AMBA-PV Extensions to TLM

Version 2.0

User Guide

Non-Confidential

Issue 14
100962_0200_14_en

AMBA-PV Extensions to TLM User Guide

This document (100962_0200_14_en) was issued on 2024-09-16. There might be a later issue at http://developer.arm.com/documentation/100962

The product version is 2.0.

See also: Proprietary Notice | Product and document information | Useful resources

Start Reading

If you prefer, you can skip to the start of the content.

Intended audience

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

You must be familiar with:

- The basic concepts of C++ such as classes and inheritance
- SystemC and TLM standards

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com.

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Contents

1. Introduction to AMBA-PV Extensions to TLM 2.0

The *AMBA-PV Extensions to TLM 2.0* (AMBA-PV) map AMBA® buses on top of TLM 2.0.

Its key features are:

- Dedicated to the *Programmer's View* (PV), it focuses on high-level, functionally accurate, transaction modeling. Low-level signals, for example, channel handshake, are not important at that level.
- It is the standard for modeling 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.

1.1 AMBA-PV classes and interfaces

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 that is 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, including exclusive accesses and atomic transactions.

Note

To use atomic transactions directly, use Fast Models version 11.27 or later.

- *Quality of Service* (QoS) support.
- Multiple region support.
- Coherency support.
- Barrier transactions.
- *Distributed Virtual Memory* (DVM) support.

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

- Definition of AMBA-PV extension and trait 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, for example, interrupts.

2. AMBA-PV extension class

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 accesses using exclusive accesses, locked accesses, or atomic transactions.

Note

- Arm recommends that you use locked accesses only 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 support locked accesses.
- To use atomic transactions directly, use Fast Models version 11.27 or later.

- 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 class 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 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.

When 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 nonzero, 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.

You must use the AMBA-PV Extension class 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 *IEEE Standard for Standard SystemC® Language Reference Manual*, January 2012. The AMBA-PV Extension class 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.

2.1 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 describes these attributes and functions.

2.1.1 Class definitions

This section describes the class definitions.

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 used as an argument to the user layer interface methods.

```
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);

enum amba_pv_physical_address_space_t {
    AMBA_PV_SECURE_PAS              = 0x0,
    AMBA_PV_NON_SECURE_PAS          = 0x1,
    AMBA_PV_ROOT_PAS                = 0x2,
    AMBA_PV_REALM_PAS               = 0x3
};

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_physical_address_space(amba_pv_physical_address_space_t);
        amba_pv_physical_address_space_t get_physical_address_space() 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_gathering(bool = true);
        bool is_gathering() const;
        void set_reordering(bool = true);
        bool is_reordering() const;
        void set_transient(bool = true);
        bool is_transient() const;
        void set_translated_access(bool);
        bool is_translated_access() const;
        void set_mmu_flow_type(amba_pv_mmuflow_t mmu_flow_type);
        amba_pv_mmuflow_t get_mmu_flow_type() 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;
        void set_user(unsigned int);
        unsigned int get_user() 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);
}
```

Related information

User layer on page 42

2.1.2 Constructors, copying, and addressing

The default constructors must set the AMBA-PV extension attributes to their default values.

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 attribute 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.

2.1.3 Default values and modifiability of attributes

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

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

Attribute	Default value	Modifiable by interconnect	Modifiable by slave	Notes
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	No	An example of Modifiable by interconnect is an exclusive monitor that flattens the exclusive access before passing it downstream.
Locked	`false`	No	No	-
Bufferable	`false`	No	No	-

Attribute	Default value	Modifiable by interconnect	Modifiable by slave	Notes
Modifiable/ cacheable	`false`	No	No	The modifiable attribute is identical to the cacheable attribute but has been renamed in AXI4 to better describe the required functionality.
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`	No	No	`AMBA_PV_WRITE_NO_SNOOP` and `AMBA_PV_READ_NO_SNOOP` have the same encoding representation.
Bar	`AMBA_PV_ RESPECT_BARRIER`	No	No	-
Response	`AMBA_PV_OKAY`	Yes	Yes	-
PassDirty	`false`	Yes	Yes	-
IsShared	`false`	Yes	Yes	-
DataTransfer	`false`	Yes	Yes	Only a valid response to upstream snoops, typically from interconnect to master.
Error	`false`	Yes	Yes	Only a valid response to upstream snoops, typically from interconnect to master.
WasUnique	`false`	Yes	Yes	Only a valid response to upstream snoops, typically from interconnect to master.
ResponseArray	`null`	No	No	-
ResponseArray complete	`false`	Yes	Yes	-

If an AMBA-PV extension object is re-used, the modifiability rules 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.

2.1.4 Burst length attribute

This 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.

The method `set_length()` must set this attribute to the value passed as argument. The method `get_length()` must return the value of this attribute.

The default value of this attribute must be 1, for single transfer.

This 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, AXI5, and ACE buses is 256.

Related information

Extension checks on page 75

2.1.5 Burst size attribute

This 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 method `set_size()` must set this attribute to the value passed as argument. The method `get_size()` must return the value of this attribute.

The value of this 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 this attribute must be 8, for 64-bit wide transfer.

This attribute is specific to the AXI, ACE, and AHB buses. It is ignored for transactions modeling transfers on the APB bus.

2.1.6 Burst type attribute

This attribute specifies the burst type.

The method `set_burst()` must set this attribute to the value passed as an argument. The method `get_burst()` must return the value of this attribute.

A transaction with a burst type attribute value of `AMBA_PV_WRAP` must have an aligned address.

Note

AXI5 has an exception to this rule. According to the AMBA AXI Protocol Specification, an AtomicCompare transaction of type `AMBA_PV_WRAP` must have an address that aligns with half the total transaction size. Total transaction size = burst size*burst length.

The default value of this attribute must be `AMBA_PV_INCR`, for incrementing burst.

This attribute is specific to the AXI, ACE, and AHB buses. It is ignored for transactions modeling transfers on the APB bus.

2.1.7 ID attribute

This attribute is mainly used for exclusive accesses.

The method `set_id()` must set this attribute to the value passed as argument. The method `get_id()` must return the value of this attribute.

This attribute must be set by the master originating the transaction. The interconnect must modify this attribute to ensure its uniqueness across all its masters before passing the transaction to the addressed slave.

The default value of this attribute must be 0.

This attribute is specific to the AXI, ACE, and AHB buses. It is ignored for transactions modeling transfers on the APB bus.

2.1.8 Privileged attribute

This attribute enables masters to indicate their processing mode. A privileged transaction typically has a greater level of access within the system.

The method `set_privileged()` must set this attribute to the value passed as argument. The method `is_privileged()` must return the value of this attribute.

The default value of this attribute must be `false`.

This attribute is specific to the AXI, ACE, and AHB buses. It is ignored for transactions modeling transfers on the APB bus.

2.1.9 Non-secure attribute

This attribute enables differentiating between secure and non-secure transactions.

The method `set_non_secure()` must set this attribute to the value passed as argument. The method `is_non_secure()` must return the value of this attribute.

The default value of this attribute must be `false`.

This attribute is specific to the AXI and ACE buses. It is ignored for transactions modeling transfers on the AHB and APB buses.

2.1.10 Exclusive attribute

This attribute selects exclusive access, and the response attribute indicates the success or failure of the exclusive access.

The method `set_exclusive()` must set this attribute to the value passed as argument. The method `is_exclusive()` must return the value of this attribute.

The AMBA-PV package provides an exclusive monitor model 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.

Arm recommends 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.

Arm recommends that every exclusive write has an earlier outstanding exclusive read with the same value for the ID attribute.

Arm recommends 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.

Atomic transactions cannot be exclusive, according to the AMBA AXI Protocol Specification.

This attribute must not have the value `true` together with the locked attribute.

The default value of this attribute must be `false`.

This attribute is specific to the AXI and ACE buses. It is ignored for transactions modeling transfers on the AHB and APB buses.

Related information

Exclusive monitor on page 47
Response attribute on page 27

2.1.11 Locked attribute

Locked transactions, those for which this 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.

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

Arm recommends that locked accesses are only used to support legacy devices. Locked transactions are currently not supported by the AMBA-PV bus decoder.

This attribute must not have the value `true` together with the exclusive attribute.

The default value of this attribute must be `false`.

This attribute is specific to the AXI3 and AHB buses. It is ignored for transactions modeling transfers on the APB, AXI4, AXI5, and ACE buses.

2.1.12 Bufferable attribute

This attribute specifies whether or not the associated transaction is bufferable.

The method `set_bufferable()` must set this attribute to the value passed as argument. The method `is_bufferable()` must return the value of this attribute.

A bufferable transaction can be delayed in reaching its final destination. This is usually only relevant to writes.

The default value of this attribute must be `false`.

This attribute is specific to the AXI and AHB buses. It is ignored for transactions modeling transfers on the APB bus.

2.1.13 Modifiable/cacheable attribute

The modifiable attribute specifies whether the associated transaction is modifiable.

The methods `set_modifiable()` and `set_cacheable()` must set this attribute to the value passed as an argument. The methods `is_modifiable()` and `is_cacheable()` must return the value of this attribute.

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 attribute with the read allocate and write allocate attributes.

The default value of this attribute must be `false`.

This 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 to this attribute for AXI4, AXI5, and ACE to better describe the required function of the attribute. The actual functionality is unchanged.

Related information

Read allocate attribute on page 20
Write allocate attribute on page 21

2.1.14 Read allocate attribute

This attribute specifies whether or not this transaction must be allocated if it is a read and it misses in the cache.

The method `set_read_allocate()` must set this attribute to the value passed as argument. The method `is_read_allocate()` must return the value of this 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 this attribute must be `false`.

This attribute is specific to the AXI and ACE buses. It is ignored for transactions modeling transfers on the AHB and APB buses.

2.1.15 Write allocate attribute

This attribute specifies whether or not this transaction must be allocated if it is a write and it misses in the cache.

The method `set_write_allocate()` must set this attribute to the value passed as argument. The method `is_write_allocate()` must return the value of this 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 this attribute must be `false`.

This attribute is specific to the AXI and ACE buses. It is ignored for transactions modeling transfers on the AHB and APB buses.

2.1.16 Read other allocate attribute

This 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 this attribute to the value passed as argument. The method `is_read_other_allocate()` returns the value of this attribute.

The default value of this attribute is `false`.

This attribute is specific to the AXI4, AXI5, 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()`.

2.1.17 Write other allocate attribute

This 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 this attribute to the value passed as argument. The method `is_write_other_allocate()` returns the value of this 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 this attribute is `false`.

This 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 read allocate attribute methods `set_read_allocate()` and `is_read_allocate()`.

2.1.18 Quality of Service (QoS) attribute

This attribute supports *Quality of Service* (QoS) schemes.

The bus protocol does not specify the exact use of the QoS identifier but recommends that it is used as a priority indicator.

The method `set_qos()` sets this attribute to the value passed as argument. The method `get_qos()` returns the value of this attribute.

The default value of this attribute is `0`, which indicates that the interface is not participating in any `QoS` scheme.

This attribute is specific to the AXI4, AXI5, and ACE buses. It is ignored for transactions modeling transfers on the AXI3, AHB and APB buses.

For AXI4, AXI5, and ACE this indicator attribute value must be between 0 and 15 inclusive.

2.1.19 Region attribute

This attribute supports multiple region interfaces. It uniquely identifies a region.

The method `set_region()` sets this attribute to the value passed as argument. The method `get_region()` returns the value of this attribute.

The default value of this attribute is `0`.

This attribute is specific to the AXI4, AXI5, and ACE buses. It is ignored for transactions modeling transfers on the AXI3, AHB and APB buses.

For AXI4, AXI5, and ACE the value of this indicator attribute must be between 0 and 15 inclusive.

2.1.20 Domain attribute

This attribute indicates the shareability domain for a transaction.

The method `set_domain()` sets this attribute to the value passed as argument. The method `get_domain()` returns the value of this attribute.

The default value of this attribute is `AMBA_PV_NON_SHAREABLE`.

This attribute is specific to ACE buses. It is ignored for transactions modeling transfers on the AXI, AHB and APB buses.

The encoding of the value of this attribute exactly matches the encoding used on the ACE channels AWDOMAIN and ARDOMAIN.

2.1.21 Snoop attribute

This attribute specifies the transaction type for shareable transactions.

The method `set_snoop()` sets this attribute to the value passed as argument. The method `get_snoop()` returns the value of this attribute.

The default value of this 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 dependent on the domain and bar attribute values and whether the transaction is a read or a write.

This attribute is specific to ACE buses. It is ignored for transactions modeling transfers on the AXI, AHB, and APB buses.

The encoding of this attribute value exactly matches the encoding used on the ACE channels AWSNOOP and ARSNOOP.

For atomic transactions, AWSNOOP must be set to all zeros, according to the AMBA AXI Protocol Specification.

2.1.22 Bar attribute

This attribute indicates barrier information for the transaction.

The method `set_bar()` sets this attribute to the value passed as argument. The method `get_bar()` returns the value of this attribute.

The default value of this attribute is `AMBA_PV_RESPECT_BARRIER`.

This attribute is specific to ACE buses. It is ignored for transactions modeling transfers on the AXI, AHB and APB buses.

The encoding of this attribute value exactly matches the encoding used on the ACE channels AWBAR and ARBAR.

2.1.23 DVM messages

To provide a *Programmer's View* (PV) 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.

2.1.23.1 DVM default values

This section defines the DVM default values.

