ARM® RealView® ESL API v2.0

Developer's Guide

ARM®

ARM DUI 0359B

ARM RealView ESL API v2.0
Developer's Guide

Release Information

Change history

Description	Issue	Confidentiality	Change
January 2007	A	Non-confidential	New document based on ARM RealView SoC Designer 6.1 Developer's Guide and updated to reflect version 1.1 of the RealView ESL API.
June 2007	B	Non-confidential	Updated for ESL API version 2.0. Mx_ naming conventions replaced with CASI, CADI, and CAPI.

Proprietary Notice

Confidentiality Status

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

Product Status

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

Web Address

`http://www.arm.com`

Contents
ARM RealView ESL API v2.0 Developer's Guide

List of Tables
ARM RealView ESL API v2.0 Developer's Guide

List of Figures
ARM RealView ESL API v2.0 Developer's Guide

Preface

This preface introduces the *ARM RealView ESL API v2.0 Developer's Guide*. It contains the following sections:

About this document

This document describes the class hierarchy and programming interfaces for version 1.1 of the RealView ESL APIs. It is intended for users writing components that comply with the SystemC and ESL API system interfaces.

Intended audience

This document has been written for experienced hardware and software developers to design systems or components.

Users must, however, be familiar with the basic concepts of SystemC (such as `sc_module` and `sc_port`) and basic concepts of C++ (such as classes and inheritance).

Organization

This document is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for an introduction to designing components.

Chapter 2 ***The Cycle Accurate Simulation Interface***

This chapter describes the SystemC-based and ESL API simulation interfaces.

Chapter 3 ***The Cycle Accurate Debug Interface***

This chapter describes the debug interfaces that enables access to memory values, register values, and code disassembly for each simulation cycle.

Chapter 4 ***The Cycle Accurate Profiling Interface***

This chapter describes the profiling interfaces that enable collection of historical data about the memory, register, or port activity.

Chapter 5 ***The CASI Memory Map Interface***

This chapter describes the memory map interfaces that enable the memory maps of the components in a system to be configured.

Appendix A ***Static Scheduling of Communication Functions***

This appendix describes the static scheduling mechanism that is used to enable combinatorial support for communication functions.

Appendix B ***AMBA AHB TLM Specification for CASI***

This appendix describes differences between the AHB transaction interface and the generic CASI TLM.

Appendix C ***AMBA AXI TLM Specification for CASI***

This appendix describes differences between the AXI transaction interface and the generic CASI TLM.

Typographical conventions

The following typographical conventions are used in this book:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes processor signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
`monospace italic`	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.

Terminology

The following table lists the SystemC terms that are replaced in this document by their corresponding ESL API terms:

SystemC term	ESL API term	Description
module	component	The models for individual devices, for example, CPU core, memory, bus interface, and I/O.
port	master port	This is a port that generates transactions or signals.
channel	slave port	This responds to transactions or signals generated by a master port. SystemC channels are also know as sc_export.

Further reading

This section lists related publications by ARM and other companies.

ARM publications

The following publications provide reference information about the ARM architecture:

- *AMBA™ Specification* (ARM IHI 0011)
- *ARM Architecture Reference Manual* (ARM DDI 0100).

The following publications provide information about related ARM products:

- *RealView® SoC Designer Developer's Guide* (ARM DUI 0315)
- *RealView® SoC Designer User Guide* (ARM DUI 0316)
- *RealView® SoC Designer SystemC Linking Guide* (DUI0360)
- *ARM RealView® Model Debugger User Guide* (DUI0314)

External publications

The following publications provide additional information on simulation:

- *IEEE 1666™ SystemC Language Reference Manual*, (IEEE Standards Association)
- *SPIRIT User Guide*, Revision 1.2, SPIRIT Consortium.

Feedback

ARM welcomes feedback both on the ESL API and on the documentation.

Feedback on this document

If you have any comments about this document, send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter describes the main features of SystemC simulation with the classes defined by the *ESL Cycle Accurate Simulation Interfaces* (CASI), the related *Cycle Accurate Debug Interface* (CADI), and *Cycle Accurate Profiling Interface* (CAPI). It contains the following sections:

1.1 Overview

The RealView ESL API is a SystemC simulation interface for easy modeling and fast simulation of integrated systems-on-chip with multiple cores, peripherals, and memories. The ESL API consists of the following parts:

- The *Cycle Accurate Simulation Interface* (CASI)
- The *Cycle Accurate Debug Interface* (CADI)
- The *Cycle Accurate Profiling Interface* (CAPI)

CASI consists of a set of communication interfaces based on the SystemC language and supports both transaction level communication and cycle-based simulation modeling.

Note

The CASI scheduler uses a cycle-based approach where the cycle is the finest granularity scheduling element. On each cycle the `communicate()` and `update()` functions are called for every clocked model.

Developing all components using the cycle-based interfaces improves the performance of the system.

The ESL API is a SystemC interface and therefore supports event-driven simulation. This document, however, focuses on using the CASI library with the faster cycle-based simulation.

1.2 ESL API interface layers

The ESL API provides three layers of interfaces (as shown in Figure 1-1 on page 1-4) and enable:

CASI *Cycle Accurate Simulation Interface* based on SystemC communication library.

The main features of the CASI API are:

- cycle based scheduling for high simulation speed
- direct communication
- optimized transaction based communication
- port-based interconnection
- hierarchical system structures
- event-driven simulation support for full SystemC compatibility
- maximum simulation performance while enabling a high level of accuracy (up to the level of full pin-accuracy) on a cycle-by-cycle basis.

Note

The CASIMMI (*Cycle Accurate Simulation Interface - Memory Map Interface*) is used to define and use memory maps for bus master components. See Chapter 5 *The CASI Memory Map Interface* for details on using memory maps.

CADI *Cycle Accurate Debug Interface* enables reading and writing memory and register values and also provides the interface to external debuggers.

CAPI *Cycle Accurate Profiling Interface* enables collecting historical data from a component and displaying the results in various formats.

CADI and CAPI are additional interfaces not covered by SystemC. Implementing the CADI and CAPI interfaces is optional, but it is recommended for all user-defined components.

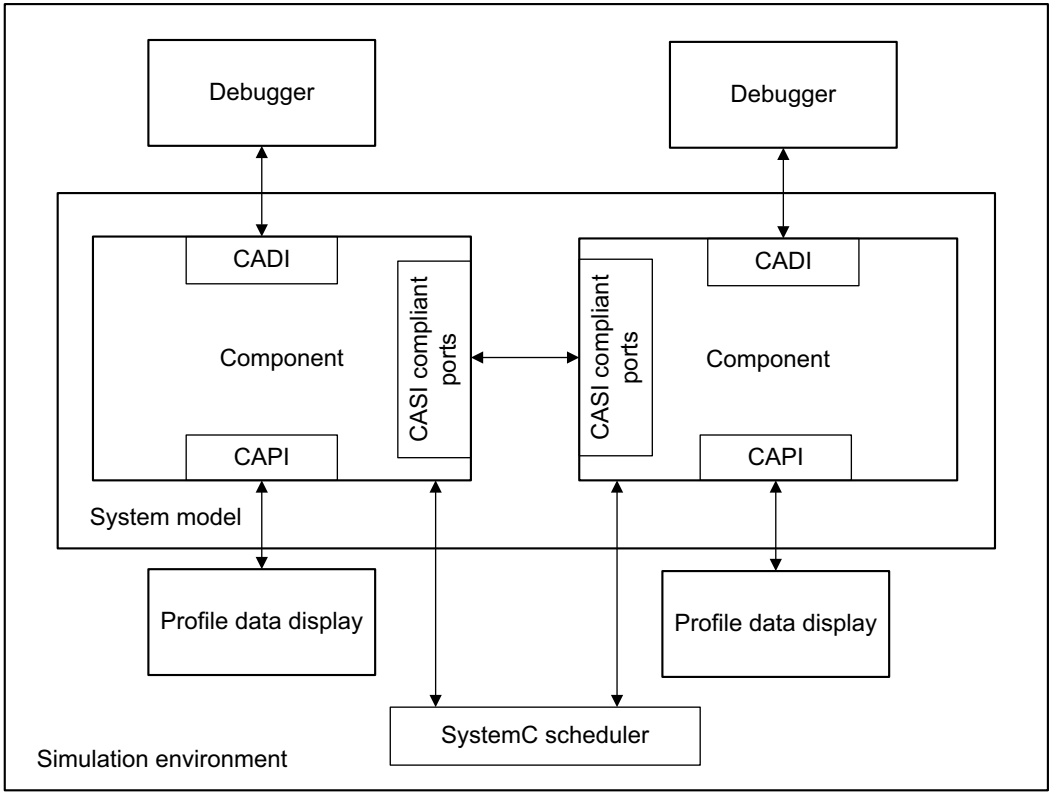

Figure 1-1 Interface layers

Note

The ESL API supports extending the interfaces while maintaining binary compatibility. See *CAInterface extensions* on page 1-41.

1.3 Theory of operation

A system consists of components and connections between components. A component might consist of sub-components connected together in a hierarchy.

A simulation of a CASI-compatible system is performed by:

- loading dynamic libraries of components (either standard library components or components that you have created yourself)
- creating instances of the components (for example by using factory function calls)
- connecting the components through standard port interfaces
- connecting the resulting model to a simulation controller that contains the timing and scheduler functionality
- the simulation scheduler clocks the cycle-based models in a lock-step fashion.

For every scheduler clock triggered by the simulation controller, the controller calls the clock interface function of each connected component. All clocked components are completely synchronized with each other.

The clock function of a component can call functions in other interfaces (such as memory interfaces or signal interfaces) to implement communications.

The cycle-based clock generators rely on the OSCI scheduler (which is itself event driven) to generate a cycle-based clock to drive all of the cycle-based components.

The internal behavior of a component is not constrained by the ESL API. It can be anything the designer wishes as long as the interface functions behave as expected by the other components. Correctly implemented system interfaces do not guarantee that the component behaves correctly in a system model. The system interfaces only standardize the way that components communicate with each other.

1.4 Components

The software duplicates the behavior of the hardware elements and provide ports that can be used for connection to other components.

Each component has properties that determine:

- which connections are allowed
- which type of object files are expected for loading
- whether debug front ends are supported.

Component parameters can be set either at system design time or at simulation time (depending on the type of parameter).

Note

Because parameters are set before the components are connected, components can have:

- configurable size for generic memories
- configurable behavior for peripherals.

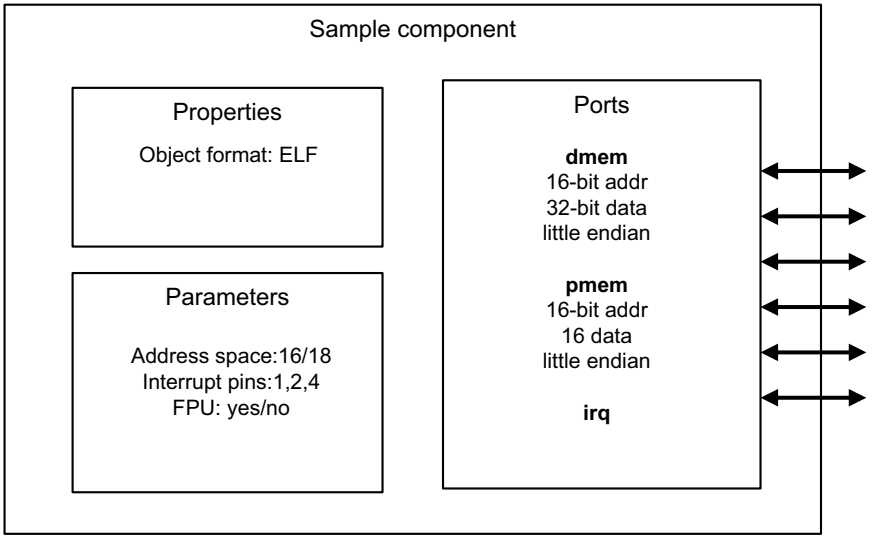

Figure 1-2 A sample component with properties, parameters, and ports

1.5 Connections

Components provide ports that enable them to be connected together.

Unlike many other system simulation solutions, the ESL API uses direct communication. One component accesses the shared resources of another component by directly calling a method provided by the owner of the shared resource.

CASI provides two different types of connection:

Signal based

The signal-based interface is very close to hardware simulators in that it simulates every signal independently.

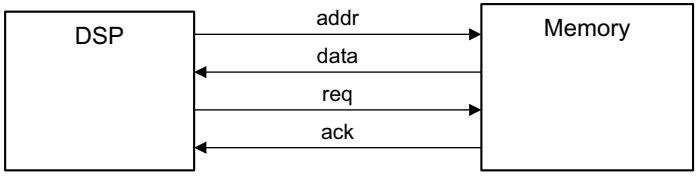

Figure 1-3 Signal-based communication

Transaction based

The transaction-based interface encapsulates a group of signals into one data structure that is manipulated read or write transactions.

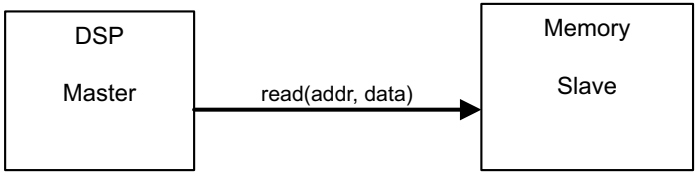

Figure 1-4 Transaction-based communication

Rather than distinguishing input and output ports, the CASI API distinguishes master and slave ports. A master port initiates the transfer and a slave-port responds to the transfer.

In Figure 1-3:

- The DSP component is the signal master for the `addr` and `req` signal ports.
- The Memory component is the signal slave for the `addr` and `req` signal ports.
- The DSP component is the signal slave for the `data` and `ack` signal ports.
- The Memory component is the signal master for the `data` and `ack` signal ports.
- Each signal has its own interface function that transmits the signal state.

In Figure 1-4 on page 1-7:

- The component DSP is the transaction master.
- The component Memory is the transaction slave.
- There is only one interface function, `read()`, that uses data structures to combine all signals into a parameter that can be passed with the single function call.

  The data is returned by `read()` in either:
  - the return value of the function
  - modified shared memory
  - modified memory that was indicated by a passed pointer.

1.5.1 Components with slave ports

A component with one or more slave ports enables other components to access its shared resources. This is, for example, used in memories that provide a transaction slave interface with `read()` and `write()` access methods.

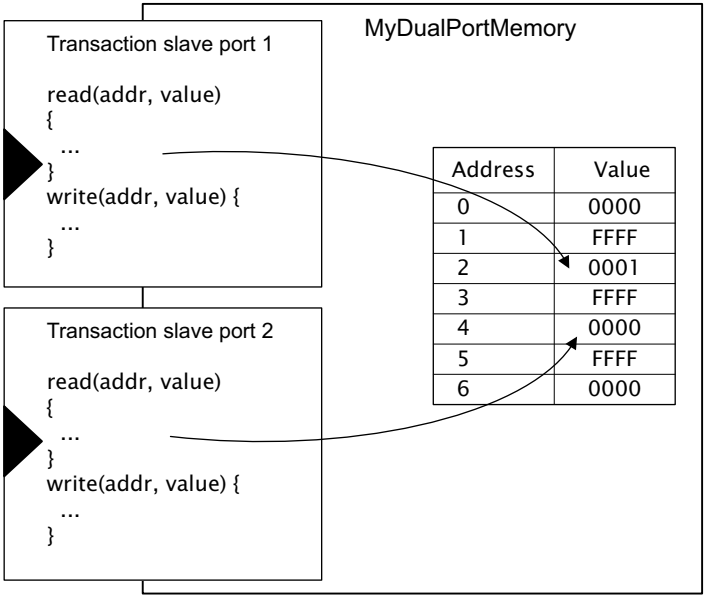

Figure 1-5 Dual ported memory component with two transaction slave ports

Figure 1-5 shows that the transaction slave ports provide access methods for the internal memory implementation of the component.

1.5.2 Components with master ports

Components with master ports can access connected slaves through well-defined access methods:

- the transaction master interface uses `read()` and `write()` methods
- the signal master interface uses `driveSignal()` and `readSignal()`.

