE-Lite	A transaction of burst type WRAP must have a length of 2, 4, 8 or 16.	-	-	Section A3.4.1 "Burst length".
AXI3	A transaction can have a burst length 1-16.	-	-	Section A3.4.1 "Burst length".
AXI4, ACE-Lite	A transaction can have a burst length 1-256.	-	-	Section A3.4.1 "Burst length".
APB, AHB, AXI3	Quality of Service values are not supported.	Section 4.1 "AMBA 3 APB signals".	Section 2.2 "Master signals".	Section A8 "AXI4 Additional Signalling".
APB, AHB, AXI3	Region values are not supported.	Section 4.1 "AMBA 3 APB signals".	Section 2.2 "Master signals".	Section A8 "AXI4 Additional Signalling".
AXI4, ACE-Lite	Quality of Service values can be 0-15.	-	-	Section A8.1.1 "QoS interface signals".
AXI4, ACE-Lite	Region values can be 0-15.	-	-	Section A8.2.1 "Additional interface signals".

6.2.3 Address checks

Table 6-5 lists the address checks performed by the protocol checker:

Table 6-5 Address checks

Bus type(s)	Description of check	AMBA3 APB Protocol Specification	AMBA3 AHB-Lite Protocol Specification	AMBA4 AXI and ACE Protocol Specification
APB, AHB, AXI4-Lite	All transactions must have an aligned address	Section 4.1 "AMBA 3 APB signals".	Section 3.5 "Burst operation".	Section B1.1.1 "Signal list".
AHB	A burst cannot cross a 1KB boundary	-	Section 3.5 "Burst operation".	-
AXI3, AXI4, ACE-Lite	A burst cannot cross a 4KB boundary	-	-	Section A3.4.1 "Burst length".
AXI3, AXI4, ACE-Lite	A transaction with a burst type of WRAP must have an aligned address	-	-	Section A3.4.1 "Burst length".

6.2.4 Data checks

Table 6-6 lists the data checks performed by the protocol checker:

Table 6-6 Data checks

Bus type(s)	Description of check	AMBA3 APB Protocol Specification	AMBA3 AHB-Lite Protocol Specification	AMBA4 AXI and ACE Protocol Specification
"*All*"	Transaction data length is greater than or equal to the beat size times the burst length.	-	-	-
APB, AHB, AXI4-Lite	All transactions must have a NULL byte enable pointer.	Section 4.1 "AMBA 3 APB signals".	Section 2.2 "Master signals".	Section B1.1.1 "Signal list".
AXI3, AXI4, ACE-Lite	Read transactions must have a NULL byte enable pointer.	-	-	Section A2.6 "Read data channel signals".
AXI3, AXI4, ACE-Lite	The byte enable length is a multiple of the transfer size for a write transaction.	-	-	-
AHB, AXI3, AXI4, ACE-Lite	The streaming width is equal to the beat size for transactions with burst type FIXED.	-	-	-

6.2.5 Response checks

Table 6-7 lists the response checks performed by the protocol checker:

Table 6-7 Response checks

Bus type(s)	Description of check	AMBA3 APB Protocol Specification	AMBA3 AHB-Lite Protocol Specification	AMBA4 AXI and ACE Protocol Specification
APB, AXI4-Lite	A response array is not appropriate as all transactions are single transfers	Section 4.1 “AMBA 3 APB signals”.	-	Section B1.1.1 “AXI4 signals not supported in AXI4-Lite”.
APB, AHB	A response can be OKAY or SLVERR	Section 2.1 “AMBA APB signals”.	Section 5.1 “Slave transfer response”.	-
AXI4-Lite	An EXOKAY response is not supported	-	-	Section B1.1.1 “AXI4 signals modified in AXI4-Lite”.
AXI3, AXI4, ACE-Lite	An EXOKAY response can only be given to an exclusive transaction	-	-	A3.4.4 “Read and write response structure”.

6.2.6 Exclusive access checks

Table 6-8 lists the exclusive access checks performed by the protocol checker:

Table 6-8 Exclusive access checks

Bus type(s)	Description of check	AMBA3 APB Protocol Specification	AMBA3 AHB-Lite Protocol Specification	AMBA4 AXI and ACE Protocol Specification
APB, AXI4-Lite	A transaction cannot be exclusive or locked.	Section 4.1 “AMBA 3 APB signals”	-	Section B1.1.1 “AXI4 signals not supported in AXI4-Lite”.
AHB	A transaction cannot be exclusive.	-	Section 2.2 “Master signals”	-
AXI3	A transaction cannot be exclusive and locked.	-	-	Section A7.4 “Atomic access signaling”.
AXI3	Recommended that locked transactions are only used to support legacy devices.	-	-	Section A7.4.1 “Legacy considerations”.
AXI4, ACE-Lite	Locked accesses are not supported.	-	-	Section A7.3 “Locked accesses”.
AXI3, AXI4, ACE-Lite	The maximum number of bytes that can be transferred in an exclusive burst is 128.	-	-	Section A7.2.4 “Exclusive access restrictions”.