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

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

2.1.23.2 DVM VMID attribute

This attribute defines the Virtual Machine Identifier for some DVM operations.

The method `is_dvm_vmid_set()` returns `true` if this attribute has been set. If the VMID attribute has not been set then this attribute value should not be used.

The method `get_dvm_vmid()` returns the value of this attribute. The method `set_dvm_vmid()` sets the value of this attribute.

This attribute is not set by default. The default value of this attribute is `0`.

2.1.23.3 DVM ASID attribute

This attribute defines the Address Space Identifier for some DVM operations.

The method `is_dvm_asid_set()` returns true if this attribute has been set. If this attribute has not been set then this attribute value should not be used.

The method `get_dvm_asid()` returns the value of this attribute. The method `set_dvm_asid()` sets the value of this attribute.

This attribute is not set by default. The default value of this attribute is `0`.

2.1.23.4 DVM Virtual Index attribute

You can use this attribute as part of the physical address by physical instruction cache invalidate DVM messages.

The method `is_dvm_virtual_index_set()` returns `true` if this attribute has been set. If this attribute has not been set then this attribute value should not be used.

The method `get_dvm_virtual_index()` returns the value of this attribute. The method `set_dvm_virtual_index()` sets the value of this attribute.

This attribute is not set by default. The default value of this attribute is `0`.

2.1.23.5 DVM Completion attribute

This attribute identifies whether completion is required for DVM Sync messages.

The method `is_dvm_completion_set()` returns `true` if this attribute has been set. The method `set_dvm_completion()` sets the value of this attribute.

By default this attribute has the value `false`.

2.1.23.6 DVM Message type attribute

This attribute specifies the required DVM operation.

The method `get_dvm_message_type()` returns the value of this attribute. The method `set_dvm_message_type()` sets the value of this attribute.

By default this attribute has the value `AMBA_PV_TLB_INVALIDATE`.

2.1.23.7 DVM Operating system attribute

This attribute specifies the operating system that the DVM operation applies to.

The method `get_dvm_os()` returns the value of this attribute. The method `set_dvm_os()` sets the value of this attribute.

By default this attribute has the value `AMBA_PV_HYPERVISOR_OR_GUEST`.

2.1.23.8 DVM Security attribute

This attribute specifies how the DVM operation applies to the secure and non-secure worlds.

The method `get_dvm_security()` returns the value of this attribute. The method `set_dvm_security()` sets the value of this attribute.

By default this attribute has the value `AMBA_PV_SECURE_AND_NON_SECURE`.

2.1.23.9 DVM Additional address attribute

This attribute defines the additional address required by some DVM operations.

The method `is_dvm_additional_address_set()` returns `true` if this attribute has been set. If this attribute has not been set then this attribute value should not be used.

The method `get_dvm_additional_address()` returns the value of this attribute. The method `set_dvm_additional_address()` sets the value of this attribute.

This attribute is not set by default. The default value of this attribute is `0`.

2.1.23.10 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 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.

2.1.24 Response attribute

This section describes the response attribute.

The method `set_resp()` must set the response attribute to the value passed as argument. The method `get_resp()` must return the value of the response attribute.

The method `is_okay()` must return `true` if and only if the 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 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 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 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`.

The method `is_incomplete()` must return `true` if and only if the value of the response attribute is `AMBA_PV_INCOMPLETE`. The method `set_incomplete()` must set the value of the response attribute to `AMBA_PV_INCOMPLETE`.

Table 2-3: AMBA-PV responses

Value	Interpretation
`AMBA_PV_OKAY`	A normal access success, or an exclusive access failure.
`AMBA_PV_EXOKAY`	Either the read or write portion of an exclusive access has been successful.
`AMBA_PV_SLVERR`	The access has reached the slave successfully, but the slave returned an error condition to the originating master.
`AMBA_PV_DECERR`	There is no slave at the transaction address. This is typically generated by an interconnect component.

Value	Interpretation
AMBA_PV_INCOMPLETE	The slave did not attempt to perform the access.

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.

Arm recommends 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.

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. The exclusive attribute of the associated transaction must have a value of true.
AMBA_PV_SLVERR	TLM_GENERIC_ERROR_RESPONSE, TLM_COMMAND_ERROR_RESPONSE, TLM_BURST_ERROR_RESPONSE, TLM_BYTE_ENABLE_ERROR_RESPONSE
AMBA_PV_DECERR	TLM_ADDRESS_ERROR_RESPONSE
AMBA_PV_INCOMPLETE	TLM_INCOMPLETE_RESPONSE

2.1.25 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 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 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`.

2.1.26 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 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 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 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`.

2.1.27 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`.

2.1.27.1 Response array element attributes

These attributes have the same semantics and accessors as the equivalent response attributes.

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

Attribute	Default value	Set methods	Get methods
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()`

Related information

Response attribute on page 27
ACE response attributes PassDirty and IsShared on page 28

ACE snoop response attributes DataTransfer, Error, and WasUnique on page 29

2.1.28 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:

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

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

2.1.29 Direct memory interface

For the AMBA-PV protocol, any of the AMBA-PV extension attributes can further indicate the address of the requested DMI access. The master must set them.

The slave can service DMI requests differently depending on the value of any AMBA-PV extension attributes. Arm recommends that the master sets all AMBA-PV extension attributes before requesting DMI access.

Related information

Default values and modifiability of attributes on page 14

2.1.30 Debug transport interface

For the AMBA-PV protocol, any of the AMBA-PV extension attributes can further indicate the address of the debug access. The master must set them.

The slave can service debug accesses differently depending on the value of any AMBA-PV extension attributes. Arm® recommends that the master sets all AMBA-PV extension attributes before performing debug accesses.

Related information

Default values and modifiability of attributes on page 14

2.1.31 Physical address space attribute

This attribute enables differentiating between secure, non-secure, root, or realm transactions.

The method `set_physical_address_space()` sets this attribute to the value passed as the argument. The method `get_physical_address_space()` returns the value of this attribute.

The default value is secure (`AMBA_PV_SECURE_PAS`).

2.1.32 Atomic attributes

The AWATOP atomic signals are modeled by class `amba_pv_atomic`, which `amba_pv_extension` inherits from.

`amba_pv_atomic` has three members that model AWATOP:

- `amba_pv_atomic_op_t m_atomic_op`
- `amba_pv_atomic_subop_t m_atomic_subop`
- `amba_pv_atomic_endianness_t m_atomic_endianness`

See the inline comments in `amba_pv_atomic.h` for detailed explanations.

As only a subset of enum values can form a valid signal, the `amba_pv_atomic` class has some helper functions to check if the member represents a valid signal. See the comments in the `amba_pv_atomic_op_t` type definition for details.

Atomic signals are supported for AXI5, ACE5-Lite, and ACE5-LiteDVM.

2.1.33 Untranslated transactions attributes

AxMMUFLOW is the only signal to support untranslated transactions that is modeled. It is modeled as `m_mmu_flow_type`.

The helper functions `is_translated_access()` and `set_translated_access()` rely on the value of `m_mmu_flow_type`. A value of AxMMUATST indicates that the transaction has already undergone PCIe ATS translation. It is equivalent to AxMMUFLOW[0] when AxMMUFLOW[1] is deasserted, according to the AMBA AXI Protocol Specification.

2.2 AMBA signal mapping

This section describes the relationships between the AMBA® hardware signals and the private attributes of the AMBA-PV extension and the TLM 2.0 Generic Payload.

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

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

Table 2-6: Address channels

Signal	Variable	Description
AxID	`amba_pv_control::m_id`	ID.
AxADDR	`tlm_generic_payload::m_address`	Address.
AxADDR	`amba_pv_extension::m_dvm_transaction`	DVM message attributes.
AxLEN	`amba_pv_extension::m_length`	Burst length.
AxSIZE	`amba_pv_extension::m_size`	Burst size.
AxBURST	`amba_pv_extension::m_burst`	Burst type.
AxLOCK	`amba_pv_control::m_exclusive` `amba_pv_control::m_locked`	Lock type.
AxCACHE	`amba_pv_control::m_bufferable` `amba_pv_control::m_modifiable` `amba_pv_control::m_axcache_allocate_bit2` `amba_pv_control::m_axcache_allocate_bit3`	Cache type.
AxPROT	`amba_pv_control::m_privileged` `amba_pv_control::m_non_secure` `amba_pv_control::m_instruction`	Protection type.
AxQOS	`amba_pv_control::m_qos`	Quality of service type.
AxREGION	`amba_pv_control::m_region`	Region type.
AxDOMAIN	`amba_pv_control::m_domain`	Domain type.
AxSNOOP	`amba_pv_control::m_snoop`	Snoop type.
AxBAR	`amba_pv_control::m_bar`	Barrier type.
AxUSER	`amba_pv_control::m_user`	User defined signals.
AxMMUFLOW	`amba_pv_control::m_mmu_flow_type`	MMU flow type.

Table 2-7: Write data and response channels

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

Signal	Variable	Description
AWATOP	`amba_pv_atomic::m_atomic_op`, `amba_pv_atomic::m_atomic_subop`, `amba_pv_atomic::m_atomic_endianness`	Atomic transaction opcode

Table 2-8: Read data channels

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

Table 2-9: Snoop data channels

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

Table 2-10: Unmapped signals

Signal	Variable	Description
xVALID	Not applicable at PV level.	Address/data/response valid.
xREADY	Not applicable at PV level.	Address/data/response ready.
xLAST	Not applicable at PV level.	Read/write last.
xACK	Not applicable at PV level.	Read/Write acknowledge.

2.3 Mapping for AMBA buses

This section describes the control signal mappings, response mappings, and response bit mappings for AMBA® buses.

The following table shows the control signal mappings for AXI, ACE, and AHB buses. The APB bus does not use these control signals.

Table 2-11: Signal mappings for amba_pv_control

amba_pv_control	ACE, ACE-Lite	AXI4	AXI3	AHB	AMBA5 AHB	CHI
`bool is_privileged() const; void set_privileged(bool = true);`	AxPROT[0]	AxPROT[0]	AxPROT[0]	HPROT[1]	HPROT[1]	-
`bool is_instruction() const; void set_instruction(bool = true);`	AxPROT[2]	AxPROT[2]	AxPROT[2]	HPROT[0]	HPROT[0]	-
`bool is_non_secure() const; void set_non_secure(bool = true);`	AxPROT[1]	AxPROT[1]	AxPROT[1]	-	-	NS
`bool is_locked() const; void set_locked(bool = true);`	-	-	AxLOCK = 2	HLOCK	HLOCK	-
`bool is_exclusive() const; void set_exclusive(bool = true);`	AxLOCK	AxLOCK	AxLOCK = 1	-	-	Excl

amba_pv_control	ACE, ACE-Lite	AXI4	AXI3	AHB	AMBA5 AHB	CHI
`void set_bufferable(bool = true); bool is_bufferable() const;`	AxCACHE[0]	AxCACHE[0]	AxCACHE[0]	HPROT[2]	HPROT[2]	MemAttr[3:0], SnpAttr[1:0], Order[1:0]
`void set_cacheable(bool = true); bool is_cacheable() const;`	-	-	AxCACHE[1]	HPROT[3]	HPROT[3]	MemAttr[2]
`void set_modifiable(bool = true); bool is_modifiable() const;`	AxCACHE[1]	AxCACHE[1]	-	-	-	MemAttr[3:0], SnpAttr[1:0], Order[1:0]
`void set_read_allocate(bool = true); bool is_read_allocate() const;`	AxCACHE[2]	AxCACHE[2]	AxCACHE[2]	-	HPROT[5]	MemAttr[3]
`void set_write_allocate(bool = true); bool is_write_allocate() const;`	AxCACHE[3]	AxCACHE[3]	AxCACHE[3]	-	HPROT[5]	MemAttr[3]
`void set_read_other_allocate(bool = true); bool is_read_other_ allocate() const;`	AxCACHE[3]	AxCACHE[3]	-	-	HPROT[4]	-
`void set_write_other_allocate(bool = true); bool is_write_other_ allocate() const;`	AxCACHE[2]	AxCACHE[2]	-	-	HPROT[4]	-
`void set_qos(unsigned int); unsigned int get_qos() const;`	AxQOS[3:0]	AxQOS[3:0]	-	-	-	QOS[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;`. See Note after table.	AxDOMAIN[1:0]	-	-	-	HPROT[6]	SnpAttr[1:0]
`void set_snoop(amba_pv_snoop_t); amba_pv_snoop_t get_snoop() const;`	AxSNOOP[3:0]	-	-	-	-	REQ channel Opcode[4:0]
`void set_bar(amba_pv_bar_t); amba_ pv_bar_t get_bar() const;`	AxBAR[1:0]	-	-	-	-	-
`void set_user(unsigned int); unsigned int get_user() const;`	AxUSER	AxUSER	AxUSER	HxUSER	HxUSER	-

Note

For masters use:

```
set_domain(HPROT[6] ? AMBA_PV_INNER_SHAREABLE : AMBA_PV_NON_SHAREABLE)
```

For slaves use:

```
HPROT[6] = get_domain() != AMBA_PV_NON_SHAREABLE
```

The following table shows the response mappings for AXI, ACE, AHB, and APB buses:

Table 2-12: Response mappings for amba_pv_resp_t

amba_pv_resp_t	AXI xRESP	AHB HRESP	AMBA5 AHB	APB PSLVERR
`AMBA_PV_OKAY`	OKAY	OKAY	OKAY	LOW
`AMBA_PV_EXOKAY`	EXOKAY	-	EXOKAY	-
`AMBA_PV_SLVERR`	SLVERR	ERROR	ERROR	HIGH
`AMBA_PV_DECERR`	DECERR	ERROR	ERROR	HIGH

Note

PSLVERR signal support is not a requirement for APB peripherals. If a peripheral does not support this signal then the corresponding appropriate response is `AMBA_PV_OKAY`.