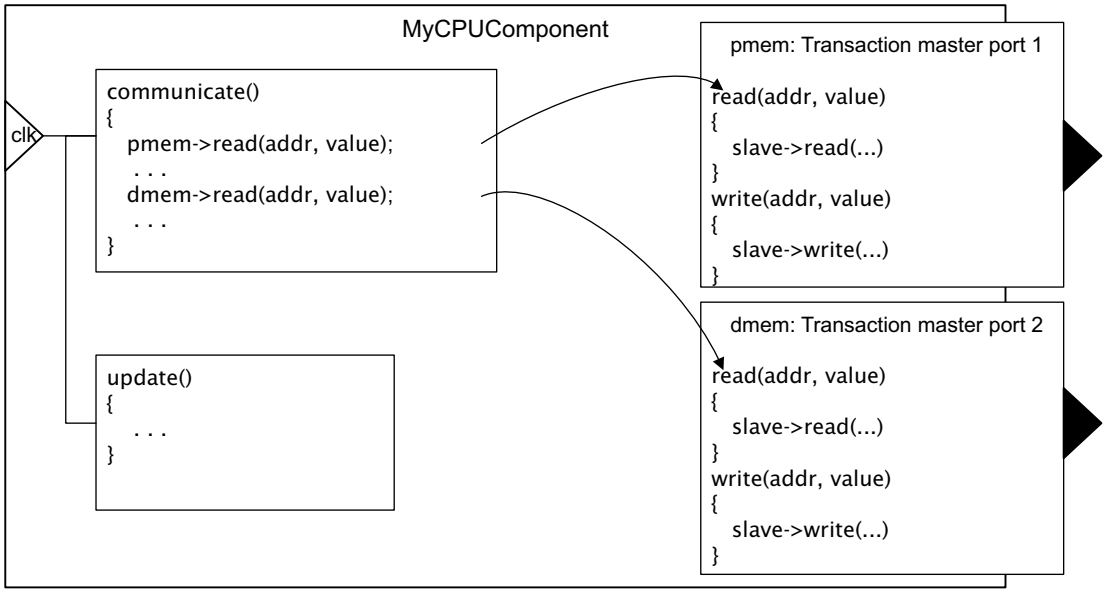

Figure 1-6 CPU component with separate transaction ports for data and program memory

Figure 1-6 shows a clocked CPU component with a Harvard architecture that has separate ports for data and program memory. In every clock cycle the component can read or write from memory by calling the appropriate access functions in the master ports. The master port then redirects these calls to the connected components.

1.5.3 Components with master and slave ports

Components can have an unlimited numbers of ports.

It is possible to pass through access calls from slave to master ports as shown in Figure 1-7. This might be required for the implementation of bus type components.

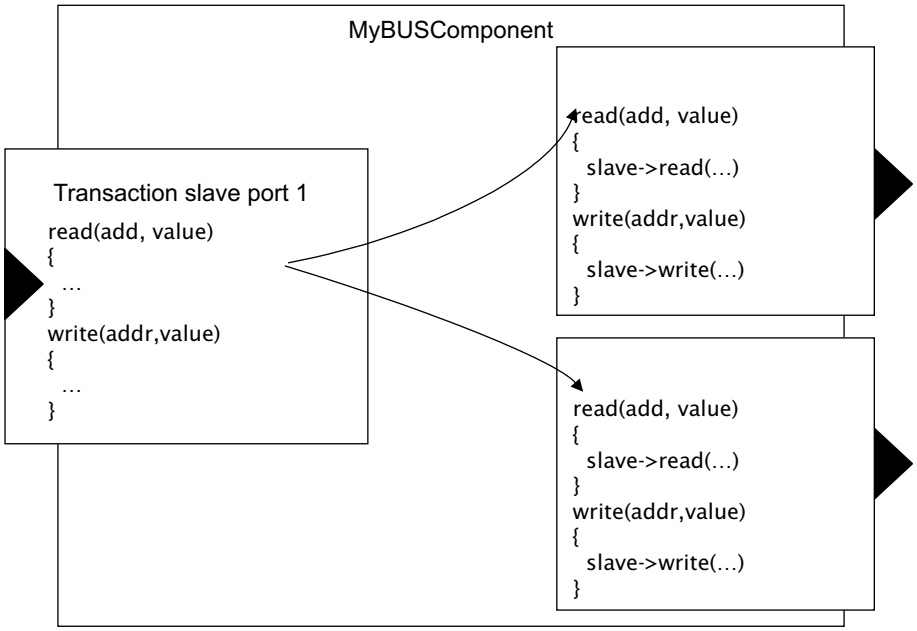

Figure 1-7 Bus component with a transaction slave and two transaction master ports

The bus component shown in Figure 1-7 redirects `read()` and `write()` calls to different master ports. Redirection might be used, for example, for sub-range decoding.

Figure 1-8 shows a system consisting of multiple components.

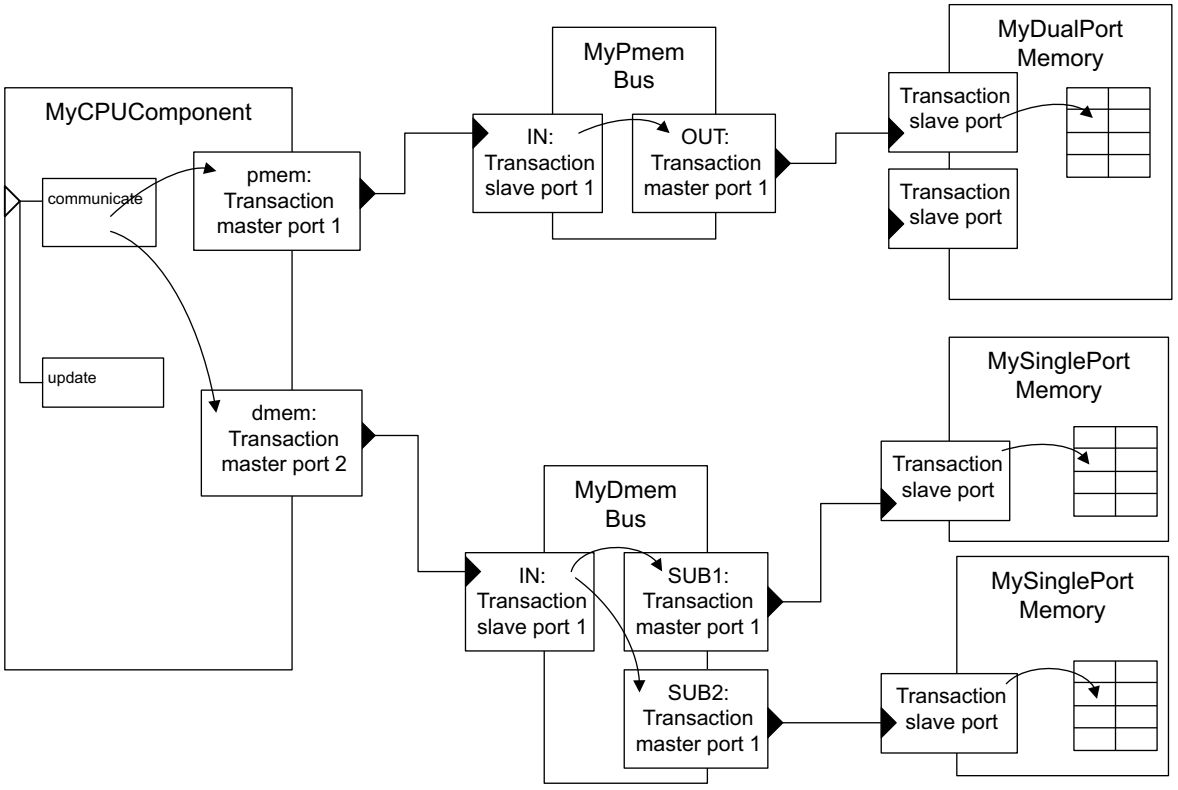

Figure 1-8 CPU component connected to memories through bus components

The ESL API interfaces enable the user to hide the implementation from the communication. When the CPU components call the read methods in its master ports, it does not know whether it is talking to a bus component or to a memory component directly. This simplifies reconfiguration and architecture exploration.

A single transaction master can drive multiple slaves as shown in Figure 1-9 on page 1-12. The master port, however, must include address decoding code to direct the transaction to the correct slave port.

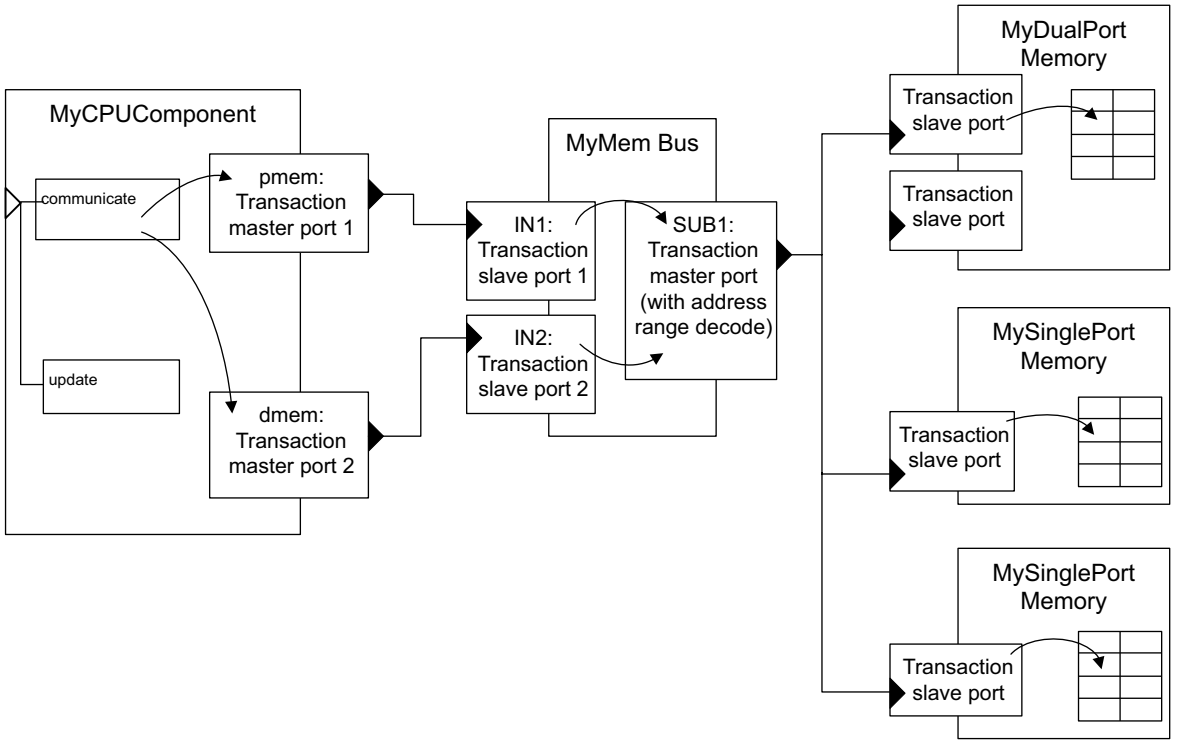

Figure 1-9 Single transaction master with multiple slave ports

1.5.4 Synchronous and asynchronous communication

The CASI signal interfaces always use synchronous communication that is accomplished by a single call to `driveSignal()`. The signal value is passed as a parameter. If a complex interconnection such as a bus is implemented with signals, there must be multiple distinct calls by the components to the `driveSignal()` functions of each individual signal.

The CASI transaction interface can perform a read or write in a single transaction call. Also, the CASI transaction interfaces are not restricted to single reads and writes. The interface supports protocols that perform burst or block reads of multiple memory locations through a single transaction. For a given communication protocol, AHB or AXI for example, the details are user-defined and must be documented as part of the protocol.

The communication schemes that can be used for a transaction model are user defined:

- synchronous
- asynchronous
- asynchronous with shared memory.

Synchronous transaction communication

The `read()` and `write()` functions enable synchronous access between different models by specifying the control information in the fields:

addr is the address for the read or write

value is the value read or to be written

control is control field for the read or write.

The `read()` and `write()` functions are expected to return in the same cycle that they were initiated. The return value indicates the status of the transaction. If they return `CASI_STATUS_OK`, the transaction has finished successfully.

The `read()` and `write()` functions can implement multi-cycled transactions. If in the first cycle they return, for example, `CASI_STATUS_WAIT`, then the initiating model calls the `read()` and `write()` function again in subsequent cycles, until it receives the `CASI_STATUS_OK` representing the end of this transaction.

The `readDbg()` and `writeDbg()` functions provide debug accesses and enable debuggers to read the desired information without advancing the simulation.

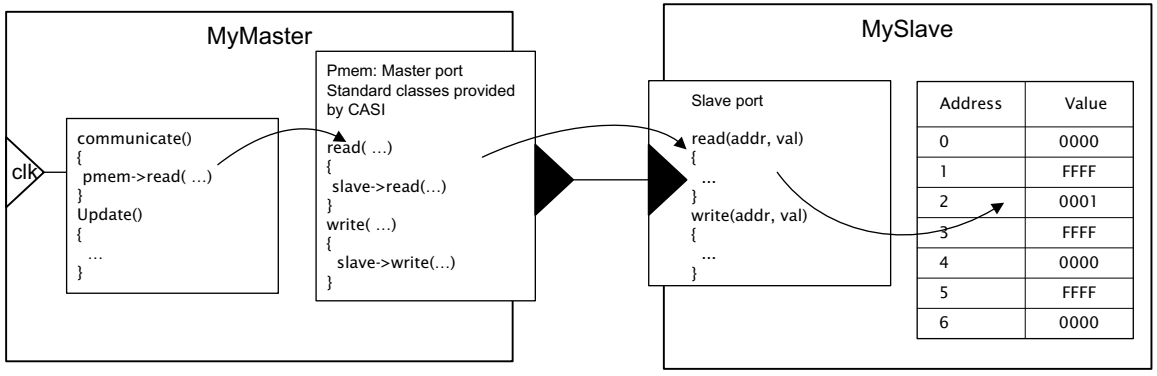

Figure 1-10 Synchronous communication example

Asynchronous transaction communication with callback

The asynchronous `readReq()` and `writeReq()` functions enable a communication model where the initiator master model provides a callback function pointer to the slave model. When the slave model is ready to serve the transaction, it calls the callback function and notifies the master that the data is ready.

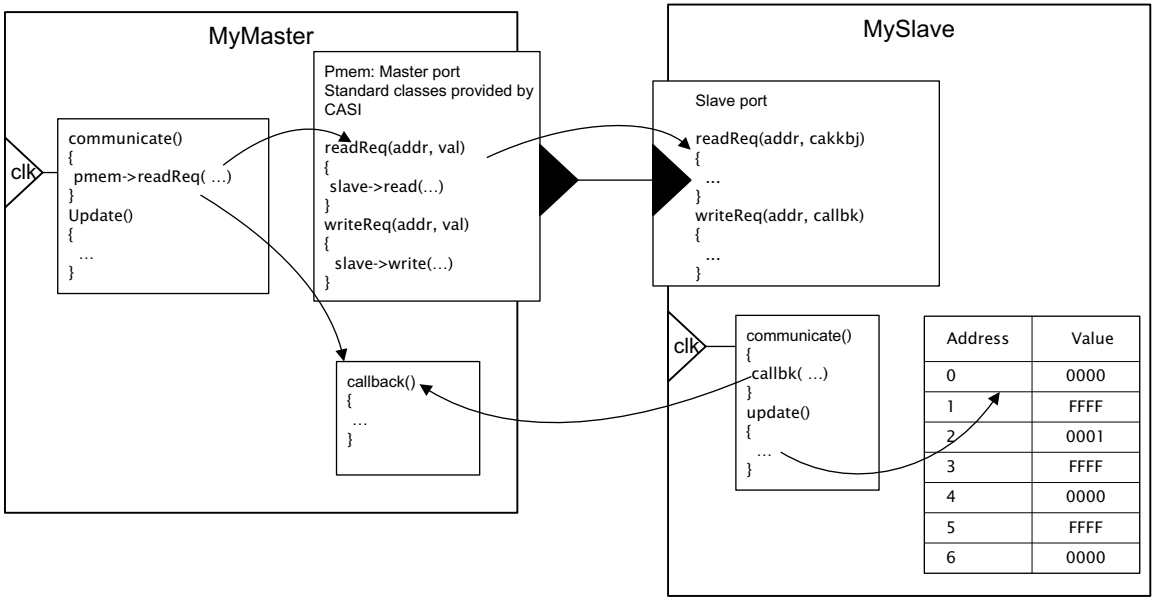

Figure 1-11 Asynchronous communication with callback example

Asynchronous transaction communication with shared memory

The shared-memory asynchronous functions provide a communication model where the initiating master model calls `driveTransaction()` providing to the slave a shared-memory data structure. This data structure is used throughout the life of the transaction to communicate the information between the master and the slave models. After the first `driveTransaction()` function call, no other function calls are required through the transaction, unless a `cancelTransaction()` is called to cancel the respective transaction. The shared data structure is stored in `CASITransactionInfo`.