Table 6-8 Exclusive access checks (continued)

Bus type(s)	Description of check	AMBA3 APB Protocol Specification	AMBA3 AHB-Lite Protocol Specification	AMBA4 AXI and ACE Protocol Specification
AXI3, AXI4, ACE-Lite	The number of bytes transferred in an exclusive access burst must be a power of 2.	-	-	Section A7.2.4 “Exclusive access restrictions”.
AXI4	The burst length for an exclusive access must not exceed 16 transfers.	-	-	Section A7.2.4 “Exclusive access restrictions”.
AXI3, AXI4, ACE-Lite	The address of an exclusive transaction is aligned to the total number of bytes in the transaction.	-	-	Section A7.2.4 “Exclusive access restrictions”.
AXI3, AXI4, ACE-Lite	Recommended that every exclusive write has an earlier outstanding exclusive read with the same ID.	-	-	Section A7.2.4 “Exclusive access restrictions”.
AXI3, AXI4, ACE-Lite	Recommended that the address, size and length of an exclusive write with a given ID is the same as the address, size and length of the preceding exclusive read with the same ID.	-	-	Section A7.2.4 “Exclusive access restrictions”.

6.2.7 Cacheability checks

Table 6-9 lists the cacheability checks performed by the protocol checker:

Table 6-9 Cacheability checks

Bus type(s)	Description of check	AMBA3 APB Protocol Specification	AMBA3 AHB-Lite Protocol Specification	AMBA4 AXI and ACE Protocol Specification
APB, AXI4-Lite	All transactions are non-cacheable, non-bufferable.	Section 4.1 “AMBA 3 APB signals” list no signals.	-	Section B1.1.1 “AXI4 signals not supported in AXI4-Lite”.
AHB	Allocate attributes are not supported.	-	Section 2.2 “Master signals” lists no signals.	-
AXI3, AXI4, ACE-Lite	When a transaction is not modifiable then allocate attributes are not set.	-	-	Section A4.4 “Memory types”.
APB, AHB, AXI3, AXI4, AXI4-Lite	Cache coherent transactions are not supported.	-	-	Section C1.3.2 “Changes to existing AXI channels”.
ACE-Lite, ACE	A barrier transaction must have a barrier transaction type.	-	-	Table C3-7 “Permitted read address control signal combinations”.

Table 6-9 Cacheability checks (continued)

Bus type(s)	Description of check	AMBA3 APB Protocol Specification	AMBA3 AHB-Lite Protocol Specification	AMBA4 AXI and ACE Protocol Specification
ACE	A coherent transaction must be inner or outer shareable.	-	-	Table C3-7 “Permitted read address control signal combinations” and Table C3-8 “Permitted write address control signal combinations”.
ACE-Lite	The only permitted coherent transaction type is ReadOnce.	-	-	Table C3-11 “ACE-Lite permitted read address control signal combinations”.
ACE, ACE-Lite	A cache maintenance transaction cannot target the system domain.	-	-	Table C3-7 “Permitted read address control signal combinations”.
ACE, ACE-Lite	A DVM transaction must be inner or outer shareable.	-	-	Table C3-7 “Permitted read address control signal combinations”.
ACE, ACE-Lite	The permitted read transaction groups are Non-snooping, Coherent, Cache maintenance, Barrier and DVM.	-	-	Table C3-7 “Permitted read address control signal combinations” and Table C3-11 “ACE-Lite permitted read address control signal combinations”.
ACE-Lite	Memory update transactions are not permitted.	-	-	Table C3-12 “ACE-Lite permitted write address control signal combinations”.
ACE	A WriteClean or WriteBack transaction cannot target the system domain.	-	-	Table C3-8 “Permitted write address control signal combinations”.
ACE	An Evict transaction must be inner or outer shareable.	-	-	Table C3-8 “Permitted write address control signal combinations”.
ACE, ACE-Lite	The permitted write transaction groups are Non-snooping, Coherent, Memory update (ACE) and Barrier.	-	-	Table C3-8 “Permitted write address control signal combinations” and Table C3-12 “ACE-Lite permitted write address control signal combinations”.
ACE	Snoop transaction type must be ReadOnce, ReadShared, ReadClean, ReadNotSharedDirty, ReadUnique, CleanShared, CleanInvalid, MakeInvalid, DVMComplete or DVMMessage.	-	-	Table C3-19 “ACSNOOP encodings”.