The following table shows the additional response bit mappings for the ACE bus:

Table 2-13: Mappings for additional ACE bus 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]	-

2.4 Basic transactions

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

2.4.1 Fixed burst example

This example shows a fixed read burst of four transfers.

In this figure each row represents a transfer:

Figure 2-1: Fixed read burst of four transfers

Address: `0x0`
Burst size: 32 bits
Burst type: fixed
Burst length: 4 transfers

3	2	1	0	`m_data[0..3]`	`m_address = 0x0`
7	6	5	4	`m_data[4..7]`	`m_address = 0x0`
B	A	9	8	`m_data[8..11]`	`m_address = 0x0`
F	E	D	C	`m_data[12..15]`	`m_address = 0x0`

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.

2.4.2 Incremental burst example

This example shows an incremental write burst of four transfers.

In this figure each row represents a transfer:

Figure 2-2: Incremental write burst of four transfers

Address: `0x0`
Burst size: 32 bits
Burst type: incremental
Burst length: 4 transfers

3	2	1	0	`m_data[0..3]`	`m_address = 0x0`
7	6	5	4	`m_data[4..7]`	`m_address = 0x4`
B	A	9	8	`m_data[8..11]`	`m_address = 0x8`
F	E	D	C	`m_data[12..15]`	`m_address = 0xC`

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;
```

2.4.3 Wrapped burst example

This example shows a wrapped burst of four transfers.

In this figure, each row represents a transfer:

Figure 2-3: Wrapped burst of four transfers

Address: `0x4`
Burst size: 32 bits
Burst type: wrapped
Burst length: 4 transfers

7	6	5	4	`m_data[0..3]`	`m_address = 0x4`
B	A	9	8	`m_data[4..7]`	`m_address = 0x8`
F	E	D	C	`m_data[8..11]`	`m_address = 0xC`
3	2	1	0	`m_data[12..15]`	`m_address = 0x0`

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 = 0x4;
m_data_length = 16;
m_streaming_width = 16;
```

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

```
m_burst = AMBA_PV_WRAP;
```

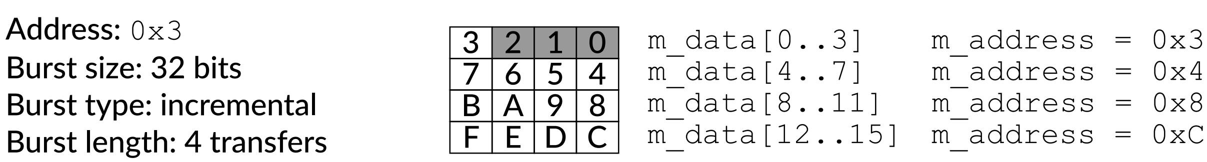
```
m_length = 4;
m_size = 4;
```

2.4.4 Unaligned burst example

This example shows an unaligned incremental write burst of four transfers.

In this figure each row 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 on 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.

3. AMBA-PV classes

This chapter describes the AMBA-PV class hierarchy and each major class.

3.1 Class description

This section describes the relationships between the AMBA-PV classes and interfaces (which use the `amba_pv` namespace) and TLM 2.0 classes and interfaces.

3.1.1 AMBA-PV extension

The AMBA-PV extension class (`amba_pv_extension`) 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.

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

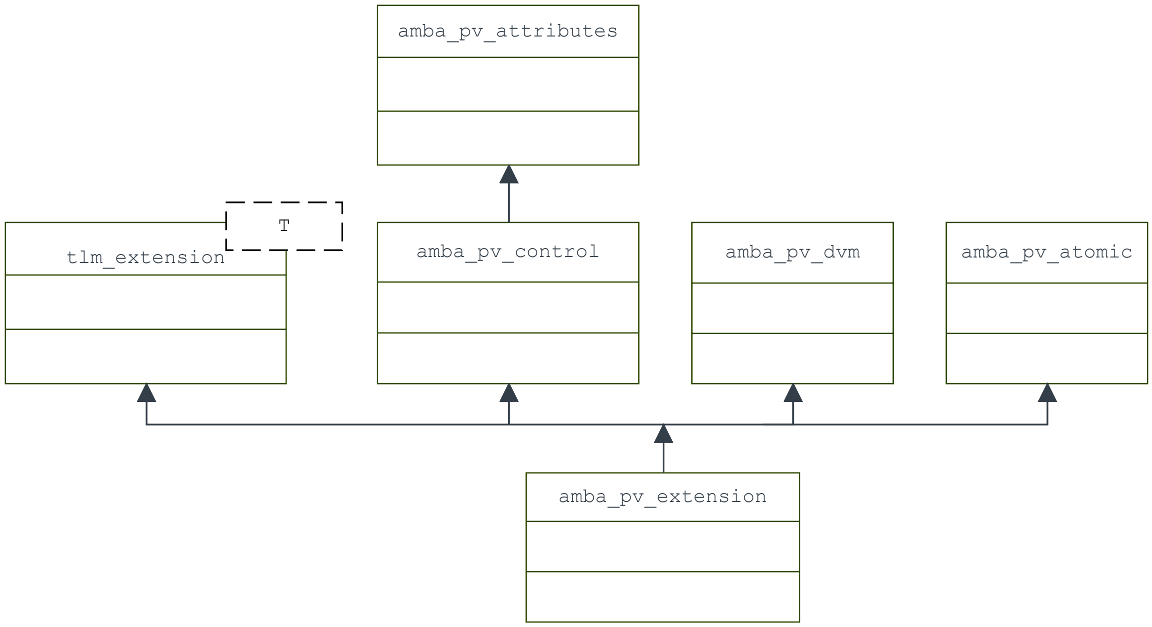
The atomic operations are modeled by the `amba_pv_atomic` class.

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



Related information

User layer on page 42

3.1.2 Core interfaces

The AMBA-PV core interfaces comprise transport and snoop interfaces.

`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`

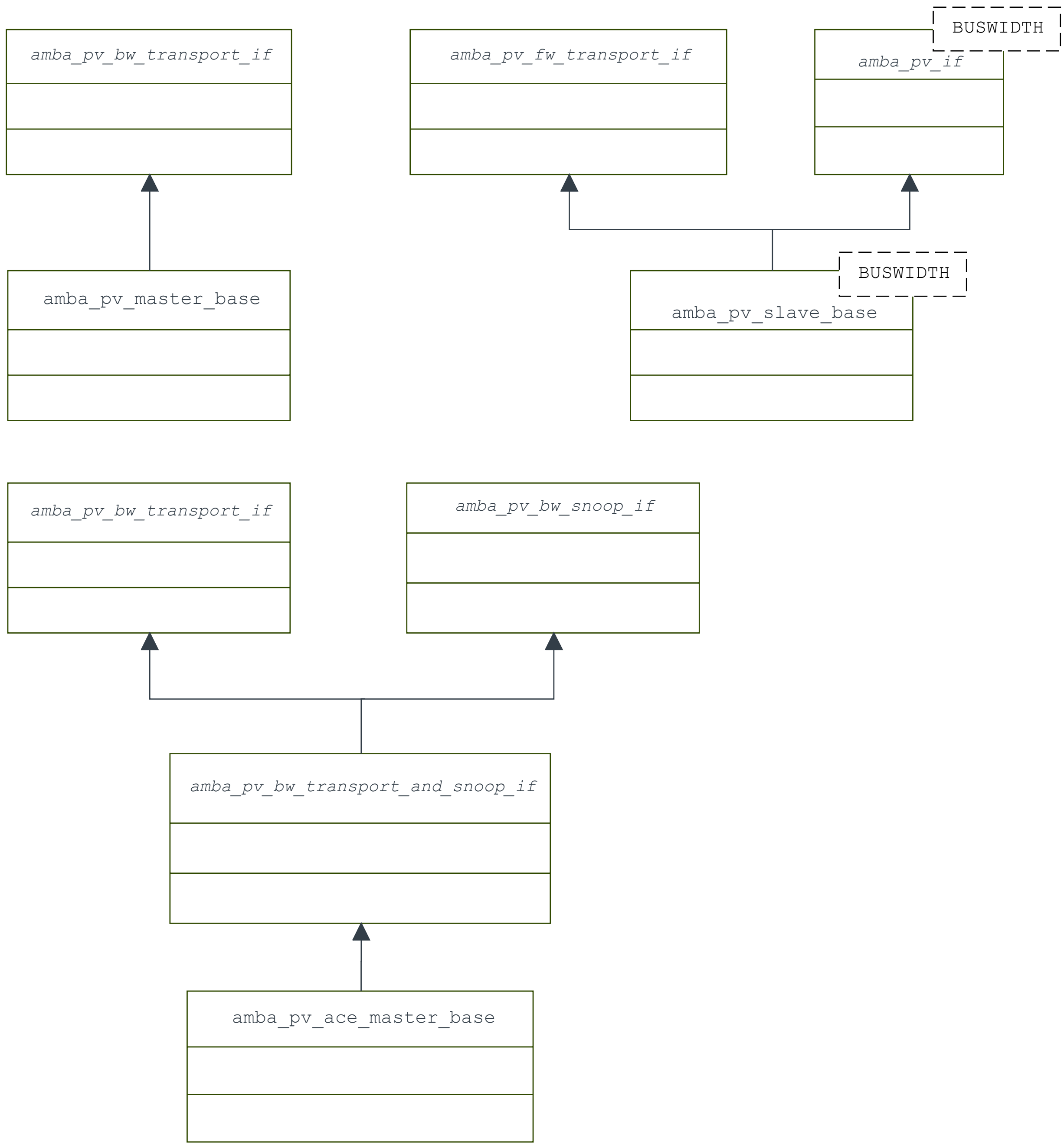
Tagged variant of `tlm_fw_transport_if`.

`amba_pv_bw_transport_and_snoop_if`

Tagged variant of `tlm_fw_transport_if` and `tlm_bw_transport_if`, which 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.

Figure 3-2: Core interfaces and user layer



Related information

User and transport layers on page 51

3.1.3 User layer

The user layer comprises an interface and base classes for modules.

amba_pv_if<>
: User-layer transaction interface providing `read()`, `write()`, `burst_read()`, `burst_write()`, `debug_read()`, `debug_write()`, `get_direct_mem_ptr()`, `atomic_store()`, `atomic_load()`, `atomic_swap()`, and `atomic_compare()` 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()`.

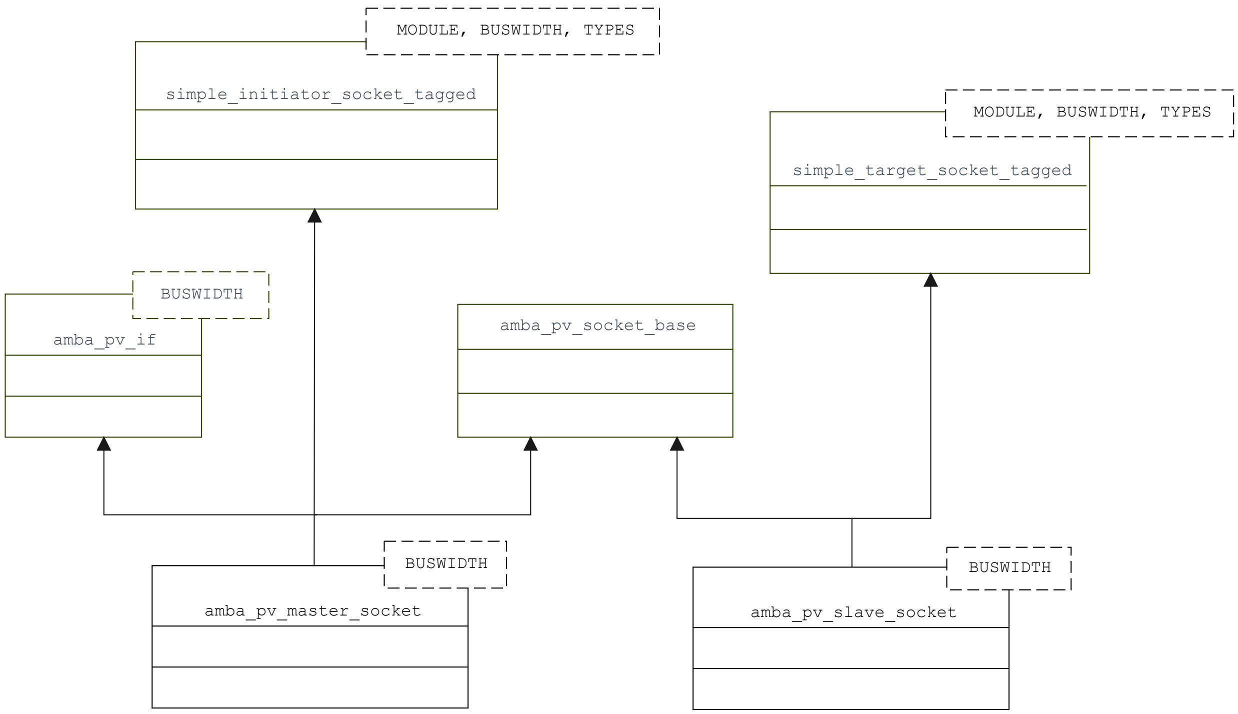
Related information

User and transport layers on page 51

3.1.4 Sockets

Both AMBA-PV socket classes provide socket identification/tagging. The master-socket class also implements the `amba_pv_if` user-layer interface.