An optional notification callback from a slave to the connected master can be implemented through a `CASINotifyHandlerIF` object. The `notifyEvent()` function can be called by the slave to inform the master that the contents of the transaction info data structure has changed. This enables the master to react to the changes in the same cycle.

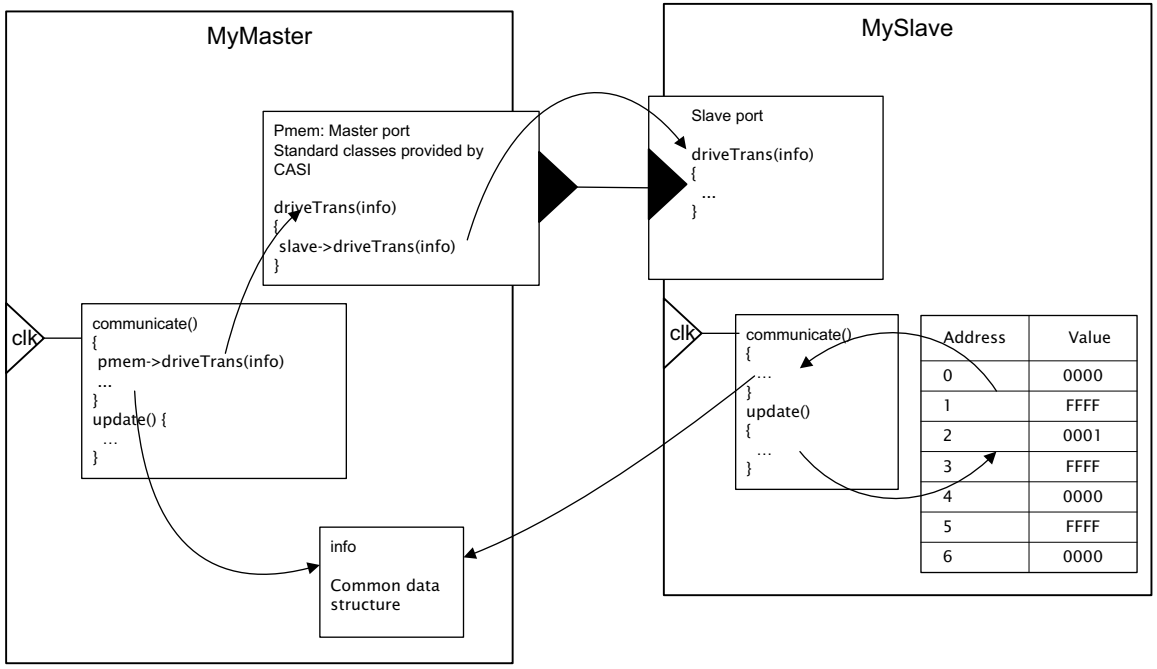

Figure 1-12 Asynchronous communication with shared memory example

The CASI implementation of the AHB protocol

CASI provides a general framework for model inter-communication, but the details of the communication protocol must be customized to match the actual protocol used. This is done by mapping the protocol fields to the CASI access functions parameters.

For instance, an AHB protocol implemented using the CASI synchronous model assigns AHB-related semantics to the parameters of the read/write functions of the casi_transaction_if interface. A example mapping is given in *Example AHB to CASI mapping* on page 1-16.

For details of the AHB protocol implementation for CASI, see the AHB CASI model documentation. Table 1-1 provides some guidance but must not be used as a reference for the AHB transactions description.

Table 1-1 Example AHB to CASI mapping

Parameter	Use	Example values
`Addr`	Address	`0x1A00000000000000` (64-bit unsigned)
`Value`	Data	`0xFFFF0001` (32-bit unsigned)
`ctrl[0]`	Transfer type	`BYTE`, `HWORD`, `DWORD`
`ctrl[1]`	Phase	`ADDR`, `DATA`
`ctrl[2]`	ACC	`HBURST`, `HSIZE`, `HTRANS`
`ctrl[3]`	Acknowledge	`DONE`, `WAIT`, `ABORT`

A typical communication sequence is listed in Table 1-2.

Table 1-2 Example AHB implementation

Cycle	Phase	Bus	Peripheral

0	ADDR	port->read(...)	`AHB_ch::read(addr,val,ctrl)`
			`…`
			`if(ctrl[1] == ADDR_PHASE)`
			`{`
			`  //decode read/write type (burst/…)`
			`  return OK;`
			`}`

1 to N	DATA	port->read(...)	`AHB_ch::read(addr,val,ctrl)`
			`…`
			`if(ctrl[1] == DATA_PHASE)`
			`  {`
			`  if(waitCount < DELAY)`
			`  {`
			`  ctrl[3] = WAIT;`
			`  }`
			`  else`
			`  {`
			`  val = getData(addr,ctrl);`
			`  ctrl[3] = DONE;`
			`  }`
			`  return OK;`
			`}`

1.6 Cycle based scheduling

Both cycle-based and event-driven simulations are supported:

- cycle-based components are executed on the edges of the clock
- event-driven components are executed based on their sensitivity lists and the events present in the system.

The execution of all cycle-based models happens in a synchronous cycle-based manner. In most cases, a simulation cycle is equivalent to a hardware clock cycle. Each simulation cycle is divided into communicate and update phases. Each clocked component is therefore called twice per cycle as shown in Figure 1-13.

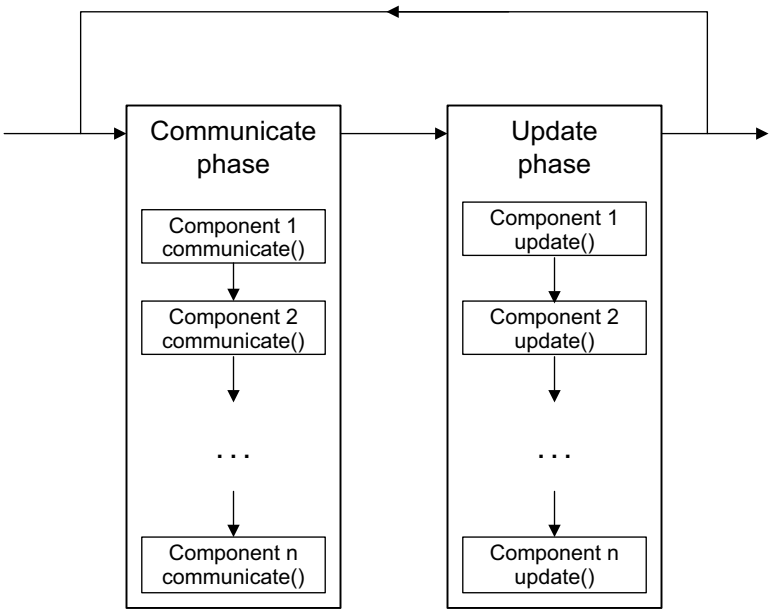

Figure 1-13 Communicate and update phases

Note

The order in which the component `communicate()` functions are called cannot be specified by the standard `registerClockSlave()` function. Some systems, however, require that communication functions are called in a specific order to, for example, control a bus request and acknowledge sequence within the same cycle. For more information on specifying the order of communication functions, see Appendix A *Static Scheduling of Communication Functions*.

During the communicate phase the components interact and during the update phase shared resources are modified:

Communicate phase

- perform all inter-component communication
- do not modify shared resources.

Update phase

- no communication between components
- update shared resources
- perform writes that were requested to internal components.

The simulation framework can enforce that the interface functions for communication are only called during the communicate stage. The simulation framework cannot, however, guarantee that the writes to shared resources are actually deferred to the update phase.

Note

The communicate and update model is typically used as described in this section and is the recommended way of implementing new components. There are, however, some special cases that do not follow the general model. The communicate phase is called only for the components that are clocked and register themselves to use communicate. It is possible, though not typical, for a component to register itself as a clocked component, but:

- only use update
- only use communicate
- ignore most phases and react, for example, to every fifth clock cycle.

1.6.1 Clocking the simulation

The CASI scheduler supports multiple clocks operating with different frequency. There exists a master clock that defines the fastest clocking granularity. All other clocks are created by dividing the master clock frequency with an integer number. If integer clocks are used, the frequency of the master clock must be the least common multiplier of the frequencies of the actual design clocks.

A single CASI model can have different parts clocked by different clocks. For example, a bus can receive data at one clock frequency and deliver it at a different clock frequency. Each differently clocked part must implement the `casi_clock_slave_port_base` interface and provide an appropriate implementation for the `communicate()` and `update()` member functions. At the implementation level, the

`casi_model_base` interface does not explicitly specify any communicate/update mechanism. Rather clocked models must create, or inherit from, one or more clock slaves that receive the necessary communicate/update calls.

As a general simulation strategy, the update calls must update the model state based on the most recently communicated data:

- For systems with single clocks, this strategy easily ensured by imposing a temporal ordering between the executions of communicate calls and update calls.
- If multiple clocks are present, ensure the ordering of communicate and update calls by executing the communicate calls at the beginning of a given cycle and the update calls in the last moment before the next cycle begins. This rule is applied for all clocks.

Clocking implementation

A master clock routine sits at the core of a CASI scheduler implementation and drives the simulation by executing a loop. In each iteration:

- for clocks running at full speed, a communicate phase is executed and that is followed by an update phase.
- For a clock with a frequency n times slower than the master clock:
  1. The communicate phase is executed during the master clock communication phase in a master clock iteration for which `(i mod n) = 0`.
  2. The update phase is executed during the master clock update phase in a master clock iteration for which `(i mod n) = (n - 1)`.

A 4 cycle window into the schedule is shown in Figure 1-14 on page 1-20 for a system has the following clocks:

- CLK1 is the master clock
- CLK2 is a clock with half the frequency of the master clock
- CLK4 is a clock with a quarter of the frequency of the master clock.

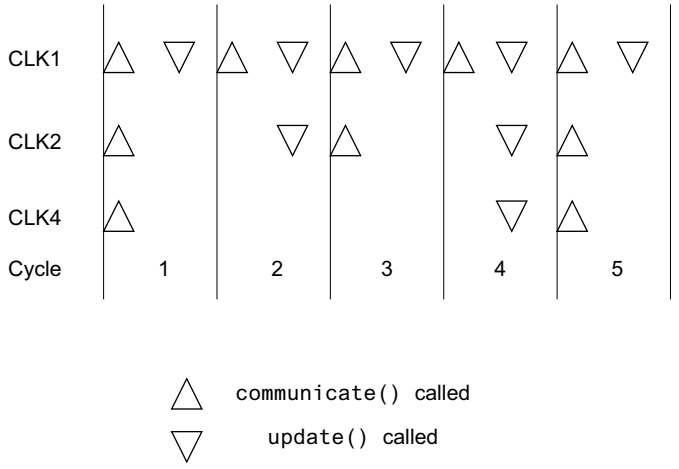

Figure 1-14 CASI schedule example for multiple clocks

If a CASI scheduler is operated together with a SystemC scheduler, it is important to preserve the communicate/update ordering with respect to SystemC events. Usually this means that the SystemC events relevant for one CASI simulation cycle must occur between the communicate and update phases of the CASI scheduler. The CASI main clock must be appropriately setup to ensure this requirement. This is typically done as shown in Figure 1-15 by:

1. the master clock communicate phase occurring on the positive clock edge of an equivalent SystemC clock
2. the master clock update phase occurring just before the positive edge of the next cycle.

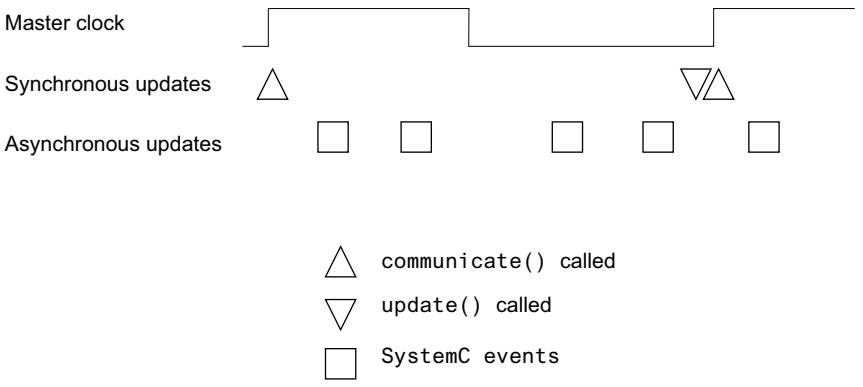

Figure 1-15 Mixed synchronous and asynchronous updates

1.7 Simulation stages

The simulation of a system is divided into multiple stages that are grouped as initialization, execution, and termination. Figure 1-16 shows the simulation stages and order of execution.

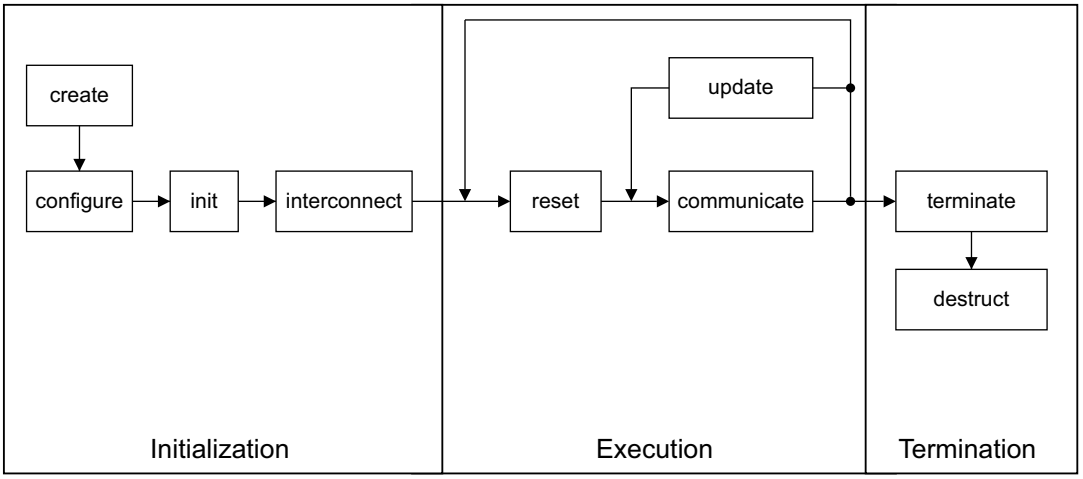

Figure 1-16 Stages of simulation

The simulation engine is the only instance that calls the functions that represent these stages. The components can request a reset or a terminate, but it is always the simulation engine that actually makes the call to the top-level component. The different stages are:

The create stage

This stage is used to load and instantiate the components. In the C++ implementation this functionality is contained in the constructor of the components:

- If a model causes a segmentation fault in its constructor, the simulation fails on startup.
- If a model allocates large amounts of memory in its constructor, this memory is allocated while creating the design as well. ARM recommends allocating any large amounts of data only in the `init()` stage of the simulation.

The configure stage

This stage is used to configure the components by setting parameters. This stage is implemented by the default behavior of the components and is normally not overridden by the individual models.

The init stage

This stage is used to initialize the model and allocate memory for its data members if necessary. Because the init stage is positioned after the configure stage, it is possible to initialize components conditionally, the size of a memory for example, based on a parameter.