Figure 3-3: Sockets



3.1.5 ACE sockets

These sockets have an extra socket as a private data member.

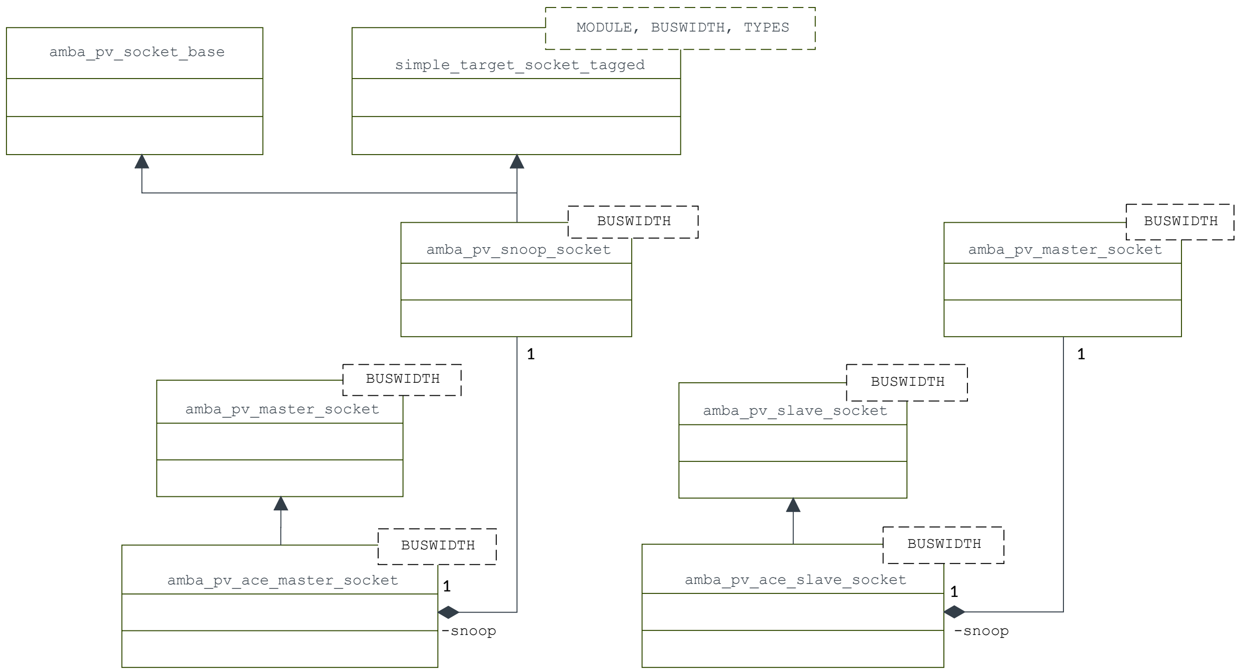
The `amba_pv_ace_master_socket<>` class provides:

- All the functions of `amba_pv_master_socket<>`.
- An `amba_pv_snoop_socket<>` as a private data member.

The `amba_pv_ace_slave_socket<>` class provides:

- All the functions of `amba_pv_slave_socket<>`.
- An `amba_pv_master_socket<>` as a private data member.

Figure 3-4: ACE sockets



3.1.6 Bridges

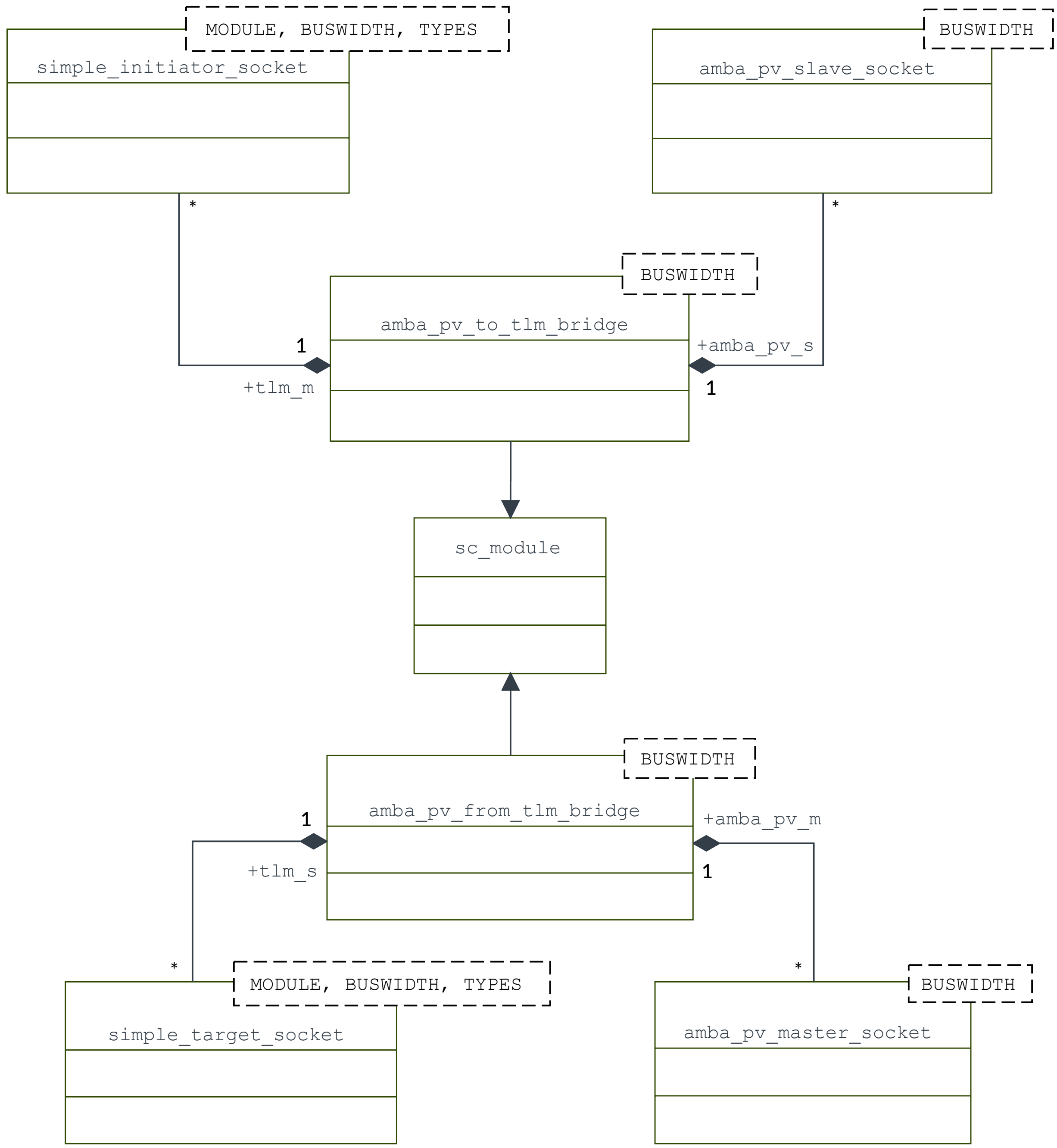
The `amba_pv_to_tlm_bridge<>` and `amba_pv_from_tlm_bridge<>` classes bridge between TLM 2.0 BP and AMBA-PV.

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 nonzero.

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

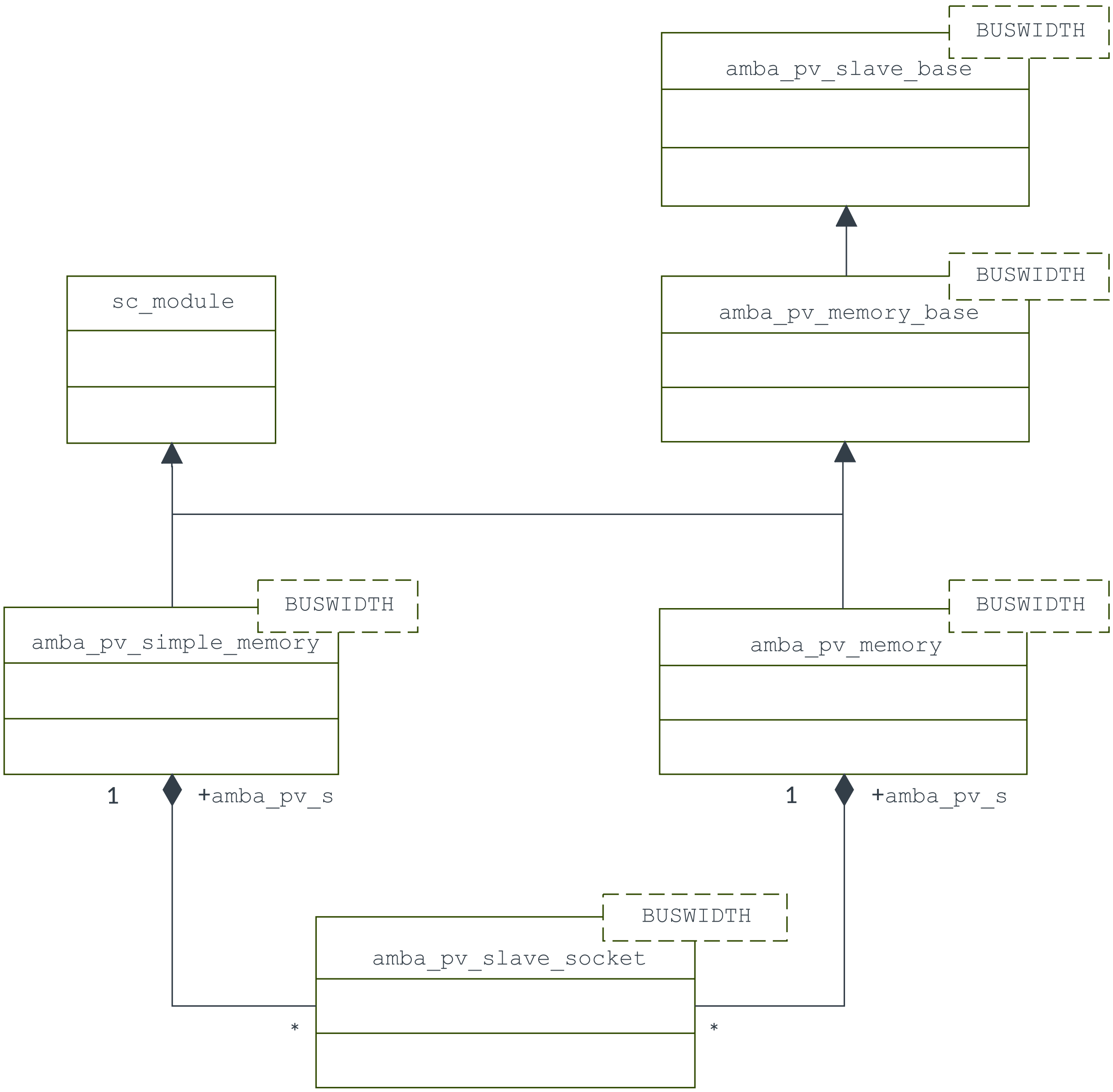
Figure 3-5: AMBA-PV to TLM bridges



3.1.7 Memory

Memories can be represented by either a simple model or an advanced model. The advanced model, class `amba_pv_memory<>`, supports optimized heap usage, save, and restore.