ARM recommends that model initialization is done in this stage to simplify code in the model constructor. Whatever is allocated in `init()` must be deleted in `terminate()`.

The interconnect stage

In this stage the connections between the components are established. The connections are established once at initialization time only. During execution time, the interconnections cannot be modified. This stage is implemented by the default behavior of the components and is normally not overridden by the individual models.

The reset stage

The main purpose of the reset stage is to bring the components into a well-defined state before the actual simulation is started:

- During a soft reset, the models are expected to reset their state. This can be done repeatedly.
- A hard reset is only executed on startup.

  During a hard reset the components can, if applicable, load their object files. A core model would, for example, reload an `axf` image.

The communicate and update stages

These two stages represent the cycle-by-cycle behavior of the system. These stages are repeatedly executed until a reset or termination is requested by one of the components or by the user.

The terminate stage

This stage can be used for cleanup purposes. It is the counterpart to init. Terminate is called before the component instance is deleted. This stage is reached if:

- one of the components requests termination
- the specified simulation cycle count has been reached
- the simulation is terminated by an external signal.

The destructor stage

The destructor is called when the component is deleted after termination.

Note

An object that was created in the `init` stage must be deleted in the destructor instead of in the terminate stage.

The following virtual functions are defined in `CASIModule.h` for use by the individual system components to support the simulation phases:

- `CASIModule()`
- `configure()`
- `init()`
- `interconnect()`
- `reset()`
- `communicate()`
- `update()`
- `terminate()`
- `~CASImodule()`.

Each component must re-implement (overload) each of these functions (except for `interconnect()` where the base implementation might be sufficient).

1.8 Organizing source files for components and systems

A SystemC example typically has almost all of the code (except for the standard library include files) in a single source file. For ESL API systems, however, there are typically multiple source files where each file provides a specific functionality.

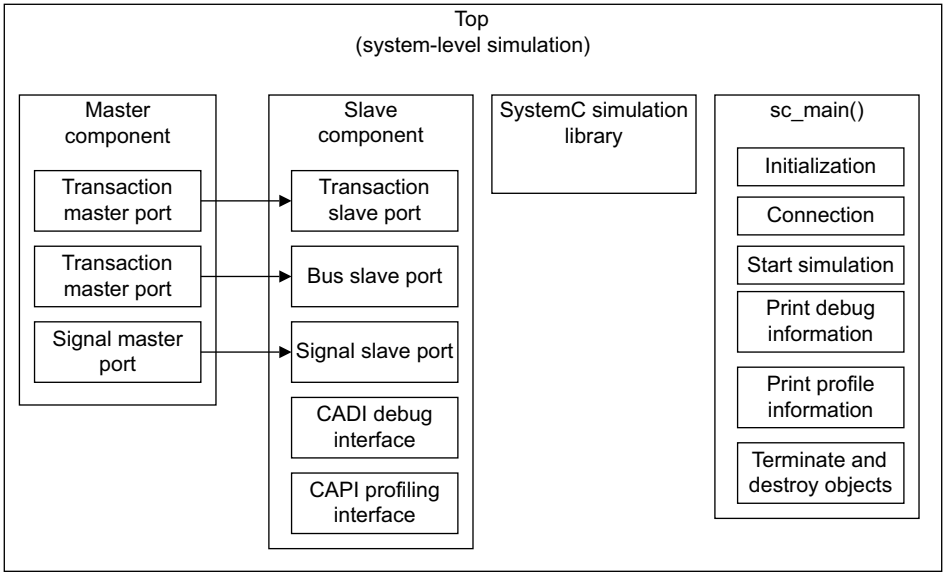

Figure 1-17 Block diagram of a system simulation

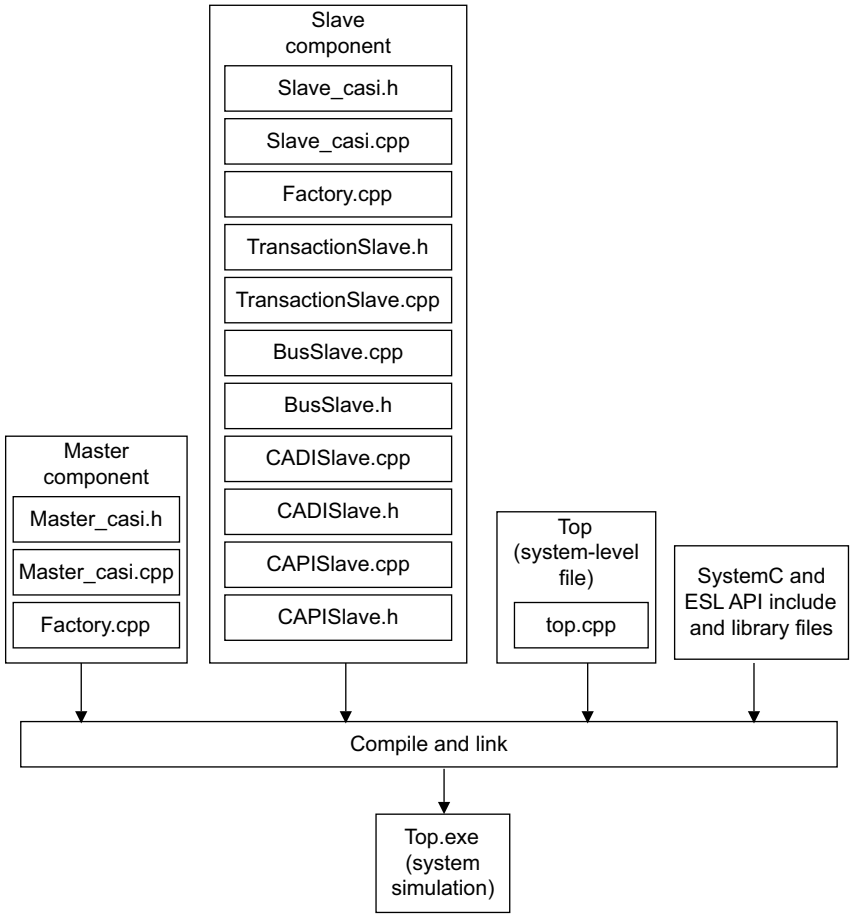

Figure 1-18 Files used for the top example

1.9 Overview of component creation

This section describes how to manually create your own components and make them available in a simulation system. The main steps are:

- creating the code with a text editor
- creating a project file for the component
- compiling and linking the code in the project
- editing any configuration options that are necessary to use your project in your simulation environment.

To create a component, implement the `CASIModule` interface and behavior and define the component resources. Components must define any ports that are required for communication with other components in the system.

Systems are hierarchical representations of hardware systems. A system can be a single component or consist of a hierarchical structure of subsystems and components.

1.9.1 Defining the component class

The component class must inherit from `CASIModule`. This module provides the necessary base functionality for components. Example 1-1 shows how the component class is entered explicitly:

Example 1-1 MyModel class

```
class MyModel : public CASIModule
{
public:
    MyModel(CASIModuleIF* parent, const std::string &name);
    string getName()     { return "MyModel"; }
    virtual void interconnect();
    virtual void init();
    virtual void terminate();

    // Interface functions for clocked components
    virtual void communicate();
    virtual void update();
};
```


Alternatively, the component class can be declared using the `CASIModule` macro as shown in Example 1-2:

Example 1-2 Using the CASIModule macro to declare a class

```
CASIModule(MyModel)
{
        //class members
};
```

1.10 Working with component ports

The CASI API distinguishes between master and slave ports:

- master ports are used to access connected slaves
- slave ports provide functions that access shared resources inside the component.

Ports can be created by instantiating the appropriate port classes.

- Master ports are instances of the `sc_port` class in SystemC using the transaction or slave interfaces.
- Slave ports are channels implementing the transaction or slave interfaces.

Example 1-3 shows the creation of master and slave ports:

Example 1-3 Creating two ports

```
sc_port<eslapi::casi_transaction_if, 1>* dram_port =
    new sc_port<eslapi::casi_transaction_if> (this, "dram");
registerPort(dram_port, "dram_port");

isrc_SSlave = new isrc_SS( this );
registerPort( isrc_SSlave, "irq_port" );
```

Note

The code in Example 1-3 creates one transaction master port and a signal slave port for IRQ:

- The transaction master port class is a generic one that is predefined in the ESL API and uses the `casi_transaction_if` interface.
- The signal slave port is a custom class derived from `casi_signal_slave` specifically for the purpose of controlling the IRQ signal. For Example 1-3, a separate file (for this example, `isrc_SS.cpp`) contains the code that implements a slave port and the `casi_signal_if` interface.

1.10.1 Using master ports

The following master ports can be created using the sc_port SystemC class:

Table 1-3 sc_port classes

Port type	sc_port definition
Signal master ports	sc_port<CASISignalIF, 1>
Transaction master ports	sc_port<CASITransactionIF, 1>
Bus master ports	sc_port<CASITransactionIF, 0>

Each created master port can communicate with the slave port that is connected to it.

A pointer to any created master port can be obtained by using the findPort() function. For performance reasons, however, it is recommended that you store a pointer for each master point immediately upon creation.

Example 1-4 shows the accessing of memory through a transaction master port.

Example 1-4 Reading from a port

```
uint64_t addr = 0xFF00;
uint32_t value[1];
uint32_t ctrl = 0;
dram_port->read(addr, value, &ctrl);
cout << "Value " << value << " has been read from address " << addr << endl;
```

For performance reasons, the read function of the master port does not check whether a slave port has been connected. This must be done once in the interconnect stage only. Therefore master ports provide the getSlaves() function as shown in Example 1-5:

Example 1-5 getSlaves function

```
if (dram_port->getSlaves() == NULL)
    cout << "Port not connected !" << endl;
```

The list of connected slave ports returned by the function getSlaves() must be non-NULL. If NULL is returned, the port has not been successfully connected.

AHB master ports

AHB transaction master ports use the dedicated `CASIAHBMasterPort` class to manage the AMBA protocol. See Appendix B *AMBA AHB TLM Specification for CASI*. the AHB documentation, and the example master and slave ports.

AXI master ports

AXI transaction master ports use the dedicated `CASIAXIMasterPort` class to manage the AMBA 3 AXI protocol. See Appendix C *AMBA AXI TLM Specification for CASI*, the AXI package documentation for details of the AXI protocol, and the example master and slave ports.

1.10.2 Using slave ports

Slave ports define access methods to shared resources in a component. Because these shared resources differ from case to case, there are no predefined port classes available for slave ports. Therefore, users must implement their own custom slave classes, derived from either `CASISignalSlave` or `CASITransactionSlave`.

Note

AMBA AXI, AHB, and APB transaction slave ports derive from intermediate classes that are themselves derived from `CASITransactionIF`.

Slave ports correspond to SystemC channels and implement the transaction or signal interfaces.

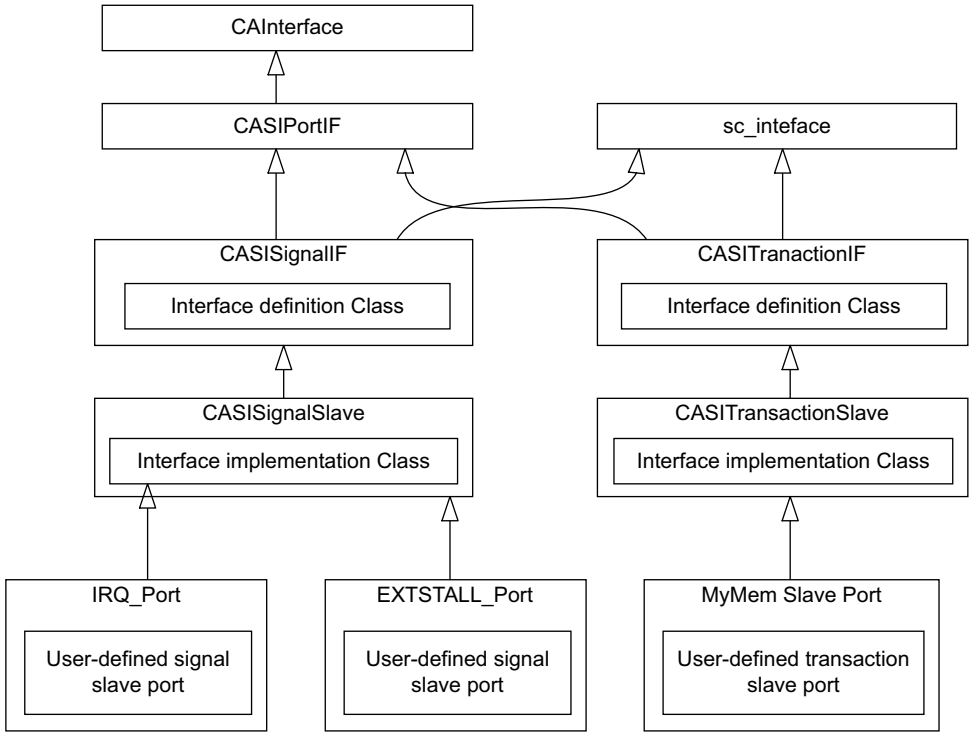

Figure 1-19 Class hierarchy showing user defined slave ports

The shared resources are typically encapsulated in the component class, but the access methods are defined in the port class (see Example 1-6). There are various ways that the port class can be given access to the resources. The most efficient way is to use `friend` class declarations This enables the port classes to access shared resources in the component without the overhead of function calls and is, therefore, the recommended way of shared resource access.

Example 1-6 MyMemSlavePort class

```
class MyMemSlavePort : public CASITransactionSlave
{
    MyModel* owner;
public:
    MyMemSlavePort(MyModel* _owner) : CASITransactionSlave("MyMemSlavePort")
    { owner = _owner; }
    CASIStatus read(uint64_t addr, uint32_t *dataP, uint32_t* ctrl=0);
    CASIStatus write(uint64_t addr, uint32_t *dataP, uint32_t* ctrl=0);
};
```

```
class MyModel : public CASIModule
{
    friend class MyMemSlavePort;
    friend class IRQ_Port;
    friend class EXTSTALL_Port;
public:
    …
};
```

Slave ports provide functions that can be accessed from master ports. These functions (typically virtual functions) must be overridden so that the appropriate function in the slave is executed when a master initiates a transaction.

Transaction Slave Ports

For transaction slave ports, the read and write functions must be overridden as shown in Example 1-7:

Example 1-7 Overriding read and write

```
CASIStatus MyMemSlavePort::read(uint64_t addr, uint32_t *value, uint32_t* ctrl)
{
    value[0] = owner->getMyMem(addr);
    return CASI_STATUS_OK;
}
CASIStatus MyMemSlavePort::write(uint64_t addr, uint32_t *value, uint32_t* ctrl)
{
    if (addr <= 0x3FF) {/* restricted access */
        owner->setMyMem(addr, value[0]);
        return CASI_STATUS_OK;
    }
    else {        /* no access above 1k */
        return CASI_STATUS_NOACCESS;
    }
}
```

Example 1-7 shows the definition of the transaction slave port read and write behavior. In this example, write access is restricted to the lower 1024 words.

Signal Slave Ports

The handling of signal slave ports is equivalent to the transaction slave ports, except that here the access functions `driveSignal()` and `readSignal()` must be implemented as shown in Example 1-8.

Example 1-8 Implementing driveSignal and readSignal

```
class IRQ_Port : public CASISignalSlave
{
    MyModel* owner;
public:
    IRQ_Port(MyModel* _owner) { owner = _owner };
    void driveSignal(uint32_t value, uint32_t* extValue);
    uint32_t readSignal();
};
IRQ_Port::driveSignal(uint32_t value, uint32_t*) {
    owner->interrupt = value;
};
uint32_t IRQ_Port::readSignal() {
    return owner->interrupt;
};
```

1.10.3 Defining the behavior of clocked components

Non-clocked components are passive and are only triggered by transactions from the outside.

Clocked components, however, can also have behavior that is independent of their interaction with other components. This behavior is defined in the communicate and update phases as shown in Example 1-9:

Example 1-9 Communicate and update phases of component

```
MyModel::communicate(){
    if (myModel.readDRAM == true)
        dram_port->read(addr, value, ctrl);
}

MyModel::update() {
    if (interrupt == true)     myCore.injectInterrupt();
    myModel.execute();
}
```

Example 1-9 on page 1-33 shows the implementation of communicate and update behavior and assumes that `MyModel` represents a core model that has a private data member called `myCore` that implements the behavior for the model.

The communicate behavior shows a read access from the `dram` port that is used if the `readDRAM` flag has been set in the model.

The update behavior injects an interrupt if the `interrupt` flag has been set externally (over the `irq` port) and then executes one cycle in the model.

1.10.4 Registering the clock port

A component that implements a clock slave interface must register this interface as the `clk-in` port in the constructor to be recognized as a clocked component.

```
registerPort(dynamic_cast<CASIClockSlaveIF*>(this), "clk-in");
```

If the clock is not registered, it might not be visible in the simulation environment.

1.10.5 Connecting a component to the clock

To connect a component to the clock, call the `casi_clocked()` function in the component constructor:

```
casi_clocked();
```

Alternatively, the `CASI_CLOCKED` macro can be used in the constructor:

```
CASI_CLOCKED();
```

Note

Instead of calling `casi_clocked()` in the constructor, you can call `registerClockSlave()` in the interconnect phase. See *The interconnect() function* on page 1-38.

1.10.6 Defining a factory class for the system

Note

This class is not required if you are simulating a standalone system using the standard SystemC and ESL API environment. It might be useful, however, in simulation environments that permit dynamic component creation and connection.

In addition to the component itself, a factory class is available that can be used by the third-party simulation systems to access the model.

The factory class must inherit from CASIFactory. The only functions that must be defined are:

- the constructor, this is named MyModelFactory() for the class in Example 1-10
- the createInstance() function.

Example 1-10 Factory class

```
class MyModelFactory : public CASIFactory
{
public:
    MyModelFactory(const std::string &name);
    ~MyModelFactory();
    CASIModuleIF * createInstance(CASIModuleIF * parent, const std::string &id);
};
```

The constructor simply calls the default constructor, passing the name of the component as a parameter. It is important that this name is equivalent to the name that is returned by the getName() function of the corresponding component. ARM recommends declaring the constant that contains the name of the model as a string:

```
#define MODEL_NAME "My_Model";
MyModelFactory::MyModelFactory() : CASIFactory (MODEL_NAME) {}
```

1.10.7 The factory member functions

The constructor implementing the system must call the parent's constructor and pass the name of the system as a parameter:

```
MyModelFactory::MyModelFactory() : CASIFactory("MyModel") {}
```

Use the createInstance() function to create an instance of the component by passing a pointer to the parent component as shown in Example 1-11:

Example 1-11 Creating an instance of a component

```
CASIModuleIF * MyModelFactory::createInstance(CASIModuleIF *parent,
                                const std::string & instance_name)
{
    return new MyModel(parent, instance_name);
}
```

1.11 Checklist for components

The implementation of any component must follow the rules listed in this section.

Use this list as a guideline to develop a new components and verify the correctness of the implementation.

1.11.1 The component class

This section contains the check list for the component class.

Defining your component class

1. Derive your component class from `CASIModule`:
   ```
   class MyModel : public CASIModule
   ```
   Note

   `CASIModule` has the public virtual function `ObtainInterface()`. All components deriving from `CASIModule` implement the `CAInterface` class. with at least revision 0 for the interface named CAInterface. The CAInterface class enables extending the ESL API interfaces with custom interfaces. The default implementation is sufficient if your component only uses the standard ESL API interfaces.

2. Declare all slave ports as `friend`:
   ```
   friend class MySlavePort;
   ```
3. Define a constant for the component name. This constant is used in the `getName()` function and in the component factory:
   ```
   #define MODEL_NAME "DLX_Core"
   ```

The component constructor

1. Create all ports in the component constructor:
   a. if you are creating a slave port, register it in the port list:
      ```
      registerPort(new MySlavePort(…), "irq");
      ```
   b. if you are creating a master port, keep a pointer to it for fast access:
      ```
      ext_mem = createPort(CASI_TRANSACTION_MASTER, "ext_mem");
      ```
      then register it in the port list:
      ```
      registerPort(ext_mem, "ext_mem");
      ```
   c. register the clock slave port if the component is clocked:
      ```
      registerPort(dynamic_cast<CASIClockSlaveIF*>(this), "clk_in");
      ```


2. If the component class is clocked, call `casi_clocked()` function:

   ```
   casi_clocked();
   ```

   Note

   You can register the clock in the interconnect stage instead of in the constructor. See *The interconnect() function* on page 1-38.

3. Define all parameters in the component constructor.

The component destructor

1. Delete only what has been created in the constructor. This includes the created ports.

The getName() function

1. Return the model-name defined in the component header file.
2. Ensure that this name is equivalent to the name used in the component factory's constructor.

The getProperty() function

Note

This function must be implemented for all models.

1. For `CASI_PROP_COMPONENT_TYPE`, use the `CASIComponentTypes` array to return the type of your component.
2. For `CASI_PROP_COMPONENT_VERSION`, return the version number of your component.
3. For `CASI_PROP_DESCRIPTION`, return a brief description of your component and its key features.
4. If your component has its own loader, use `CASI_PROP_LOADFILE_EXTENSION` identify the types of files to look for.

The setParameter() function

1. Implement this function if your component is configurable.
2. Distinguish initialization time parameters and runtime parameters.

3. If possible, do not access resources directly but instead set flags to control resource allocation.
4. Do not access resources that are allocated during `init()`.

   Note

   `setParameter()` is called before the `init()` function.

5. For initialization time parameters, defer the parameter evaluation to the init stage.
6. Call `CASIModule::setParameter()` to ensure that the parameter changes are registered in the parameter list.

The configure() function

1. Implement this function only if your component has subcomponents.
2. Do not define this function with an empty behavior, call `CASIModule::configure()` instead.
3. After calling `CASIModule::configure()`, set the parameters of all of your subcomponents here if necessary.

The init() function

1. Allocate all resources necessary for simulation here.
2. Call `CASIModule::init()`.
3. Allow for parameters that might have been set during the configure stage.

The interconnect() function

1. If your component has subcomponents, connect your subcomponents here.
2. If your component is clocked, register it with the `clockMaster`.

   ```
   getClockMaster()->registerClockSlave(this);
   ```

   Note

   Use this function if the clocks were not registered by calling `casi_clocked()` in the constructor stage. See *The component constructor* on page 1-36.

3. Call `CASIModule::interconnect()`.

The reset() function

1. Implement your reset behavior here.
2. Check the reset level to distinguish hard and soft reset if applicable.
3. If your model supports loading of object code, retrieve the filename for your model from the file list only during a hard reset.
4. Call `CASIModule::reset()`.

The terminate() function

1. You must implement this function if you allocate memory in the `init()` function.
2. Delete all objects created in `init()`.

The communicate() and update() functions

1. If your component is clocked these functions must be defined.
2. All communication with other components must be done in the `communicate()` function.
3. Updating of resources must be done in the `update()` function.

Note

It is technically possible to perform updating during the communicate phase, but this is not recommended because it can cause side effects or difficult to diagnose errors.

1.11.2 The port classes

1. It is typically only necessary to create your own slave port classes.

   It is recommended to use the master port classes that are provided by the ESL API.
2. In the constructor of each slave port, pass a pointer to the component class that the slave port belongs to.
3. Access the resources of the component class directly and avoid unnecessary function calls.

1.11.3 The factory class

1. You must have a factory class for every component you define if your simulation environment dynamically creates and configures components.
2. Pass the name of the component to the constructor of `CASIFactory`.
3. Ensure that the name is equivalent to the name used in the component factory constructor by using the name defined in the header file of the component.

1.12 CAInterface extensions

The ESL API interfaces support the `CAInterface` class to enable adding extensions to the APIs without breaking binary compatibility for already compiled models.

`CAInterface` is the base class for extendable component interfaces and uses the following software model:

components A component is a black-box entity that has a unique identity. A component provides concrete implementations of one or more interfaces. Each of these interfaces may expose different facets of the component's behavior. These interfaces are the only way to interact with the component.

interfaces An interface is an abstract class (consisting entirely of pure virtual methods), which derives from `CAInterface`, and which provides a number of methods for interacting with a component.

Because there is no way for a client to enumerate the set of interfaces that a component implements, the client must ask for specific interfaces by name. If the component does not implement the requested interface, it returns a NULL pointer.

(The implementation of a component's interfaces can be provided by one or several interacting C++ objects. The implementation is transparent to the client).

Interfaces are identified by a string name (of type `if_name_t`), and an integer revision (of type `if_rev_t`). A higher revision number indicates a newer revision of the same interface.

The `CAInterface::ObtainInterface()` method enables a client to obtain a reference to any of the interfaces that the component implements.

The client specifies the id and revision of the interface that it wants to request. The component returns NULL if it does not implement that interface or only implements a lower revision.

Each interface derives from `CAInterface` and a client can call `ObtainInterface()` on any interface pointer to obtain a pointer to any other interface implemented by the same component.

The following rules govern the use of components and interfaces:

- Each component is distinct. No two components can return the same pointer for a given interface. An `ObtainInterface()` call on one component must not return an interface on a different component.

- Each interface consists of a name, a revision number, and a C++ abstract class definition. The return value of `ObtainInterface()` is either NULL, or is a pointer that can be cast to the class type.
- Where two interfaces have the same `if_name_t`, the newer revision of the interface must be backwards-compatible with the old revision. (This includes the binary layout of any data-structures that it uses, and the semantics of any methods).
- During the lifetime of a component, any calls to `ObtainInterface()` for a given interface name and revision must always return the same pointer value. It must not matter which of the component's interfaces is used to invoke `ObtainInterface()`.
- All components must implement revision 0 of `eslapi::CAInterface`. This interface implements `eslapi::CAInterface` class.

Chapter 2
The Cycle Accurate Simulation Interface

This chapter describes the classes and member functions of the *Cycle Accurate Simulation Interface* (CASI) that are used in the creation and simulation of components. This chapter has the following sections:

2.1 Class overview

The *Cycle Accurate Simulation Interface* (CASI) defines how components are called by the scheduler and how they communicate with each other. Essentially CASI provides a communication library for SystemC that provides clock, transaction, and signal classes to enable transaction-level communication and cycle-based simulation modeling.

CASI provides the following types of class:

Component classes

The `CASIModule` class is the base class for any component. It provides the functionality for instantiating and configuring the model and sub-components. It also provides the API required for connecting other components. Unlike the interface classes, `CASIModule` is derived from `CASIModuleIF`.

Interface classes

The interface classes manage communication between components.

Clock classes

The clock classes manage connection to the simulation clocks.

Port classes

These classes inherit from other basic classes and simplify creating commonly used objects.

Support classes

These classes provide functionality for creating, saving, and restoring components. These classes are typically used by the simulator to enable additional functionality during simulation. The `CASIMMI` class, for example, enables describing and configuring the memory maps for a bus master.

Note

See the `CASITypes.h` file for definitions of enumerations and data structures that are used with the CASI interface.

2.1.1 Interface classes

The interface classes listed in Table 2-1 have a common base class, `CASIPortIF` that defines common functionality.

Table 2-1 Interface classes

Class Name	Class Description
`CASIPortIF`	This class is the base class for all interface classes. Both master and slave ports (including master and slave clock ports) derive from this class. It defines the basic port functionality.
`CASITransactionIF`	This class defines the interface for transaction slave ports (for example, memories). This can be used by any component that can be accessed like a memory (for example, DMA, Cache, and Bus). See *The CASITransactionIF interface* on page 2-56.
`CASISignalIF`	This class defines the interface for signal slave ports. This can be used for any model that has input signal pins (IRQ or Stall for example). See *The CASISignalIF Interface* on page 2-92.
`CASITransactionMasterIF`	This class defines the interface for transaction master ports. See *The CASITransactionMasterIF class* on page 2-72.
`CASISignalMasterIF`	This class defines the interface for signal master ports. See *The CASISignalMasterIF class* on page 2-96.

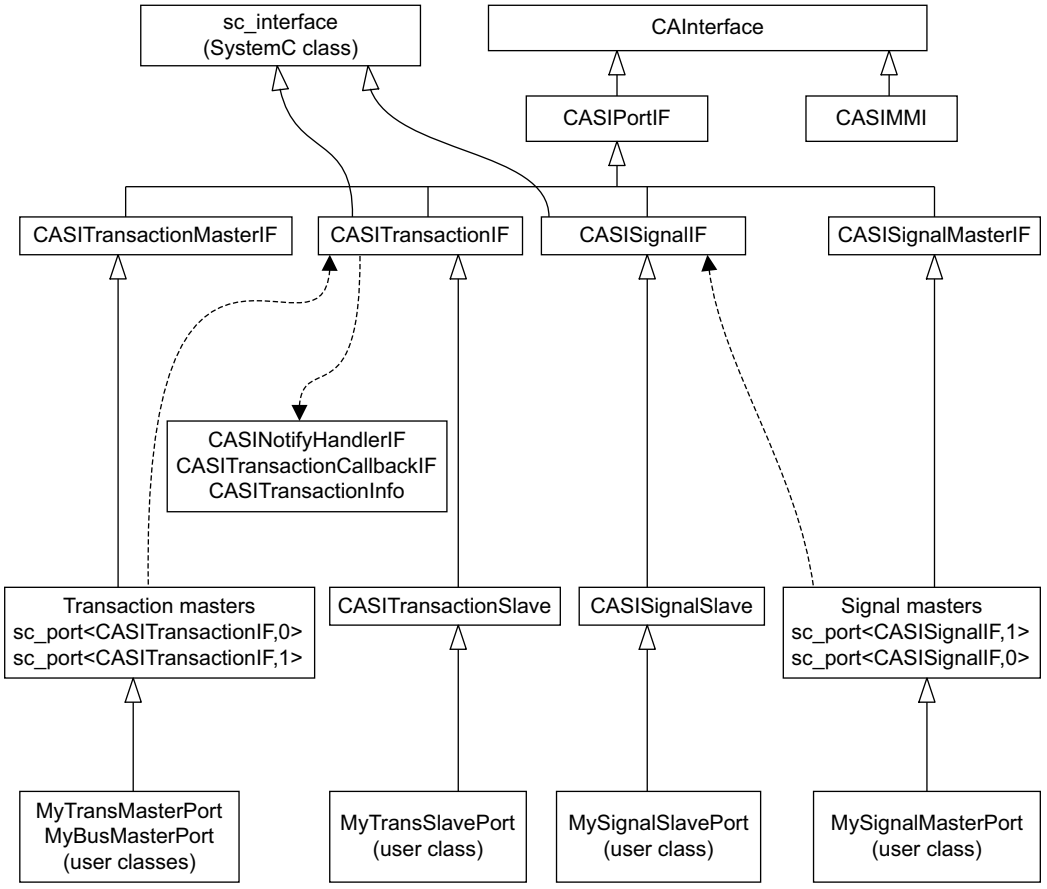

Figure 2-1 Class hierarchy of the interface classes

2.1.2 Port classes

The classes in Table 2-2 are convenience classes that are used to create the ports for a component.

Table 2-2 Predefined classes

Class Name	Class Description
`CASISignalSlave`	This is the base class for signal slave ports. It is derived from the `CASISignalIF` class. This type of port is used for any component that has input signal pins (IRQ or Stall for example). See *The CASISignalSlave class* on page 2-94.
`CASITransactionSlave`	This is the base class for transaction slave ports. It is derived from the `CASITransactionIF` class. This type of port is used for any component that can be accessed like memory (for example, DMA, Cache, and Bus). See *The CASITransactionSlave class* on page 2-66.
`sc_port<CASISignalIF, 1>`	This is the base class for signal master ports. The basic `sc_port` class is templated. Using `CASISignalIF` creates a signal master. Multiple signal slaves can be connected to a signal master. See *The signal interface classes* on page 2-91.
`sc_port<CASITransactionIF, 0>`	This is the base class for bus master ports. The basic `sc_port` class is templated. Using `CASITransactionIF` creates a transaction master. The use of 0 as the second parameter in the template indicates that the master can access multiple transaction slaves. That is, the class contains the address decoder logic that are required if using multiple transaction slaves. See *The predefined sc_port< CASITransactionIF, 0> class* on page 2-87.
`sc_port<CASITransactionIF, 1>`	This is the base class for transaction master ports. The basic `sc_port` class is templated. Using `CASITransactionIF` creates a transaction master. The use of 1 as the second parameter in the template indicates that the master can only access a single transaction slave. See *The predefined sc_port<CASITransactionIF, 1> class* on page 2-83.