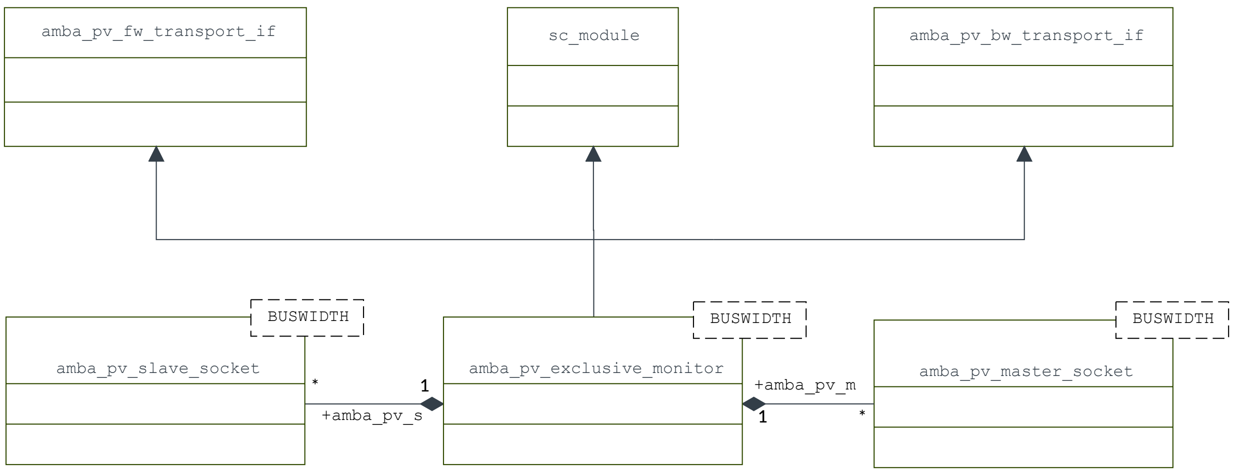
Figure 3-6: Memory



3.1.8 Exclusive monitor

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

Figure 3-7: Monitor



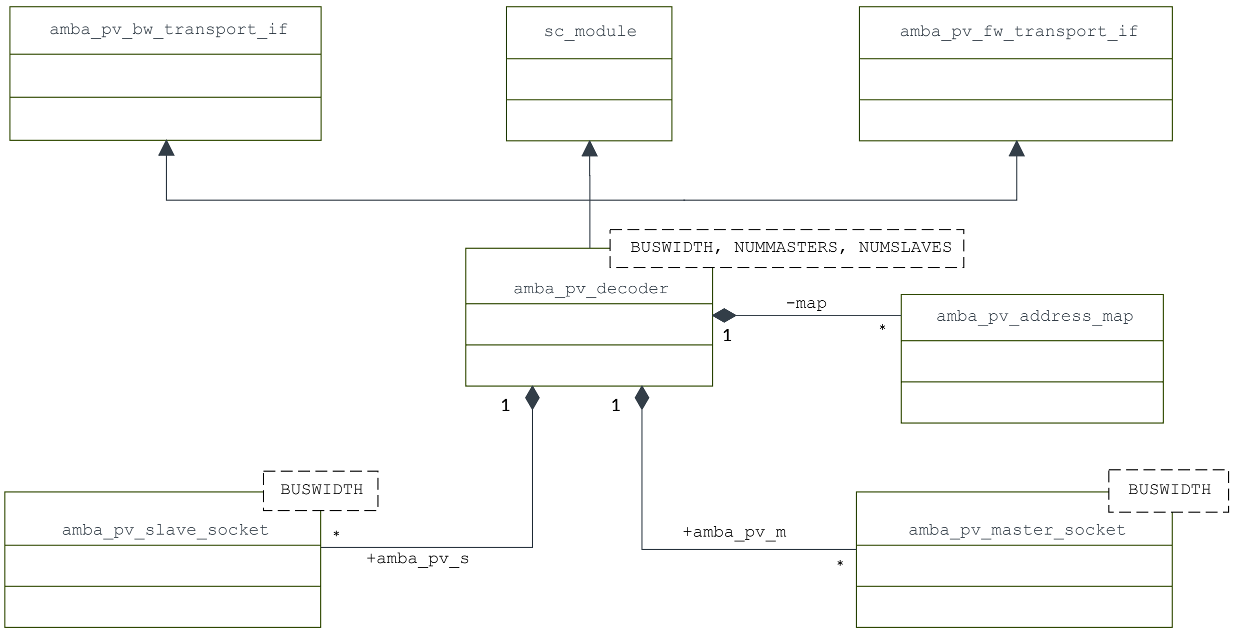
3.1.9 Bus decoder

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

Figure 3-8: Bus decoder



3.1.10 Protocol checker

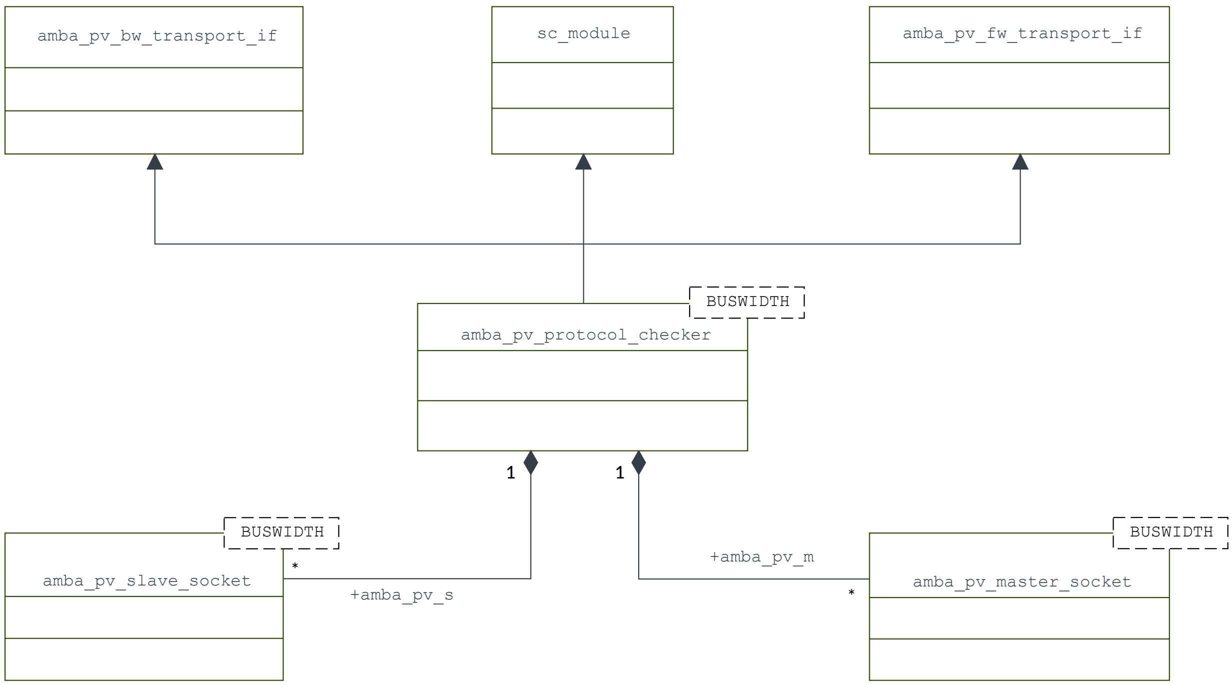
The `amba_pv_protocol_checker<>` class is used for confirming that a model complies with AMBA® bus protocols.

The transactions that pass through are checked against the AMBA® bus 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



Related information

AMBA-PV protocol checker on page 73

3.1.11 Signaling

The Signal API defines classes and interfaces for the modeling of side-band signals such as interrupts.

There are two variants:

- The Signal variant permits components to indicate a signal state change to other components and uses the `signal_` prefix.
- The SignalState variant 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.

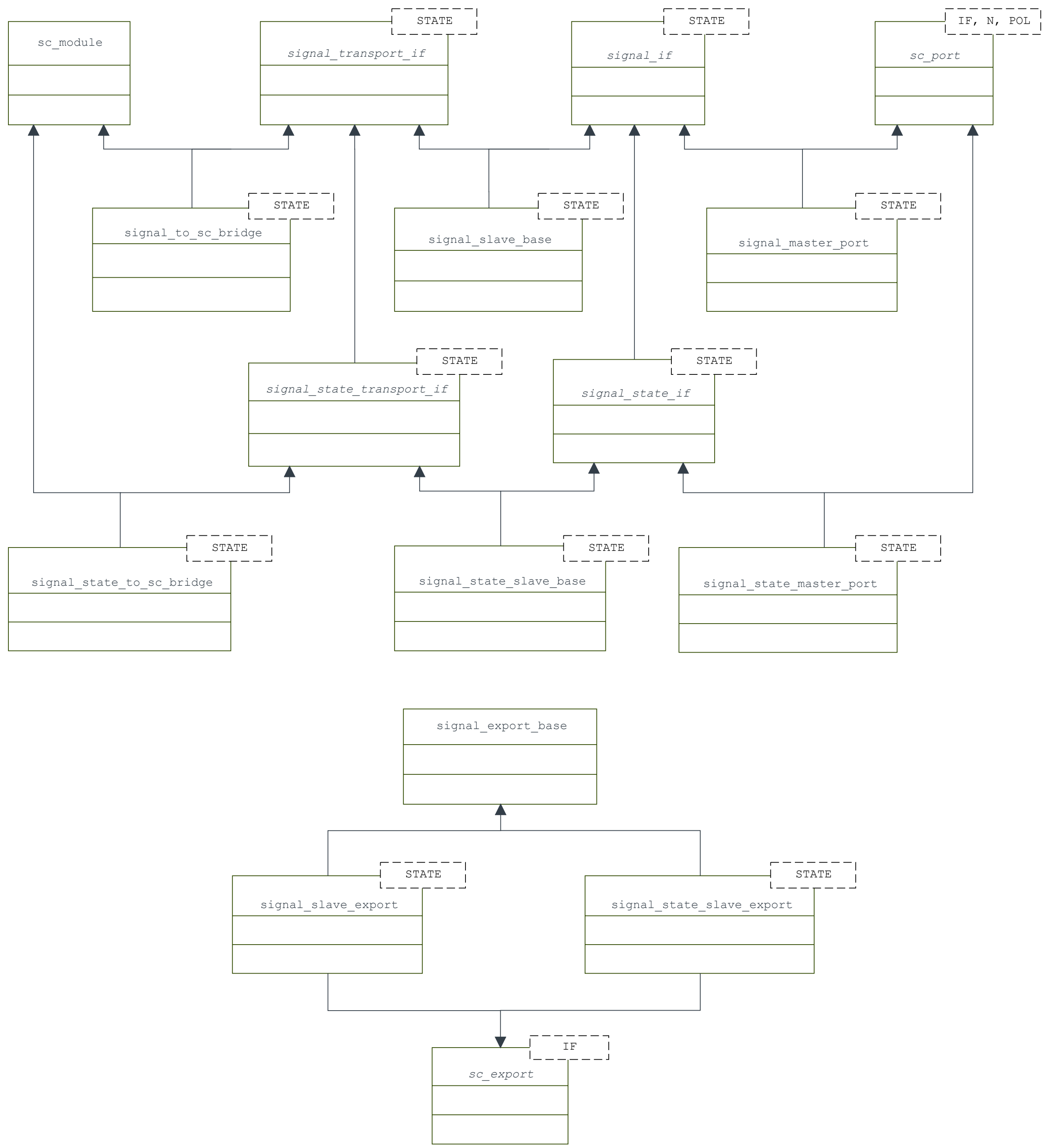
The Signal classes and interfaces feature a `STATE` template parameter.

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



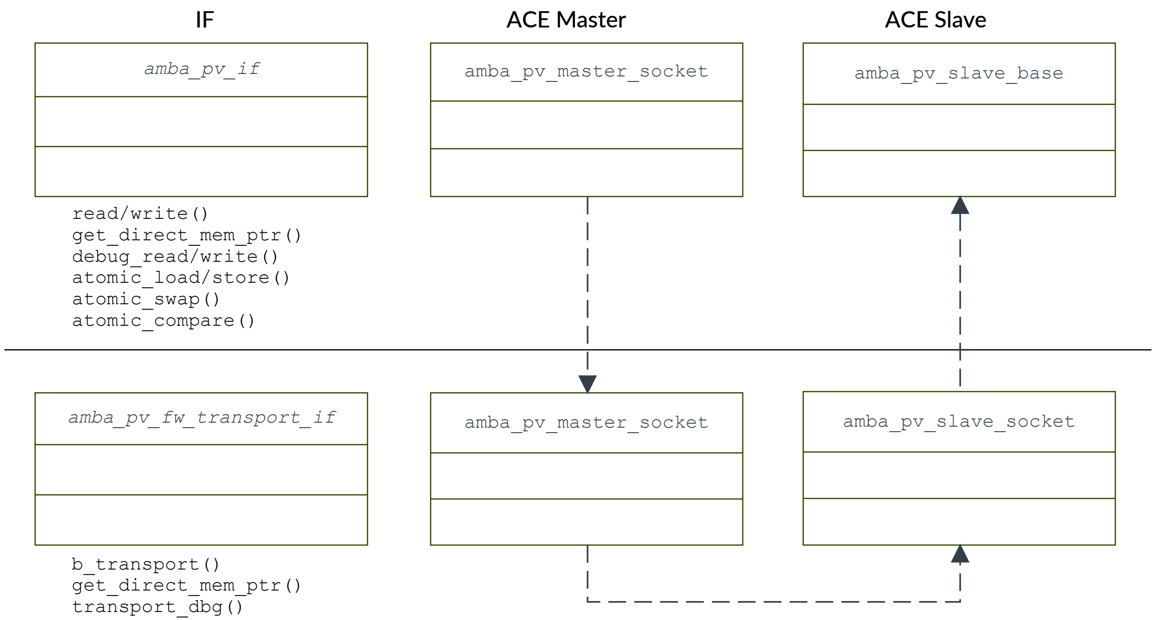
3.1.12 User and transport layers

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

3.1.12.1 Forward calls from master to slave

These calls go from the user layer through the transport layer and back to the user layer.

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()`
- `burst_read()`
- `write()`
- `burst_write()`
- `get_direct_mem_ptr()`
- `debug_read()`
- `debug_write()`
- `atomic_store()`
- `atomic_load()`

- `atomic_swap()`
- `atomic_compare()`

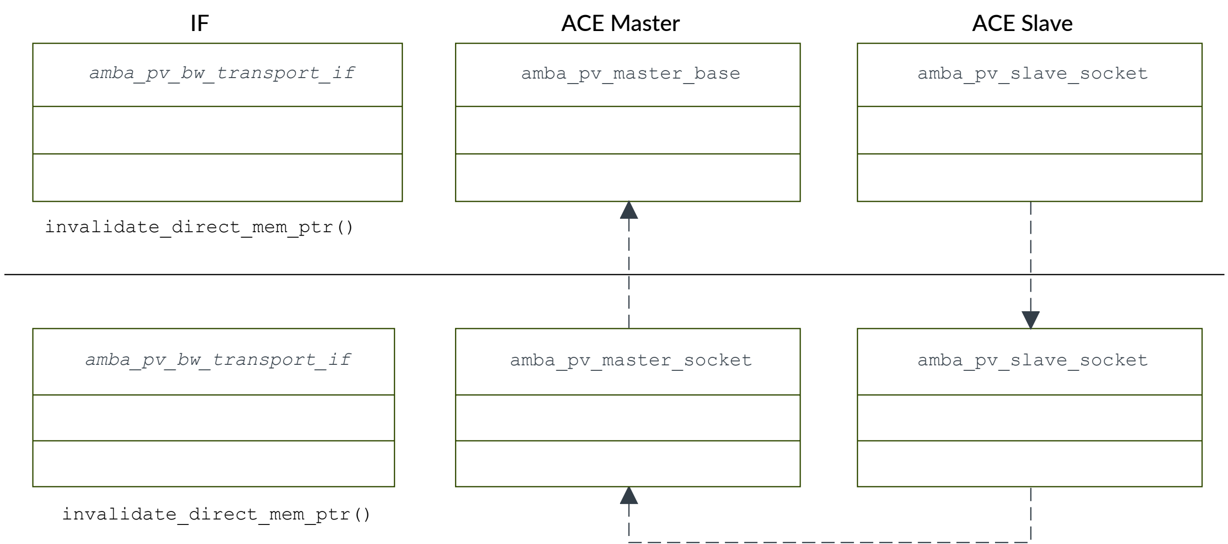
The `amba_pv_fw_transport_if` interface is an AMBA-PV core interface. Class `amba_pv_slave_base<>` also inherits from this interface. The interface defines the following member functions:

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

3.1.12.2 Backward calls from slave to master

These calls go from the user layer through the transport layer and back to the user layer.

Figure 3-12: Slave to master calls



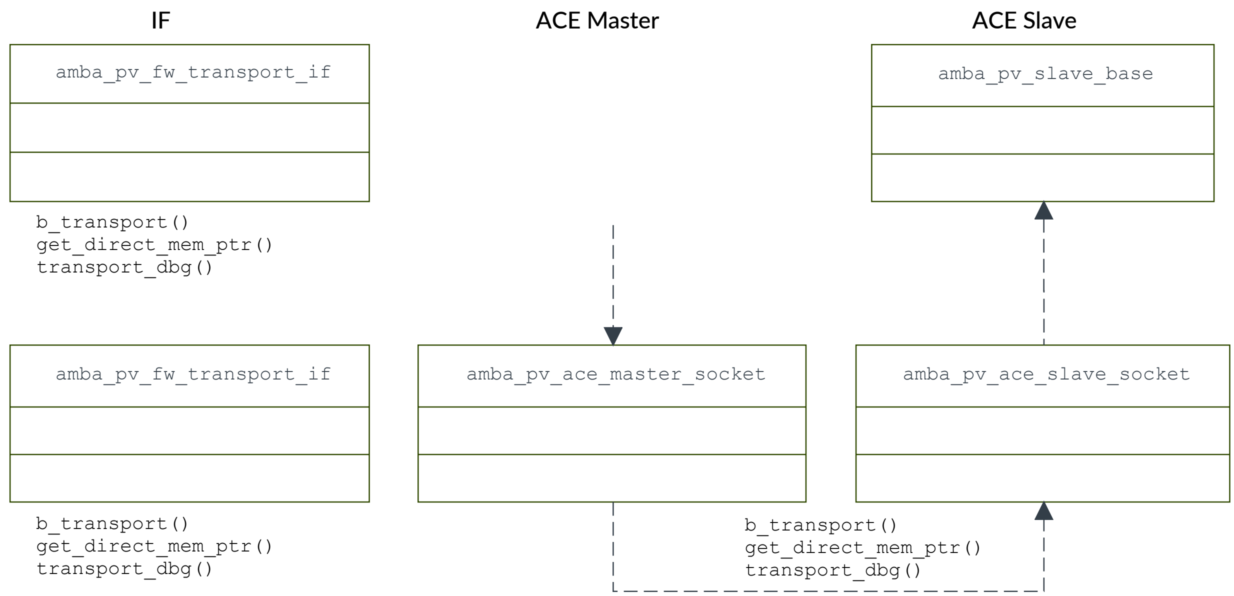
The `amba_pv_bw_transport_if` interface is an AMBA-PV core interface. It defines the `invalidate_direct_mem_ptr()` member function to invalidate pointers that were previously established for a DMI region in the slave and features tagging through its socket identification parameter.

3.1.12.3 Forward and backward calls with ACE sockets

This section describes how ACE sockets work.

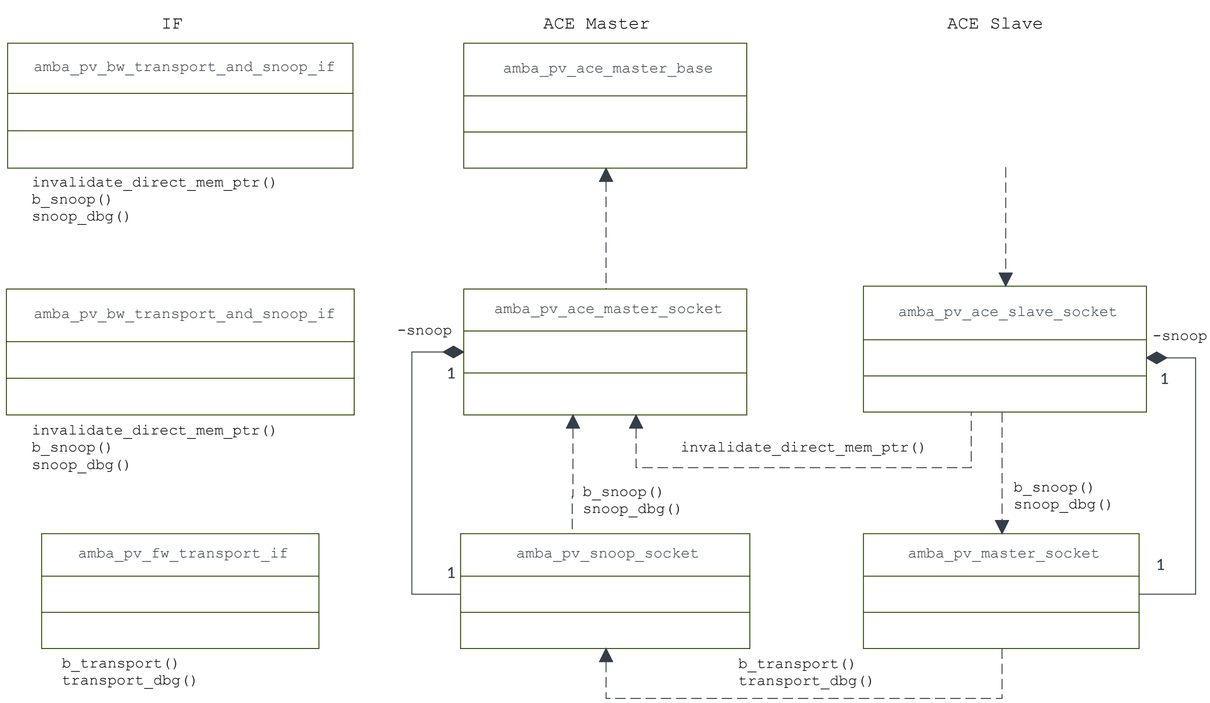
The forward calls from ACE masters to ACE slaves follow a similar flow to the flow for the non-ACE sockets.

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`. This is to maintain source level compatibility with previous versions of AMBA-PV.

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.

3.1.13 Transaction memory management

`amba_pv_trans_pool` and `amba_pv_trans_ptr` provide efficient memory management of AMBA-PV transactions, via a transactions pool and dedicated smart pointers.

The class manages extensions alongside the transactions: each transaction that returns from the pool has an extension that is associated with it.

3.2 Class summary

This section summarizes the AMBA-PV classes and interfaces.

3.2.1 List of classes and interfaces

This section lists the AMBA-PV classes and interfaces.

`amba_pv_ace_master_base`
The base class for AMBA-PV ACE master modules.

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

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

`amba_pv_atomic`
This class provides transaction information for atomic operations.

`amba_pv_attributes`
This class supports user-defined attributes.

amba_pv_bw_snoop_if
A tagged variant of the `tlm_bw_transport_if` interface, for AMBA-PV ACE master modules to implement.

amba_pv_bw_transport_and_snoop_if
A simple combination of the interfaces `amba_pv_bw_snoop_if` and `amba_pv_bw_transport_if`.

amba_pv_bw_transport_if
A tagged variant of the `tlm_bw_transport_if` interface, for AMBA-PV master modules to implement.

amba_pv_control
This class supports control information that is part of the AMBA® buses.

amba_pv_dvm
This class provides transaction information for DVM operations.

amba_pv_extension
This class is the AMBA-PV extension type.

amba_pv_fw_transport_if
This interface is a tagged variant of the `tlm_fw_transport_if` interface, for AMBA-PV slave modules to implement.

amba_pv_if<>
The user-layer transaction interface.

amba_pv_master_base
The base class for AMBA-PV master modules.

amba_pv_master_socket<>
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<>
The base class for AMBA-PV slave modules.

amba_pv_slave_socket<>
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.

amba_pv_trans_pool
This class implements the `tlm::tlm_mm_interface` and provides a memory pool from which to allocate and free transactions.

amba_pv_trans_ptr
This smart pointer retains sole ownership of a transaction through a pointer and releases that transaction when the `amba_pv_trans_ptr` goes out of scope.

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.

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.

3.2.2 Classes for virtual platforms

This section describes these classes and interfaces for modeling virtual platform components.

`amba_pv_ace_simple_probe<>`
: This simple probe with ACE support dumps the contents of transactions, including snoops.

`amba_pv_address_map`
: This class defines the address map structures.

`amba_pv_decoder<>`
: This class is the bus decoder model.

`amba_pv_exclusive_monitor<>`
: This class supports AMBA® 3 exclusive accesses.

`amba_pv_from_tlm_bridge<>`
: This class is the bridge module for interface between TLM 2.0 BP and AMBA-PV. It provides interoperability at subsystem boundaries. The component uses the TLM 2.0 extension mechanism.

`amba_pv_memory<>`
: This class is the advanced memory model that features optimized heap usage, save, and restore.

`amba_pv_memory_base<>`
: The base class for memory models.

`amba_pv_protocol_checker<>`
: The protocol checker that is used for conforming that a platform or model complies with the AMBA-PV protocol.

`amba_pv_simple_memory<>`
: The simple memory model.

amba_pv_simple_probe<>
: The simple probe component that dumps the contents of transactions.

amba_pv_to_tlm_bridge<>
: The bridge module for interface between TLM 2.0 BP and AMBA-PV. It provides interoperability at subsystem boundaries. The component uses the TLM 2.0 extension mechanism.

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

3.2.3 Classes for side-band signals

This section describes these classes and interfaces for modeling 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<>
: The Signal export base class.

signal_from_sc_bridge<>
: The generic bridge module from `sc_signal<>` to Signal.

signal_if<>
: The user-layer interface for Signal.

signal_master_port<>
: The port to instantiate on the Signal master side.

signal_request<>
: The Signal request type.

signal_response<>
: The Signal response type.

signal_slave_export<>
: The export to instantiate on the Signal slave side.

signal_slave_base<>
: The base class for Signal slave modules.

signal_state_if<>
: The user-layer interface for SignalState.

signal_state_nonblocking_transport_if<>
: The core non-blocking transport interface for SignalState.

signal_state_to_sc_bridge<>
: A generic bridge module from SignalState to `sc_signal<>`.

signal_state_from_sc_bridge<>
: The generic bridge module from `sc_signal<>` to SignalState.

`signal_state_master_port<>`
: The port to instantiate on the SignalState master side.

`signal_state_slave_base<>`
: The base class for SignalState slave modules.

`signal_state_slave_export<>`
: The export to instantiate on the SignalState slave side.

`signal_to_sc_bridge<>`
: The generic bridge module between Signal and `sc_signal<>`.

`signal_nonblocking_transport_if<>`
: The core non-blocking transport interface for the Signal.

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

4. Example systems

This chapter describes how to build and run the example systems, in `$MAXCORE_HOME/AMBA-PV/examples/`.

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`.

They use SystemC and TLM headers and libraries and require the `SYSTEMC_HOME` environment variable to be set to the SystemC installation directory. This variable is set when AMBA-PV is installed. To use a different copy of SystemC or TLM, modify the variable before building the examples.

SystemC and TLM headers and libraries are installed in `$MAXCORE_HOME/Accellera`, 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. The SystemC and TLM packages are link-compatible with the Accellera download version.

The AMBA-PV examples rely on a certain directory structure for libraries and header files. The structure of the Accellera packages is different because AMBA-PV supports a different range of compilers. To use the original Accellera packages with the AMBA-PV examples, apply a set of patch files to the Accellera package that adjusts the directory names. To rebuild the packages, follow the instructions from the `README.txt` file available in the `$MAXCORE_HOME/Accellera/source` directory.

On Linux hosts, running 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, Arm provides Microsoft Visual Studio project files (for example, `bridge_VC20XX.vcxproj`).