2.1.3 Component and clock classes

The classes in Figure 2-2 on page 2-8 provide the clocking functionality for the ESL API and the system components.

Table 2-3 Interface classes

Class Name	Class Description
`CASIModule`	This is the base class for all CASI-compliant modules. The functions in this class provide the code for the different simulation stages and manage the component hierarchy and component interconnection. See *The CASIModule class* on page 2-10. The `communicate()` and `update()` functions (inherited from `CASIClockSlaveIF`) are used by the simulation engine to provide the cycle-based behavior for the models.
`CASIClockMasterIF`	This class defines the basic cycle-accurate simulation driver. The master clock calls `communicate()` and `update()` for all registered slave clocks based on the stimuli generated by the simulation kernel. This interface can be used to generate clock dividers or other types of custom clock driving components.
`CASIClockSlaveIF`	This class defines the basic cycle-accurate simulation execution unit for a module. A CASI module can define zero, one or several clock slaves. Each clock slave represent the cycle-based behavior of the appropriate model part. Communicate implements the cycle-based communication with other components, while update implements the cycle-based updating of the internal resources of the component. A `CASIClockSlaveIF` object must be connected to an appropriate `CASIClockMasterIF` object to receive the `communicate()` and `update()` calls.
`CASIClockDriver`	This class defines the interface for registering a slave port clock to a clock driver. The profiling interface uses the clock driver to control updating profiling information with the current cycle information. **Note** `CASIClockDriver` and `CASIClockDriverRoot` are used to setup a top-level clock for pure CASI systems. These classes are provided as reference implementations of the system clock and are not required for developing CASI components. Normally you are not required to derive your classes from this unless you implement a complex clock that does not have a simple ratio to the original clock or you implement a custom clock source.

Table 2-3 Interface classes (continued)

Class Name	Class Description
`CASIClockDriverRoot`	The `CASIClockDriverRoot` class is used by the CAPI profiling interface (see Chapter 4 *The Cycle Accurate Profiling Interface*).
`CASIClockMaster`	This class provides a basic implementation of the clock master components. You do not have to derive your classes from this unless you implement a complex clock that does not have a simple ratio to the original clock. See *The CASIClockMaster class* on page 2-41.
`CASIClockSlave`	This class provides a basic implementation of the clock slave port. The virtual `communicate()` and `update()` functions are implemented in the port instance. See *The CASIClockSlave class* on page 2-50.

Typically, these classes are not used directly, but methods in the classes are called by the component. An example of a slave clock registering itself with the master clock is shown in Example 2-1:

Example 2-1 Registering a clock

```
getMaster()->registerClockSlave(this);
```

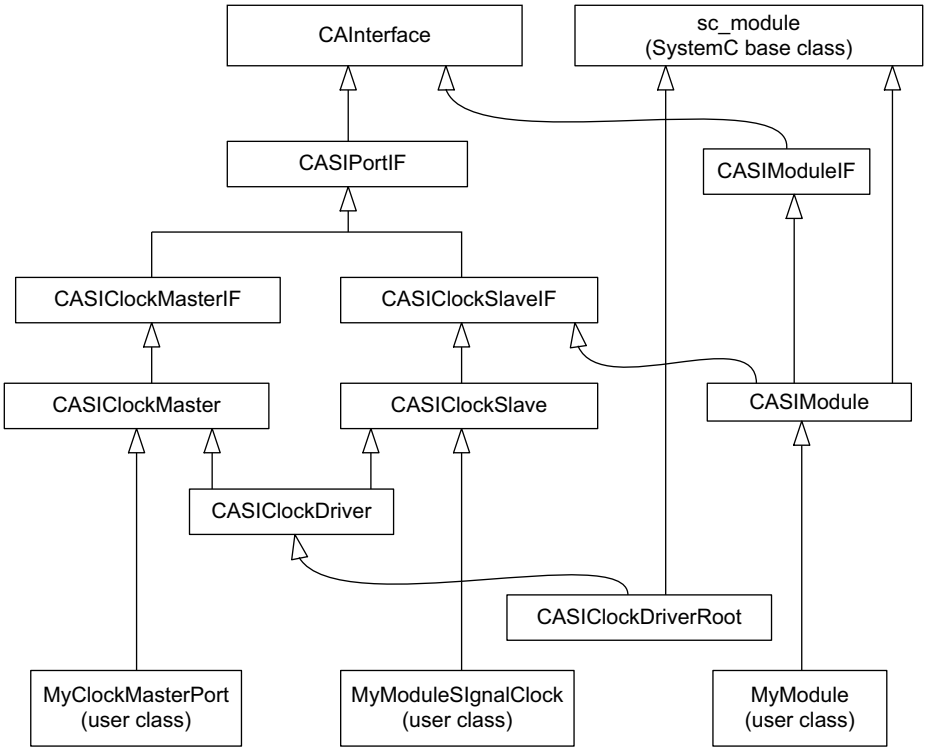

Figure 2-2 Class hierarchy for the component and clock classes

2.1.4 Support classes

In addition to the core simulation classes, the CASI library provides additional support interfaces.

Factory class

To facilitate dynamic loading of models, a `CASIFactory` interface is defined. Models implementing this interface can be loaded in a simulation environment dynamically, without requiring recompilation of the model and/or simulation environment. See *The component factory class CASIFactory* on page 2-100 for more information on the factory classes

Note

The factory classes are not required for simulation in a pure SystemC environment that does not dynamically load models.

Save and restore

The `CASISaveRestoreIF` interface is provided, to facilitate saving simulation state and resuming at a later time. Models implementing this interface can be saved on disk and restored at a later time with the same cycle count as when they were stopped. This is especially important if you are debugging long simulations and only the final part of the simulation requires replaying.

The data stream is defined in the `CASISaveRestore.h file`. See *The save/restore interface CASISaveRestore* on page 2-102 for more information on the save and restore classes.

Memory map interface

The CASIMMI interface enables configuration of memory maps for bus masters. See Chapter 5 *The CASI Memory Map Interface*.

2.2 The CASIModule class

The CASIModule class is the fundamental simulation class. Its primary function is to facilitate the instantiation and connection of the component models.

This section contains the following subsections:

- *CASIModule definition*
- *Functions to implement* on page 2-14
- *Functions for the simulation stages* on page 2-16
- *Functions for identification of a component* on page 2-22
- *Functions for the configuration of components* on page 2-25
- *Functions for port administration* on page 2-28
- *Functions for connecting components* on page 2-30
- *Functions to administer the component hierarchy* on page 2-32
- *Miscellaneous functions* on page 2-35.

2.2.1 CASIModule definition

The CASIModule class is defined by the class header listed in Example 2-2:

Example 2-2 CASIModule class

```
class CASIModule : public CASIClockSlaveIF, public CASIModuleIF,
        public sc_module
{
public:
    /* Constructor / Destructor */
    CASIModule(const sc_module_name & name, CASIModuleIF *_parent);
    CASIModule(CASIModuleIF *_parent, const sc_module_name & name);
    CASIModule();
    virtual ~CASIModule();

    // Functions called each cycle during the cycle-accurate simulation.
    // Ensure that the communicate and update functions of clocked
    // components are implemented in the component
    virtual void communicate (){
         message(CASI_MSG_ERROR, "Default communicate called.");
    }
    virtual void update (){
        message(CASI_MSG_ERROR, "Default update called.");
    }

    /* Functions for the different stages of simulation */
    virtual void configure();
```



```
virtual void init();
virtual void interconnect();
virtual void reset(CASIResetLevel level,
    const CASIFileMapIF *filelist =NULL);
virtual void terminate();

/* Functions for identification */
virtual std::string getName() = 0;
virtual CASIInterfaceType getType();
virtual std::string getInstanceName();
virtual std::string getInstanceID();

// Functions for model parameters
virtual std::string getProperty (CASIPropertyType property);
virtual void setParameter(const std::string & key,
    const std::string & value);
virtual std::string getParameter(const std::string & key);

// Interface with CADI/CAPI
virtual CADI * getCADI();
virtual CAPI * getCAPI();

virtual int getNumCAMMIs ();
virtual CAMMI * getCAMMI (int index = 0);

// Helper functions
virtual void casi_clocked(void);
virtual void casi_clocked(CASIPhase _phases);

// Retrieves the parameter list.
const CASIParameterMapIF* getParameterList();
// Checks to see if a parameter is valid.
virtual bool testParameter(const std::string& key,
    const std::string& value, std::string& error_message)
    { return true; }
// Registers a port.
void registerPort(CASIPortIF* port, const std::string& name);
void registerPortArray(CASIPortIF** port, const std::string& name);
void exportPort(CASIPortIF *c, const std::string& portName,
    const std::string& extPortName);
CASIPortIF* createPort(CASIInterfaceType type,
    const std::string& name, int ID = 0);
CASIPortIF* findPort(const std::string& name);
const CASIPortMapIF* getPortList();

// Creates a subcomponent based on dynamic component creation.
// Currently dynamic component creation is not fully supported.
CASIModuleIF * createSubcomponent( const std::string& type,
    const std::string& name);
// Creates a subcomponent based on dynamic component creation.
```

```
    // Currently dynamic component creation is not fully supported.
    // Unix/ELF specific function.
    CASIModuleIF * createSubcomponent( bool bDlopenGlobal,
        const std::string& type, const std::string& name);
    // Adds a child subcomponent.
    void addSubcomponent(CASIModuleIF *c);
    CASIModuleIF *getSubcomponent(const std::string &name);
    const std::vector<CASIModuleIF*>& getSubcomponentList();

    // Sets a parameter for a subcomponent.
    void setSubcomponentParameter(const std::string& name,
        const std::string& key, const std::string& value);
    // Connects subcomponent ports.
    void connect_ports(const std::string& comp1, const std::string& port1,
        const std::string& comp2, const std::string& port2);
    void setClockMaster(CASIClockMasterIF *clockFromParent);
    CASIClockMasterIF *getClockMaster();
    virtual CASIModuleIF * getParent() { return parent; }

    // Prints a message.
    virtual void message( const std::string& msg,
        CASIMessageType type = CASI_MSG_INFO );
    // Prints a message, vararg variant.
    virtual void message( CASIMessageType type, const char *fmt, ... );
    virtual CASIComponentLayout* getComponentLayout() { return NULL; }
    virtual CASIObjectLoader* getObjectLoader() { return NULL; }
    virtual CASIStatus launchDebugger(char * appli)
        { return CASI_STATUS_NOTSUPPORTED; }

    // Returns the external port map.
    // Currently port export is not fully supported.
    CASIExtPortMap * getExtPortMap(void) { return & extPortMaps; }

    // Define a parameter.
    void defineParameter(const std::string &key, const std::string &value,
                  const CASIParameterProperties *prop = NULL);

protected:
    // Defines a special parameter.
    void defineParameter(const std::string &key, const std::string &value,
        CASIParameterType type, bool is_runtime = true,
        bool is_private = false, bool is_readonly = false,
        const std::string &description="");
public:
    // slaves now know their master
    virtual void connect(CASIClockMasterIF * master);
    virtual void disconnect(CASIClockMasterIF * master);
    virtual CASIClockMasterIF * getMaster() { return clock; }

    // instance names remembered by ports
```



```
    virtual void setPortInstanceName(const std::string& name)
        { casiPortInstanceName = name; }
    virtual std::string getPortInstanceName()
        { return casiPortInstanceName; }

    // every interface/port has an owner
    virtual void setCASIOwner(CASIModuleIF * owner)
        { casiOwner = owner; }
    virtual CASIModuleIF * getCASIOwner() { return casiOwner; }

    // ports can be enabled/disabled
    virtual void enablePort(bool enable);
    virtual const bool isPortEnabled();

    // message functions now also available in ports
    virtual void pmessage( const std::string& msg,
        CASIMessageType type = CASI_MSG_INFO );
    virtual void pmessage( CASIMessageType type, const char *fmt, ... );

    // Return interface if requested
    virtual CAInterface * ObtainInterface(if_name_t ifName,
            if_rev_t minRev, if_rev_t *   actualRev)
    {
         if((strcmp(ifName,"eslapi.CASIModule2") == 0) && (minRev <= 0))
         {
               *actualRev = 0;
                return this;
         }
         return NULL;
    }

protected:
    CASIClockMasterIF * clock;   // component default master clock
    std::vector<CASIModuleIF*> subcomponents;     // List of children
    CASIParameterMap parameters; // List of component's parameters
    std::string instanceName;    // Name of the instance given at creation time

private:
    CASIModuleIF *parent;           // Direct parent of the component
    std::string instanceID;         // Name of the instance + parents
                                    // ("parent1.parent2.name")
    CASIPortMap ports;              // List of component's port objects
    CASIPortMap thisPorts;          // List of all ports that are this object
    CASIExtPortMap   extPortMaps;   // Map between the external ports and the
                                    // ports of this component
    std::vector<CASIPortArrayIF *> portArrays; // Registered arrays of ports
    bool m_bRegisterClockSlave; // This component will be registered as a
                                    //clock slave during interconnect
                                    // port name & internal port
    bool m_bOptimizedClockSlave;
```

```
    CASIPhase phases;
    uint32_t numOfTimes;
    uint32_t ratio;
    uint32_t offset;

    std::string casiPortInstanceName;    // every port knows its instance name
    CASIModuleIF* casiOwner;             // every port knows its owner
    bool casiIsEnabled;                  // ports can be disabled
    bool casiIsConnected;                // whether port is connected to a real
                                 // (architectural) port, or left unconnected.
};
```

Note

All component models must implement the interfaces present in `CASIModule`.

The `CASIModule` class provides methods to create a hierarchical model structure, search for child objects and ports, get and set parameters in a component, and to make connections between components.

The `configure()`, `init()`, `interconnect()`, `reset()`, `communicate()`, `update()` and `terminate()` functions represent the simulation stages in CASI. After constructing a module:

- `configure()` configures the parameters of the model
- `init()` initializes the model and allocates any internal data structures required
- `interconnect()` performs any internal interconnections required in the modules
- `reset()` resets the module (but typically does not reallocate memory)
- `communicate()` and `update()` manage the cycle-based behavior of the model
- `terminate()` performs any model clean-up and data de-allocation required.

2.2.2 Functions to implement

Some of the member functions present in the class definition in Example 2-2 on page 2-10 must be implemented by the user. See the following sections for the function definitions:

- *CASIModule::CASIModule()* on page 2-17
- *CASIModule::configure()* on page 2-17
- *CASIModule::init()* on page 2-17
- *CASIModule::interconnect()* on page 2-18
- *CASIModule::reset()* on page 2-18
- *CASIModule::communicate()* on page 2-19 (if cycle-based)
- *CASIModule::update()* on page 2-20 (if cycle-based)

- *CASIModule::terminate()* on page 2-20
- *CASIModule::getName()* on page 2-22
- *CASIModule::getProperty ()* on page 2-23
- *CASIModule::setParameter()* on page 2-25
- *CASIModule::testParameter()* on page 2-27
- *CASIModule::getCADI()* on page 2-36 (if CADI is used)
- *CASIModule::getCAPI()* on page 2-37 (if CAPI is used)
- *CASIModule::registerPort()* on page 2-28
- *CASIModule::registerPortArray()* on page 2-29 (if port arrays are used).

The following sections describe functions in the `CASIModule` class that have been added for convenience:

- *CASIModule::defineParameter()* on page 2-26
- *CASIModule::createPort()* on page 2-28
- *CASIModule::connect_ports()* on page 2-31.