Related information

Accellera

4.2 Bridge example

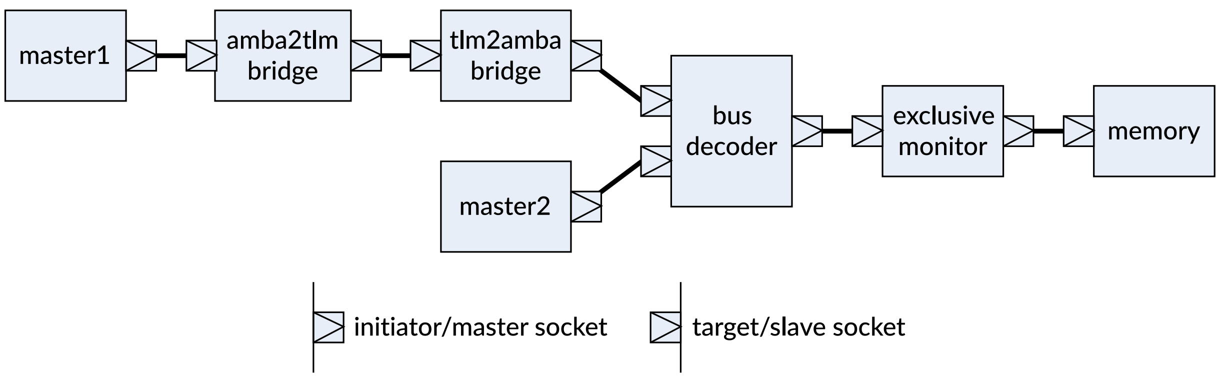
This example illustrates bridging to and from the TLM BP using the `amba_pv_to_tlm_bridge<>` and `amba_pv_from_tlm_bridge<>` classes.

It is based on the exclusive example, 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<>` to `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`.

Related information

Exclusive example on page 66

4.2.1 Building and running the bridge example

This section describes how to build and run this example.

About this task

To build the debug version:

- Under Linux, enter at the command prompt:

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

- Under Microsoft Windows, open `bridge_VC20XX.vcxproj` with Microsoft Visual Studio and build the `bridge` project, with the `Debug` configuration active.

To build the release version of this example:

- Under Linux, enter at the command prompt:

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

- Under Microsoft Windows, open `bridge_VC20XX.vcxproj` with Microsoft Visual Studio and build the `bridge` project, with the `Release` configuration active.

Note

Under Linux, the `make clean` command is optional.

To run this example, enter at the command prompt:

- Under Linux:

```
./bridge.x
```

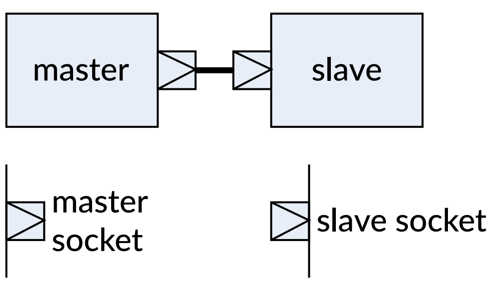
- Under Microsoft Windows:

```
bridge.exe
```

4.3 Debug example

This example 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 debug example

This section describes how to build and run this example.

About this task

To build the debug version:

- Under Linux:

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

- Under Microsoft Windows, open `dbg_VC20XX.vcxproj` with Microsoft Visual Studio and build the `dbg` project, with the `Debug` configuration active.

To build the release version:

- Under Linux, enter at the command prompt:

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

- Under Microsoft Windows, open `dbg_VC20XX.vcxproj` with Microsoft Visual Studio and build the `dbg` project, with the `Release` configuration active.

Note

Under Linux, the `make clean` command is optional.

To run this example, enter at the command prompt:

- Under Linux:

```
./dbg.x
```

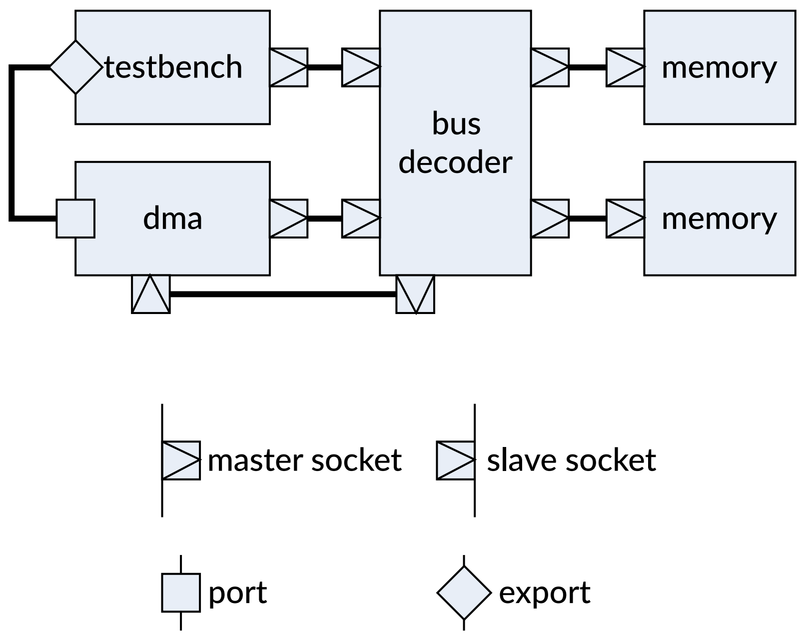
- Under Microsoft Windows:

```
dbg.exe
```

4.4 DMA example

This example 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 two memories. Additionally, it illustrates the use of DMI for simulation performance 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 DMA example

This section describes how to build and run this example.

About this task

To build the debug version:

- Under Linux, enter at the command prompt:

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

- Under Microsoft Windows, open `dma_VC20XX.vcxproj` with Microsoft Visual Studio and build the `dma` project, with the `Debug` configuration active.

To build the release version of this example:

- Under Linux, enter at the command prompt:

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

- Under Microsoft Windows, open `dma_VC20XX.vcxproj` with Microsoft Visual Studio and build the `dma` project, with the `Release` configuration active.

Note

Under Linux, the `make clean` command is optional.

To run this example, enter at the command prompt:

- Under Linux:

```
./dma.x
```

- Under Microsoft Windows:

```
dma.exe
```

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

- 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:

```
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 at the command prompt:

- Under Linux:

```
--dmi 400000
```

- Under Microsoft Windows:

```
--dmi 400000
```

Simulation statistics are displayed:

```
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
------------------------------------------------------------------------
```

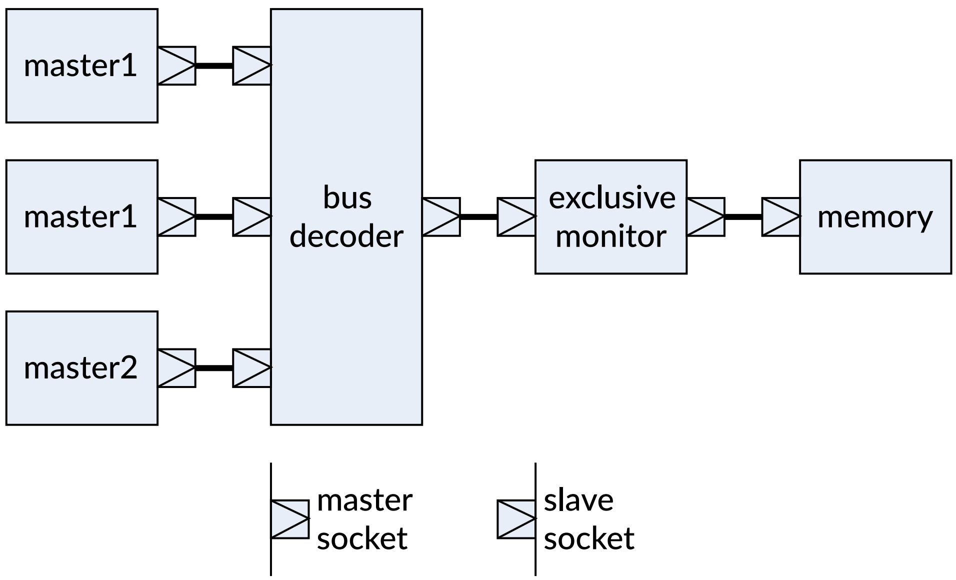
Note

These figures are examples. They do not constitute any reference in terms of timing. They vary with the host configuration.

4.5 Exclusive example

This example 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 exclusive example

This section describes how to build and run this example.

About this task

To build the debug version:

- Under Linux, enter at the command prompt:

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

- Under Microsoft Windows, open `exclusive_VC20XX.vcxproj` with Microsoft Visual Studio and build the `exclusive` project, with the `Debug` configuration active.

To build the release version:

- Under Linux, enter at the command prompt:

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

- Under Microsoft Windows, open `exclusive_VC20XX.vcxproj` with Microsoft Visual Studio and build the `exclusive` project, with the `Release` configuration active.

To build the `PROBE` version:

- Under Linux, enter at the command prompt:

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

- Under Microsoft Windows, open `exclusive_VC20XX.vcxproj` with Microsoft Visual Studio and build the `exclusive` project, with the `Probe` configuration active.

Note

Under Linux, the `make clean` command is optional.

To run this example, enter at the command prompt:

- Under Linux:

```
./exclusive.x
```

- Under Microsoft Windows:

```
exclusive.exe
```

4.6 Atomic example

This example consists of an example initiator, a decoder, and some simple memory.

The example initiator has three main sections:

SendExampleRequest()

Sends an example atomic transaction and elaborates the basic requirements while setting the attributes.

SendRequestWithTwoDifferentWays()
: Demonstrates sending atomic transactions through the user layer (atomic_*) or the transport layer (b_transport).

SendAtomicCompareRequest()
: Demonstrates how to send data and how the data is returned with AtomicCompare transactions.

The example is located in `$MAXCORE_HOME/AMBA-PV/examples/atomic_example/`. See the readme.txt for instructions on building and running it.

5. Creating AMBA-PV compliant models

This chapter describes a set of guidelines for the creation of AMBA-PV-compliant models of masters, slaves, and interconnect components.

5.1 Creating an AMBA-PV master

This section describes how to create an AMBA-PV master.

Procedure

1. Derive the master class from class `amba_pv_master_base` (in addition to `sc_module`).
2. Instantiate one master socket of class `amba_pv_master_socket` for each connection to an AMBA® bus. Specify a distinct identifier for each socket.
3. Implement the method `invalidate_direct_mem_ptr()`.
   A master does not need to implement this method explicitly if it does not support DMI.
4. 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()`, `debug_write()`, `atomic_store()`, `atomic_load()`, `atomic_swap()`, or `atomic_compare()`.
5. On completion of the transaction, check the returned response value.

5.2 Creating an AMBA-PV slave

This section describes how to create an AMBA-PV slave.

Procedure

1. Derive the slave class from class `amba_pv_slave_base` (in addition to `sc_module`).
   A memory slave can derive from class `amba_pv_memory_base` instead.
2. Instantiate one slave socket of class `amba_pv_slave_socket` for each connection to an AMBA® bus. Specify a distinct identifier for each socket.
3. Implement the methods `read()`, `write()`, `get_direct_mem_ptr()`, `debug_read()`, `debug_write()`, `atomic_store()`, `atomic_load()`, `atomic_swap()`, and `atomic_compare()`.
   A slave does not need to implement any method other than `read()` and `write()` if it does not support DMI, debug transactions, or atomic transactions.
4. 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.
5. 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.

6. 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 Creating an AMBA-PV interconnect

This section describes how to create an AMBA-PV interconnect.

Procedure

1. Derive the interconnect class from classes `amba_pv_fw_transport_if` and `amba_pv_bw_transport_if` (in addition to `sc_module`).
2. 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.
   The interconnect can alternatively use the class `amba_pv_socket_array` for master and slave sockets.
3. 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.
   Each master/slave socket is identified by its `socket_id`, the first parameter of those methods.
4. Pass on incoming method calls as appropriate on both the forward and backward paths.
   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.
5. In the implementation of `b_transport()`, the only AMBA-PV extension attributes modifiable by an interconnect component are the ID and the response attributes.
6. 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.
7. 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.
8. 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.
   The interconnect can use the class `amba_pv_address_map` for representing the address map.
9. 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.
10. In the implementation of `invalidate_direct_mem_ptr()`, modify the address range arguments before passing the call along the backward path.

5.4 Creating an AMBA-PV ACE master

This section describes how to create an AMBA-PV ACE master.

Procedure

1. Derive the master class from class `amba_pv_ace_master_base` (in addition to `sc_module`).
2. 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.
3. Implement the method `invalidate_direct_mem_ptr()`.
   An ACE master does not need to implement this method explicitly if it does not support DMI.
4. Implement the methods `b_snoop()` and `snoop_dbg()`.
   An ACE master does not need to implement the method `snoop_dbg()` if it does not support debug transactions.
5. 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()`.
6. On completion of the transaction, check the returned response status.

5.5 Creating an AMBA-PV ACE slave

This section describes how to create an AMBA-PV ACE slave.

Procedure

1. Derive the slave class from class `amba_pv_slave_base` (in addition to `sc_module`).
2. 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.
3. Implement the methods `b_transport()`, `get_direct_mem_ptr()`, and `transport_dbg()`.
   A slave does not need to implement any other method than `b_transport()` if it does not support DMI or debug transactions.
4. 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.
5. 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.
6. 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.

6. AMBA-PV protocol checker

This chapter describes the AMBA-PV protocol checker and the checks it performs.

You can use the AMBA-PV protocol checker with any model that is designed to implement the AMBA-PV protocol. You can instantiate the protocol checker, class `amba_pv_protocol_checker`, between any pair of AMBA-PV master and slave sockets. You can instantiate the protocol checker, class `amba_pv_ace_protocol_checker`, between any pair of AMBA-PV ACE master and slave sockets.

The behavior of the model you test is checked against the protocol by a set of checks in the protocol checker. The transactions that pass through are checked against the AMBA-PV protocol. 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 AMBA protocol check selection: check_protocol()

The `check_protocol()` method configures the AMBA® protocol checks performed by the protocol checker.

Table 6-1: AMBA® protocol checks method

Name	Allowed values	Default value	Description of check
`check_protocol()`	`AMBA_PV_APB`, `AMBA_PV_AHB`, `AMBA_PV_AXI`, `AMBA_PV_AXI3`, `AMBA_PV_AXI4_LITE`, `AMBA_PV_AXI4`, `AMBA_PV_ACE_LITE`, `AMBA_PV_ACE`, `AMBA_PV_AXI5`	`AMBA_PV_AXI3`	Select the AMBA® protocol checks to perform. Note that `AMBA_PV_AXI` is the same as `AMBA_PV_AXI3`. Arm deprecates the use of `AMBA_PV_AXI`.

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, this 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, this warning is issued:

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

6.2 Recommended checks: recommend_on()

The `recommend_on()` method enables or disables the reporting of protocol recommendations by the protocol checker.

Table 6-2: Reporting of protocol recommendations method

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

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

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

6.3 Checks that the protocol checker performs

This section describes the checks that the protocol checker performs, and the areas of the specifications that they apply to.

6.3.1 About the protocols

The checker uses the following protocols:

- AMBA® APB Protocol Specification.
- AMBA® 3 AHB-Lite Protocol Specification.
- AMBA® AXI and ACE Protocol Specification.

6.3.2 Architecture checks

This section describes the architecture checks performed by the protocol checker.

Table 6-3: Architecture checks performed by the protocol checker

Bus types	Description of check	AMBA® APB Protocol Specification	AMBA® 3 AHB-Lite Protocol Specification	AMBA® 4 AXI and ACE Protocol Specification
APB	The data bus can be up to 32 bits wide.	Section 2.1 *AMBA® 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, AXI5, ACE-Lite	The data bus can be 32, 64, 128, 256, 512, or 1024 bits wide.	-	-	Section A1.3.1 *Channel definition*

6.3.3 Extension checks

This section describes the extension checks performed by the protocol checker.

Table 6-4: Extension checks performed by the protocol checker

Bus types	Description of check	AMBA® APB Protocol Specification	AMBA® 3 AHB-Lite Protocol Specification	AMBA® 4 AXI and ACE Protocol Specification	AMBA® AXI 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 2.1 *AMBA® APB signals*	-	Section B1.1.1 *AXI4 signals not supported in AXI4-Lite*	-
AHB, AXI3, AXI4, AXI5, 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 2.1 *AMBA® 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, AXI5, ACE-Lite	A transaction of burst type WRAP must have a length of 2, 4, 8, or 16.	-	-	Section A3.4.1 *Burst length*	-

Bus types	Description of check	AMBA® APB Protocol Specification	AMBA® 3 AHB-Lite Protocol Specification	AMBA® 4 AXI and ACE Protocol Specification	AMBA® AXI Protocol Specification
AXI3	A transaction can have a burst length 1-16.	-	-	Section A3.4.1 *Burst length*	-
AXI4, AXI5, 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 2.1 *AMBA® APB signals*	Section 2.2 *Master signals*	Section A8 AXI4 *Additional Signalling*	-
APB, AHB, AXI3	Region values are not supported.	Section 2.1 *AMBA® APB signals*	Section 2.2 *Master signals*	Section A8 AXI4 *Additional Signalling*	-
AXI4, AXI5, ACE-Lite	Quality of Service values can be 0-15.	-	-	Section A8.1.1 *QoS interface signals*	-
AXI4, AXI5, ACE-Lite	Region values can be 0-15.	-	-	Section A8.2.1 *Additional interface signals*	-
AXI5	An AtomicCompare transaction of burst type WRAP can have a burst length of 1.	-	-	-	Section A7.4.3 *Atomic transactions attributes*

6.3.4 Address checks

This section describes the address checks performed by the protocol checker.

Table 6-5: Address checks performed by the protocol checker

Bus types	Description of check	AMBA® APB Protocol Specification	AMBA® 3 AHB-Lite Protocol Specification	AMBA® 4 AXI and ACE Protocol Specification	AMBA® AXI and ACE Protocol Specification
APB, AHB, AXI4-Lite	All transactions must have an aligned address.	Section 2.1 *AMBA® 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, AXI5, ACE-Lite	A burst cannot cross a 4KB boundary.	-	-	Section A3.4.1 *Burst length*	-
AXI3, AXI4, AXI5, ACE-Lite	A transaction with a burst type of WRAP must have an aligned address.	-	-	Section A3.4.1 *Burst length*	-

Bus types	Description of check	AMBA® APB Protocol Specification	AMBA® 3 AHB-Lite Protocol Specification	AMBA® 4 AXI and ACE Protocol Specification	AMBA® AXI and ACE Protocol Specification
AXI5	For an AtomicCompare with a burst type of WRAP, the address must be aligned to half of the total transaction size. For a burst type of INCR, it must be aligned to the total transaction size.	-	-	-	Section A7.4.3 *Atomic transactions attributes*

6.3.5 Data checks

This section describes the data checks performed by the protocol checker.

Table 6-6: Data checks performed by the protocol checker

Bus types	Description of check	AMBA® APB Protocol Specification	AMBA® 3 AHB-Lite Protocol Specification	AMBA® 4 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 2.1 *AMBA® APB signals*	Section 2.2 *Master signals*	Section B1.1.1 *Signal list*
AXI3, AXI4, AXI5, ACE-Lite	Read transactions must have a NULL byte enable pointer.	-	-	Section A2.6 *Read data channel signals*
AXI3, AXI4, AXI5, ACE-Lite	The byte enable length is a multiple of the transfer size for a write transaction.	-	-	-
AHB, AXI3, AXI4, AXI5, ACE-Lite	The streaming width is equal to the beat size for transactions with burst type FIXED.	-	-	-

6.3.6 Response checks

This section describes the response checks performed by the protocol checker.

Table 6-7: Response checks performed by the protocol checker

Bus types	Description of check	AMBA® APB Protocol Specification	AMBA® 3 AHB-Lite Protocol Specification	AMBA® 4 AXI and ACE Protocol Specification
APB, AXI4-Lite	A response array is not appropriate as all transactions are single transfers.	Section 2.1 *AMBA® 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, AXI5, ACE-Lite	An EXOKAY response can only be given to an exclusive transaction.	-	-	A3.4.4 *Read and write response structure*

6.3.7 Exclusive access checks

This section describes the exclusive access checks performed by the protocol checker.

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

Bus types	Description of check	AMBA® APB Protocol Specification	AMBA® 3 AHB-Lite Protocol Specification	AMBA® 4 AXI and ACE Protocol Specification
APB, AXI4-Lite	A transaction cannot be exclusive or locked.	Section 2.1 *AMBA® 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, AXI5, ACE-Lite	Locked accesses are not supported.	-	-	Section A7.3 *Locked accesses*
AXI3, AXI4, AXI5, ACE-Lite	The maximum number of bytes that can be transferred in an exclusive burst is 128.	-	-	Section A7.2.4 *Exclusive access restrictions*
AXI3, AXI4, AXI5, 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, AXI5	The burst length for an exclusive access must not exceed 16 transfers.	-	-	Section A7.2.4 *Exclusive access restrictions*
AXI3, AXI4, AXI5, 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, AXI5, 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, AXI5, 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*

6.3.8 Cacheability checks

This section describes the cacheability checks performed by the protocol checker.

Table 6-9: Cacheability checks performed by the protocol checker

Bus types	Description of check	AMBA® APB Protocol Specification	AMBA® 3 AHB-Lite Protocol Specification	AMBA® 4 AXI and ACE Protocol Specification
APB, AXI4-Lite	All transactions are non-cacheable, non-bufferable.	Section 2.1 *AMBA® APB signals* lists 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, AXI5, ACE-Lite	When a transaction is not modifiable then allocate attributes are not set.	-	-	Section A4.4 "Memory types".
APB, AHB, AXI3, AXI4, AXI5, 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".
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".

Bus types	Description of check	AMBA® APB Protocol Specification	AMBA® 3 AHB-Lite Protocol Specification	AMBA® 4 AXI and ACE Protocol Specification
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”.

6.3.9 Atomic checks

The protocol checker performs the following checks for atomic transfers. They are based on the AMBA® AXI and ACE Protocol Specification.

These checks are defined in `$MAXCORE_HOME/AMBA-PV/include/models/amba_pv_protocol_checker_base.h`:

- Atomic operation signal. `amba_pv_atomic_subop_t` can be any value for AtomicStore and AtomicLoad, but can only be `AMBA_PV_ATOMIC_ADD` (`0x0`) for AtomicSwap and AtomicCompare.
- Atomic endianness signal. `amba_pv_atomic_endianness_t` can be any value for AtomicStore and AtomicLoad, but can only be `AMBA_PV_LITTLE_ENDIAN` (`0x0`) for AtomicSwap and AtomicCompare.
- TLM command. The TLM command of atomic transactions is WRITE.
- Supported protocol.
- Snoop signal.
- Exclusive access.
- Burst size and length. AtomicCompare has a different requirement compared to other atomic transactions.
- Burst size and bus width.
- Address alignment and burst mode:
    - For AtomicStore, AtomicLoad, and AtomicSwap, the address must be aligned to the total transaction size (size * length), and the burst type must be INCR.
    - For AtomicCompare, if the address is aligned to the total transaction size, the burst type must be INCR. If the address is aligned to half the total transaction size (size * length / 2), the burst type must be WRAP.

Related information

AMBA AXI and ACE Protocol Specification

Proprietary Notice

Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in the Arm documents.

Product status

All products and Services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

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

Revision history

These sections can help you understand how the document has changed over time.

Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

Document history

Issue	Date	Confidentiality	Change
0200-14	16 September 2024	Non-Confidential	Update for v11.27.
0200-13	19 June 2024	Non-Confidential	Update for v11.26.
0200-12	13 March 2024	Non-Confidential	Update for v11.25.
0200-11	6 December 2023	Non-Confidential	Update for v11.24.
0200-10	7 December 2022	Non-Confidential	Update for v11.20.
0200-09	14 September 2022	Non-Confidential	Update for v11.19.

Issue	Date	Confidentiality	Change
0200-08	15 June 2022	Non-Confidential	Update for v11.18.
0200-07	16 February 2022	Non-Confidential	Update for v11.17.
0200-06	29 June 2021	Non-Confidential	Update for v11.15.
0200-05	22 September 2020	Non-Confidential	Update for v11.12.
0200-04	22 June 2018	Non-Confidential	Update for v11.4.
0200-03	23 February 2018	Non-Confidential	Update for v11.3.
0200-02	17 November 2017	Non-Confidential	Update for v11.2.
0200-01	31 August 2017	Non-Confidential	Update for v11.1.
0200-00	31 May 2017	Non-Confidential	Update for v11.0. Document numbering scheme has changed.
K	17 February 2017	Non-Confidential	Update for v10.3.
J	11 November 2016	Non-Confidential	Update for v10.2.
I	31 August 2016	Non-Confidential	Update for v10.1.
H	31 May 2016	Non-Confidential	Update for v10.0.
G	29 February 2016	Non-Confidential	Update for v9.6.
F	30 November 2015	Non-Confidential	Update for v9.5.
E	31 August 2015	Non-Confidential	Update for v9.4.
D	31 May 2015	Non-Confidential	Update for v9.3.

Issue	Date	Confidentiality	Change
C	28 February 2015	Non-Confidential	Update for v9.2.
B	30 November 2014	Non-Confidential	Update for v9.1.
A	31 May 2014	Non-Confidential	New document for Fast Models v9.0, from DUI0455H for v8.3.

For technical changes to this documentation, see the Fast Models Release Notes.

Conventions

The following subsections describe conventions used in Arm documents.

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 the Arm Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses 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	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

Caution

We recommend the following. If you do not follow these recommendations your system might not work.

Warning

Your system requires the following. If you do not follow these requirements your system will not work.

Danger

You are at risk of causing permanent damage to your system or your equipment, or of harming yourself.

Note

This information is important and needs your attention.

Tip

This information might help you perform a task in an easier, better, or faster way.

Remember

This information reminds you of something important relating to the current content.

Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Table 1: Arm publications

Document name	Document ID	Licensee only
AMBA, https://developer.arm.com/Architectures/AMBA	-	No
AMBA-PV Extensions to TLM 2.0 Reference Manual	DUI 0847	No

Table 2: Arm publications

Document name	Document ID	Licensee only
AMBA® AHB Protocol Specification	IHI 0033	No
AMBA® APB Protocol Specification	IHI 0024	No
AMBA® AXI and ACE Protocol Specification	IHI 0022	No

Table 3: Other publications

Document ID	Organization	Document name
-	http://www.accellera.org	*Accellera Systems Initiative*
IEEE 1666-2011	http://ieee.org	*IEEE 1666-2011, IEEE Standard for Standard SystemC Language Reference Manual*