The following sections describe functions that are primarily intended for the use only by the front end and are not typically used from within a component implementation and do not require reimplementation:

- *CASIModule::getType()* on page 2-22
- *CASIModule::getInstanceName()* on page 2-22
- *CASIModule::getInstanceID()* on page 2-22
- *CASIModule::getParameter()* on page 2-26
- *CASIModule::exportPort()* on page 2-29 (reserved for future use)
- *CASIModule::findPort()* on page 2-30
- *CASIModule::connect()* on page 2-31
- *CASIModule::disconnect()* on page 2-32
- *CASIModule::createSubcomponent()* on page 2-32
- *CASIModule::setSubcomponentParameter()* on page 2-34.
- *CASIModule::addSubcomponent()* on page 2-33.
- *CASIModule::getSubcomponent()* on page 2-33.
- *CASIModule::setClockMaster()* on page 2-34.
- *CASIModule::getClockMaster()* on page 2-35
- *CASIModule::getComponentLayout()* on page 2-34
- *CASIModule::getObjectLoader()* on page 2-34
- *CASIModule::launchDebugger()* on page 2-34
- *CASIPortIF::setPortInstanceName()* on page 2-39
- *CASIPortIF::getPortInstanceName()* on page 2-39
- *CASIPortIF::setCASIOwner()* on page 2-39

- *CASIPortIF::getCASIOwner()* on page 2-39
- *CASIPortIF::enablePort()* on page 2-39
- *CASIPortIF::isPortEnabled()* on page 2-39
- *CASIClockMaster::pmessage()* on page 2-48.

All functions that must be implemented in the derived class are declared as virtual. Overwriting the other functions has no effect.

The following sections group the functions by category:

- *Functions for the simulation stages*
- *Functions for identification of a component* on page 2-22
- *Functions for the configuration of components* on page 2-25
- *Functions for port administration* on page 2-28
- *Functions for connecting components* on page 2-30
- *Functions to administer the component hierarchy* on page 2-32
- *Miscellaneous functions* on page 2-35.

2.2.3 Functions for the simulation stages

The functions for the stages of simulation are usually called in a hierarchical way. The top-level component is called which in turn calls the same function in its subcomponents and so on until all components in the hierarchy have been called.

See the following sections for details of the constructors and member functions for the simulation stages:

- *CASIModule::CASIModule()* on page 2-17
- *CASIModule::configure()* on page 2-17
- *CASIModule::init()* on page 2-17
- *CASIModule::interconnect()* on page 2-18
- *CASIModule::reset()* on page 2-18
- *CASIModule::communicate()* on page 2-19
- *CASIModule::update()* on page 2-20
- *CASIModule::terminate()* on page 2-20
- *CASIModule::casi_clocked()* on page 2-20.

2.2.4 CASIModule::CASIModule()

The constructor for the component class is used to load and instantiate all of its sub-components.

```
CASIModule(const sc_module_name & name, CASIModuleIF *_parent)

CASIModule(CASIModuleIF *_parent, const std::string & name)

CASIModule()
```

The constructor expects a pointer to another `Component` object (that is the parent or owner) and a string (the ID), assigned to the new component. The pointer might be `NULL`, but if a parent is given then the ID name is formed from the name of the parent and the assigned ID separated by a period. This dotted notation is used extensively to uniquely identify a component in a system. Component models that instantiate other components directly must always pass a pointer to the owner.

2.2.5 CASIModule::configure()

The `configure()` function of a component is used to configure its sub-components by calling the appropriate `setParameter()` functions.

```
virtual void configure()
```

It is assumed that all of the sub-components have already been instantiated in the constructor. Implementing this function is only required if the component is a hierarchical component (that is, if it contains subcomponents).

The function is called after all models are constructed.

2.2.6 CASIModule::init()

This function is used to initialize the component.

```
virtual void init()
```

Every model that contains resources must implement this function if it is necessary to initialize private data members and allocate memory. This function is called after the configure function, therefore it is possible to initialize the model dependant on the parameters set.

The function is called after setting the model parameters.

Note

Objects created with `new` in the `init()` phase must be destroyed in the `terminate()` phase

2.2.7 CASIModule::interconnect()

This function is used to connect the components in a hierarchical manner.

```
virtual void interconnect()
```

Each component is responsible for connecting its sub-components. Implementing this function, therefore, is only required if the component is a hierarchical component (that is, if it contains subcomponents).

The function is called after ports are connected.

If using `casi_clocked` and the function is overridden, ensure that the base function is called.

2.2.8 CASIModule::reset()

The `reset()` function is used for resetting the state of the component and load object files.

```
virtual void reset(CASIResetLevel level, const CASIFileMapIF *filelist=NULL)
```

where:

`level` determines the reset level and can currently be one of the following values:

`CASI_RESET_HARD`
: This reset reloads the object files. This reset level is called once after the system has been created and when the simulation is restarted. Components that can load object files (for example Cores) are expected to load their object files.

`CASI_RESET_SOFT`
: This is a regular reset. The parameter `filelist` is empty.

`filelist` is used as shown in Example 2-3 on page 2-19 if new object files are to be loaded. The file list is a pointer to an `CASIFileMapIF` object containing the files to be loaded by the components. The components can pass their component ID and obtain a string containing the name of the file to be loaded.

Note

The `filelist` parameter will be obsoleted in a future version. `CADIExecLoadApplication()` will be used instead.

This function can be called more than once:

- It is first called after the interconnection is completed as a hard reset.
- It can also be called during the execution to restart a simulation and, optionally, reload new object files.

Caution

Because the return value of `getFile()` might not persist, you must immediately copy the return value to a new string.

Example 2-3 Using the filelist parameter

```
if (filelist->fileExists(getInstanceID())){
    // hard reset only: user has selected an input file
        string basename = filelist->getFile( getInstanceID() );
        string inputFileName = basename + ".lod";
        myModel->loadFile(inputFileName);
}
else    {
        // no file selected by user => e.g. load default file
        myModel->loadFile(default_filename);
}
// Now call the base classes
CASIModule::reset(level, filelist);
```

A convenience version of the function exists as:

```
void _reset(CASIResetLevel level, const char * name)
```

2.2.9 CASIModule::communicate()

This function is called each cycle during cycle-based simulation.

```
virtual void communicate() {
        message(CASI_MSG_ERROR, "Default communicate called.");
    }
```

During the communicate phase, the cycle-based components interact and exchange data. You must implement this function in your component if it uses the cycle-based simulation clock.

2.2.10 CASIModule::update()

This function is called each cycle during cycle-based simulation.

```
virtual void update() {
        message(CASI_MSG_ERROR, "Default update called.");
    }
```

During the update phase, the cycle-based components process their internal data. You must implement this function in your component if it uses the cycle-based simulation clock.

2.2.11 CASIModule::terminate()

This function is provided to do any clean up when the simulation is about be ended. Termination of the simulation happens when either:

- the `Close simulation` command is run in the front end
- one of the component requests termination due to a `HALT` instruction or an illegal state.

```
virtual void terminate()
```

The difference between `terminate()` and the destructor is that the terminate function allows the user to perform some cleanup actions without destroying the object itself. This function can be seen as the counterpart to the init function. Any memory allocated in the `init()` function must be freed in the terminate function.

Note

The functions `configure()`, `init()`, `reset()`, and `terminate()` are usually overridden because most models have special tasks to perform in these stages. The default behavior is to call the corresponding functions of any child objects (sub-components). Because of the hierarchical structure, each component is responsible for calling the appropriate function in their sub-components.

2.2.12 CASIModule::casi_clocked()

This function is provided for clocking the component to the default clock master.

By default, calling this function ensures that the model clock slave is registered with the default clock master of the model during the interconnect stage. Using this function requires that the function `CASIModule::interconnect()` is called for the model.

The longer version of the `casi_clocked()` function is provided for conditionally clocking the component.


```
virtual void casi_clocked(void)

virtual void casi_clocked(CASIPhase _phases, uint32_t _numOfTimes=0,
               uint32_t _ratio=1, uint32_t _offset=0)
```

where:

`phases` specifies the phase to register the slave for:

- `CASI_PHASE_BOTH`
- `CASI_PHASE_COMMUNICATE`
- `CASI_PHASE_UPDATE`

`numOfTimes` specifies the number of times the component is to be called. 0 means the component will be called indefinitely.

Note

This parameter has been deprecated and must not be used in new code.

`ratio` specifies the clock ratio between the master clock and the local clock to be used. A ratio of `n` means that the component will be called once every `n` cycles. It must be greater than 0.

Note

This parameter has been deprecated and must not be used in new code.

`offset` denotes the number of master clock cycles to elapse before the component is called for the first time. Default is 0.

Note

This parameter has been deprecated and must not be used in new code.

Note

Conditional registration requires a more complex scheduling mechanism that can impact performance.

Conditional registration is only recommended for components that are only called occasionally. In this case, the simulation performance benefits.

2.2.13 Functions for identification of a component

This section describes functions that return information about a component:

- *CASIModule::getName()*
- *CASIModule::getType()*
- *CASIModule::getInstanceName()*
- *CASIModule::getInstanceID()*
- *CASIModule::getParent()* on page 2-23
- *CASIModule::getProperty ()* on page 2-23.

2.2.14 CASIModule::getName()

This is a pure virtual and must be implemented by an object that implements the `CASIModule` interface.

```
virtual std::string getName() = 0
```

It is required to return a string that is the name of the component model. The name of the model is used to identify it in the ESL API installation and it is typically mapped to a library name.

2.2.15 CASIModule::getType()

This function returns the component type. The component type defines whether a component is a core, memory, or other type of model. This function is used for grouping and display purposes only.

```
virtual CASIInterfaceType getType()
```

2.2.16 CASIModule::getInstanceName()

This function returns a string that is the name assigned to the component object when it was created by its parent. It must not be overloaded. The default behavior implemented in the `CASIModule` class is sufficient.

```
virtual std::string getInstanceName()
```

2.2.17 CASIModule::getInstanceID()

This function returns a string that is the ID name assigned to the component object when the constructor is invoked. The ID name usually consists of its name and the names of all the parents. It must not be overloaded. The default behavior is sufficient.

```
virtual std::string getInstanceID()
```


2.2.18 CASIModule::getParent()

This function returns a pointer to the parent component as assigned during construction. There is no requirement to overload this function.

```
virtual CASIModuleIF* getParent()
```

2.2.19 CASIModule::getProperty()

This function enables retrieving certain component properties.

```
virtual std::string getProperty(CASIPropertyType property)
```

where:

`property` specifies the desired `property`. The valid values for this enumeration are:

`CASI_PROP_LOADFILE_EXTENSION`

This property defines the file extension for the files that can be loaded by the component. If this property is defined, the user can select an input file to be loaded by the component on startup.

`CASI_PROP_COMPONENT_TYPE`

The component type defines whether a component is a core, memory, or other type of model. This function is used for grouping and display purposes only.

The following types are defined in the `CASIComponentType` enumerator:

- `CASI_TYPE_CLOCKDIVIDER`
- `CASI_TYPE_CORE`
- `CASI_TYPE_CORE_DSP`
- `CASI_TYPE_CORE_MC`
- `CASI_TYPE_COPROCESSOR`
- `CASI_TYPE_MEMORY`
- `CASI_TYPE_BUS`
- `CASI_TYPE_PERIPHERAL`
- `CASI_TYPE_ARBITER`
- `CASI_TYPE_CACHE`
- `CASI_TYPE_FIFO`
- `CASI_TYPE_OTHER`

By accessing the string array `CASIComponentTypes` with the component type, you can retrieve the correct type string. For example,

```
return CASIComponentTypes[CASI_TYPE_CORE];
```

CASI_PROP_COMPONENT_VERSION

This property defines the version number of the model or system. This feature is useful for revision control and bug tracking in the models.

CASI_PROP_CADI_SUPPORT

If the component provides a CADI interface, the function must return yes for this property. Default is no.

CASI_PROP_MULTI_INSTANCE_CAPABLE

Components that are not multi-instance capable can indicate this by returning no. This tells the simulation environment to forbid multiple instances of such a component in a design. Default is yes.

CASI_PROP_SAVE_RESTORE

Default is no. If a component supports save/restore, it must indicate this by returning yes for this property.

CASI_PROP_CADI_DEBUGGER

This entry contains the name of the external debugger. If this property returns a non-empty string, The simulation environment allows launching an external debugger script.

The script-name is start*debug_name*Debugger with a .bat extension on Windows, where *debug_name* is the name of the debugger that was specified.

CASI_PROP_DBG_START_COMMAND

If this property is specified together with the DEBUGGER property, the command returned here is executed from the current directory. No script is called.

CASI_PROP_DOCUMENTATION

This property can be used to provide the path and filename of the documentation, relative to the location of the model library (.dll or .so).

CASI_PROP_IP_PROVIDER

Name of the IP Provider.

CASI_PROP_COMPONENT_VARIANT

Variant information.

`CASI_PROP_MSG_PREPEND_NAME`

If this property returns `yes`, all messages for this component are prepended with the component instance name, and messages from ports are prepended with the instance name and port name.

`CASI_PROP_REPORT_FILE_EXT`

If set, the model expects the application file extension to be present when reset is invoked.

`CASI_PROP_WINDOW_COLOR`

If this property is a window color hint.

`CASI_PROP_LOADS_APPLICATION_CODE`

Returns `yes` if this component can load application code (such as, for example, a top level component for a multicore processor like MPCore).

`CASI_PROP_EXECUTES_SOFTWARE`

Returns `yes` if this component can execute software (it is, for example a core) and allows debuggers to connect to it (for example the leaf components for a multicore processor, such as MPCore).

2.2.20 Functions for the configuration of components

This section describes functions that configure component parameters:

- *CASIModule::setParameter()*
- *CASIModule::getParameter()* on page 2-26
- *CASIModule::getParameterList()* on page 2-26
- *CASIModule::defineParameter()* on page 2-26
- *CASIModule::testParameter()* on page 2-27.

2.2.21 CASIModule::setParameter()

This function sets variables in the model.

```
virtual void setParameter(const std::string& key, const std::string& val)
```

where:

`key` is the key for the parameter to set.

`val` is the value to be set.

An example might be the base address of a memory component. Parameter values are always passed as strings but they can represent any type of data. This function is typically overloaded. The default behavior is to maintain the list of parameters and values and the parent function `CASIModule::setParameter()` must always be called.

Note

This function might get called before all resources have been allocated.

2.2.22 CASIModule::getParameter()

This function returns the string that represents the value of a parameter.

```
virtual std::string getParameter(const std::string& key)
```

where:

`key` is a string that is the name of the desired parameter.

This function must not be overloaded. The default behavior is to search the parameter list and return the value string if found.

2.2.23 CASIModule::getParameterList()

This function returns a list of parameters.

```
const CASIParameterMapIF* getParameterList()
```

This function is fully defined in the `CASIModule` class and is typically used only by the front ends and there must not be any requirement to modify or call this function from the user side. See `CASIAuxIF.h` for details on the parameter list format.

2.2.24 CASIModule::defineParameter()

This function is called from the constructor of a component to define the admissible parameters and initialize them with default values.

```
void defineParameter(const std::string &key, const std::string &value,
            CASIParameterType type, bool is_runtime = true,
            bool is_private = false, bool is_readonly = false,
            const std::string &description="")

void defineParameter(const std::string &key, const std::string &value,
            const CASIParameterProperties *prop = NULL)
```


where:

`type` can be `string`, `bool`, `value`, or `undefined`.

`range` is used when the type is `value`, it defines admissible value ranges.

`is_runtime` is set to `true` to indicate that a parameter can be changed at runtime.

`is_private` is set to `true` to indicate that a parameter is only available in design-mode.

`is_readonly` is set to `true` to make the parameter read only.

`description` is an optional text string that describes the parameter.

`prop` provides the parameter values as a passed structure:

```
struct CASIParameterProperties
{
    CASIParameterType type;
    CASINumberRange* range;
    bool is_runtime;
    bool is_private;
    bool is_readonly;
    CASIParamSymbols *symbols;
    char description[CASI_DESCRIPTION_SIZE];
};
```

See the `CASITypes.h` file for definitions of `CASIParameterType`, `CASINumberRange`, and `CASIParamSymbols`.

The function calls `setParameter()` to set the parameter to its initial value. This function must not be overloaded.

2.2.25 CASIModule::testParameter()

Plugins can call `testParameter()` to check whether a parameter is admissible.

```
virtual bool testParameter(const std::string& /* key */,
              const std::string& /* value*/,
              std::string& /* error_message */)
```

where:

`key` is the key for the parameter to set.

`val` is the value to be set.

`error_message` is an error string that can report error details to the user if the function returns `false`.

This function must be implemented when the parameter properties do not contain sufficient information for identifying valid parameter entries (if only a specific set of strings is accepted, for example).

2.2.26 Functions for port administration

This section describes functions that create and administer ports:

- *CASIModule::createPort()*
- *CASIModule::registerPort()*
- *CASIModule::exportPort()* on page 2-29
- *CASIModule::findPort()* on page 2-30
- *CASIModule::getPortList()* on page 2-30.

2.2.27 CASIModule::createPort()

This function is used to create master ports because these ports do not require customization.

```
CASIPortIF * createPort(CASIInterfaceType type, const std::string& name,
        int ID=0)
```

where:

`type` is the type of port to create.

The definition of `CASIInterfaceType` is:

```
enum CASIInterfaceType{CASI_COMPONENT = 0,CASI_CLOCK_SLAVE = 1,
CASI_CLOCK_MASTER = 2, CASI_TRANSACTION_SLAVE = 3,
CASI_TRANSACTION_MASTER = 4, CASI_SIGNAL_SLAVE = 5,
CASI_SIGNAL_MASTER = 6, CASI_TRANSACTION_CALLBACK = 7
}
```

`name` is the port name.

`ID` is an optional numeric identifier for the port.

All master ports have the same simple functionality, they can be connected to a slave port and they pass on the function call to the connected slave port.

2.2.28 CASIModule::registerPort()

If ports are created by using the `new` operator, the port objects must be registered so that the port information can be queried by the scheduler.

```
virtual void registerPort(CASIPortIF * port, const std::string& name)
```

where:

`port` is a pointer to an interface.

`name` is the port name.

`registerPort()` allows registering all types of ports and expects a pointer to an `CASIPortIF` class. Both user defined as well as predefined port classes can be registered using this method.

CASIPortIF is the base class for all ports and is inherited by the CASIClockMaster, CASIClockSlave, CASISignalSlave and the sc_port<> classes.

2.2.29 CASIModule::registerPortArray()

Components can have an array of similar ports (for example interrupt input ports). The port objects must be registered so that the port information can be queried by the scheduler.

```
void registerPortArray(CASIPortIF ** portArray, const uint32_t arraySize,
                       const std::string& name)
```

where:

portArray	is a pointer to an array of interfaces.
arraySize	is number of elements in portArray.
name	is the port name.

registerPortArray() allows registering all types of ports and expects a pointer to an array of pointers to CASIPortIF class. Both user defined as well as predefined port classes can be registered using this method.

CASIPortIF is the base class for all ports and is inherited by the CASIClockMaster, CASIClockSlave, CASISignalSlave and the sc_port<> classes.

2.2.30 CASIModule::exportPort()

This function is reserved for future use. It can be used to create ports by passing on a subcomponent port. This allows creating ports that do not require an additional level of function calls in the model.

```
void exportPort(CASIModuleIF *c, const std::string& portName,
                const std::string& extPortName)
```

where:

c	is a pointer to a module. CASIModuleIF defines the basic interfaces for integrating the CASI module in the simulation environment. The configure(), init(), interconnect(), reset(), communicate(), update(), and terminate() functions in the class represent the simulation stages in CASI.
portName	is the port name in the module.
extPortName	is the name of the exported port.

2.2.31 CASIModule::findPort()

This function takes a port name as a parameter and returns a pointer to the port object if it is found. It returns `NULL` if a port of this name could not be found.

```
CASIPortIF * findPort(const std::string& name)
```

Note

Ports must be explicitly deleted in the destructor of the component. Registered ports are not automatically deleted.

By default, calling this function ensures that the model clock slave is registered with the default clock master of the model during the interconnect stage. This also requires that `CASIModule::interconnect()` is called for the model.

2.2.32 CASIModule::getPortList()

This function returns a list of ports.

```
const CASIPortMapIF* getPortList()
```

This function is fully defined in the `CASIModule` class and is typically used only by the front ends and there is no requirement for the user to modify or call this function.

See the `CASIAuxIF.h` file for a description of `CASIPortMapIF`.

2.2.33 Functions for connecting components

This section describes functions that are used to connect ports:

- *CASIModule::connect_ports()* on page 2-31
- *CASIModule::connect()* on page 2-31
- *CASIModule::disconnect()* on page 2-32.

2.2.34 CASIModule::connect_ports()

The `connect_port()` function is used to connect two subcomponents. The components are connected through their ports (which must also be passed as parameters).

```
void connect_ports(const std::string& comp1, const std::string& port1,
                const std::string& comp2, const std::string& port2)
```

where:

`comp1` is the name of a module.

`port1` is the name of the port in module `comp1`.

`comp2` is the name of a module.

`port2` is the name of the port in module `comp2`.

Note

Transaction slave ports can only be connected to transaction master ports and signal slave ports can only be connected to signal master ports.

It is not possible to connect:

- transaction ports to signal ports
- two slave ports to each other
- two master ports to each other.

2.2.35 CASIModule::connect()

```
virtual void connect(CASIPortIF * master)
```

The `connect()` function is used during the interconnect stage by the simulation engine to pass the pointer of the master to the slave port. Use the `getMaster()` function to determine the master this slave is connected to.

The components are connected through their ports (which must also be passed as a parameter).

Note

Transaction slave ports can only be connected to transaction master ports and signal slave ports can only be connected to signal master ports.

It is not possible to connect:

- transaction ports to signal ports
- two slave ports to each other
- two master ports to each other.

2.2.36 CASIModule::disconnect()

The `disconnect()` function is used to remove the interface specified as the parameter from the list of connections.

```
virtual void disconnect(CASIPortIF * master)
```

Note

It is important to properly implement the `disconnect()` function because it is used by probes that are dynamically connected and disconnected at runtime.

It is not possible to connect transaction ports to signal ports or to connect two slave ports or two master ports to each other.

2.2.37 Functions to administer the component hierarchy

These functions are already fully defined in the module class. There is no requirement to define any additional behavior for the following:

- *CASIModule::createSubcomponent()*
- *CASIModule::addSubcomponent()* on page 2-33
- *CASIModule::getSubcomponent()* on page 2-33
- *CASIModule::getSubcomponentList()* on page 2-34
- *CASIModule::setSubcomponentParameter()* on page 2-34
- *CASIModule::getComponentLayout()* on page 2-34
- *CASIModule::getObjectLoader()* on page 2-34
- *CASIModule::launchDebugger()* on page 2-34
- *CASIModule::setClockMaster()* on page 2-34
- *CASIModule::getClockMaster()* on page 2-35.

2.2.38 CASIModule::createSubcomponent()

The `createSubcomponent()` function is used to instantiate a sub-component by calling the component factory. It automatically calls `registerComponent()` for the created component if the component factory finds the component.

```
CASIModuleIF *createSubcomponent(const std::string& type,
        const std::string& name)
```

where:

`type` is a string that indicates the module type.

`name` is the name of the module.

The function is already fully defined in the Component class. There is no requirement to define any additional behavior.

The function declaration below is reserved for future use and will create a subcomponent based on dynamic component creation:

```
CASIModuleIF* createSubcomponent( bool bDlopenGlobal,
        const std::string& type, const std::string& name)
```

2.2.39 CASIModule::addSubcomponent()

The addSubcomponent() function is used to place a pointer to the component interface of a child object in the list of children of the parent.

```
void addSubcomponent(CASIModuleIF *c)
```

where:

c is a pointer to a module.
CASIModuleIF defines the basic interfaces for integrating the CASI module in the simulation environment.

Any model that creates child components must call this function passing a pointer to the interface of the child component.

2.2.40 CASIModule::getSubcomponent()

```
CASIModuleIF *getSubcomponent(const std::string& name)
```

The getSubcomponent() function is used to search through the list of child objects.

where:

name is a string that is the instance ID of the desired object.

It checks its own ID first and then searches through the list of any child objects. If the ID matches any object, that object's pointer to the Component interface is returned. If nothing matches, a null is returned. The search method is recursive so the children of children are searched. The comparison method at the level of a discrete component is designed to also match on the last discrete ID in a dotted-notation instance ID (for example compxID.compyID would match compyID) so component models can still be found in a hierarchical system design.

2.2.41 CASIModule::getSubcomponentList()

The `getSubcomponentList()` function returns a pointer to a copy of a list of the component and any children it might have.

```
const std::vector<CASIModuleIF*>& getSubcomponentList()
```

This function is to be used by the front end only and there is no requirement to call it from within a component model.

2.2.42 CASIModule::setSubcomponentParameter()

The `setSubcomponentParameter()` function is used to set values of a parameter `name` in a model. Parameter values are always passed as strings but they can represent any type of data.

```
void setSubcomponentParameter(const std::string& name, const std::string& key,
                        const std::string& value)
```

This function is usually overloaded. The default behavior is to maintain the list of parameters and values.

2.2.43 CASIModule::getComponentLayout()

This function is used to return display details for the component.

```
virtual CASIComponentLayout* CASIModule::getComponentLayout()
```

2.2.44 CASIModule::getObjectLoader()

This function is used by the environment to get the loader for a module.

```
virtual CASIObjectLoader* CASIModule::getObjectLoader()
```

2.2.45 CASIModule::launchDebugger()

This function is used by the environment to launch the debugger for the target and load the specified application.

```
virtual CASIStatus CASIModule::launchDebugger (char * appli)
```

2.2.46 CASIModule::setClockMaster()

The `setClockMaster()` function is used to assign the component to the master clock.

```
void setClockMaster(CASIClockMasterIF *clockFromParent)
```


2.2.47 CASIModule::getClockMaster()

The `getClockMaster()` function is used to get the handle to the master clock

```
CASIClockMasterIF *getClockMaster()
```

2.2.48 Miscellaneous functions

See the following sections for details on miscellaneous functions:

- *CASIModule::message()*
- *CASIModule::getCADI()* on page 2-36
- *CASIModule::getCAPI()* on page 2-37
- *CASIModule::getNumCAMMIs()* on page 2-37
- *CASIModule::getCAMMI()* on page 2-37.

2.2.49 CASIModule::message()

Use the `message()` functions to display messages in the output window of the simulator. All output from a component must be created in this way. There are two forms of the message function:

- one takes a simple string and an optional message type as a parameter
- the other has a similar form to `printf()`.

```
void message(const std::string& msg, CASIMessageType type = CASI_MSG_INFO )
```

```
void message(CASIMessageType type, const char *fmt, ... )
```

In addition to displaying messages in the output window, the message function can also be used to inform the simulation front-end about errors and exceptions.

The message-type parameter provides information about the type of message. The supported message types are:

`CASI_MSG_FATAL_ERROR`

This message type signals a fatal error. In this case, the error message is displayed in a separate message box and also printed in the output window preceded with the text `FATAL ERROR:`

The only way to recover from a fatal error is to load the system again.

`CASI_MSG_ERROR`

This message type indicates an error in the simulation. The simulation can be continued from this point. The message is displayed in a separate message box window, and is also printed in the output window preceded with the text `ERROR:`.

The user can now choose whether to abort the simulation or ignore the error.

`CASI_MSG_WARNING`

This type classifies the message to be a warning. In this case the message string is printed in the output window preceded by the text `WARNING:` .

Note

It is also possible to make the simulation stop the execution on warnings.

`CASI_MSG_INFO`

This is the default message type. The message string is simply printed in the output window.

`CASI_MSG_DEBUG`

If a message is classified as being a debug message, it is only printed if the debug version of build is used. The message is preceded with the text `DEBUG:`

ARM recommends using the `message()` function to create user output.

2.2.50 CASIModule::getCADI()

The function `getCADI()` must be implemented if an CADI interface exists for the component.

```
virtual CADI *getCADI()
```

This function simply returns a pointer to the CADI interface. See Chapter 3 *The Cycle Accurate Debug Interface* for details about defining an CADI interface for your component.

2.2.51 CASIModule::getCAPI()

The function `getCAPI()` must be implemented if an CAPI interface exists for the component.

```
virtual CAPI *getCAPI()
```

This function simply returns a pointer to the CAPI interface. See Chapter 4 *The Cycle Accurate Profiling Interface* for details about defining an CAPI interface for your component.

2.2.52 CASIModule::getNumCAMMIs()

The function `getNumCAMMIs()` returns the number of Memory Map Interfaces for the component.

```
virtual int *NumCAMMIs()
```

2.2.53 CASIModule::getCAMMI()

The function `getCAMMI()` returns a pointer to the specified Memory Map Interface.

```
virtual CAPI *getCAMMI(int index=0)
```

2.3 The CASIPortIF class

This is the base class for all master and slave CASI interfaces. The class definition is shown in Example 2-4.

Example 2-4 CASIPortIF

```
class CASIPortIF
{
public:
    virtual ~CASIPortIF() {};
    virtual std::string getName() = 0;
    virtual CASIInterfaceType getType() = 0;
    virtual std::string getPortInstanceName() = 0;
    virtual void setPortInstanceName(const std::string& name) = 0;
    virtual void setCASIOwner(CASIModuleIF *) = 0;
    virtual CASIModuleIF *getCASIOwner() = 0;
    virtual void enablePort(bool enable) = 0;
    virtual const bool isPortEnabled() = 0;
    virtual void setConnected(bool connected) = 0;
    virtual const bool isConnected() = 0;
    static if_name_t IFNAME() {return "eslapi.CASIPortIF2";}
    static if_rev_t IFREVISION() {return 0;}
};
```

The following functions are present in all classes that inherit from `CASIPortIF`:

- *CASIPortIF::getName()*
- *CASIPortIF::getType()* on page 2-39
- *CASIPortIF::getPortInstanceName()* on page 2-39
- *CASIPortIF::setPortInstanceName()* on page 2-39
- *CASIPortIF::setCASIOwner()* on page 2-39
- *CASIPortIF::getCASIOwner()* on page 2-39
- *CASIPortIF::enablePort()* on page 2-39
- *CASIPortIF::isPortEnabled()* on page 2-39
- *CASIPortIF::setConnected()* on page 2-40
- *CASIPortIF::setConnected()* on page 2-40.

2.3.1 CASIPortIF::getName()

This function can be overridden to return the name of the port.

```
std::string getName() =0
```


2.3.2 CASIPortIF::getType()

This function returns the port type.

```
CASIInterfaceType getType() =0
```

2.3.3 CASIPortIF::getPortInstanceName()

This function can be overridden to get the port instance name.

```
virtual std::string getPortInstanceName() =0
```

2.3.4 CASIPortIF::setPortInstanceName()

This function can be overridden to set the port instance name.

```
virtual void setPortInstanceName(const std::string & name) =0
```

2.3.5 CASIPortIF::setCASIOwner()

This function sets the module that owns the port.

```
virtual void setCASIOwner(CASIModuleIF* owner)=0
```

2.3.6 CASIPortIF::getCASIOwner()

This function gets the module that owns the port.

```
virtual CASIModuleIF *setCASIOwner()=0
```

2.3.7 CASIPortIF::enablePort()

This function enables the port.

```
virtual void enablePort(bool enable) =0
```

2.3.8 CASIPortIF::isPortEnabled()

This function returns the enable state. Default implementation is always enabled.

```
virtual const bool isPortEnabled() =0
```

2.3.9 CASIPortIF::setConnected()

This function sets a flag weather this port has been connected to a real (architectural) port or not. Ports that are left unconnected are connected by default by the tool to default sinks or sources.

```
virtual void setConnected(bool connected) =0
```

2.3.10 CASIPortIF::isConnected()

This function returns the connection status for the port.

```
virtual const bool isConnected() =0
```

2.3.11 CASIExtPortMapIF

This class is used to find external ports by name.

Example 2-5 CASIExtPortMapIF

```
class WEXP CASIExtPortMapIF
{
    public:
        virtual ~CASIExtPortMapIF() {};
        virtual bool find( const std::string &extName ) const = 0;
        virtual bool add( const std::string &extName, const CASIModuleIF *comp,
                          const CASIPortIF *port )  = 0;

        virtual bool remove( const std::string &extName ) const = 0;
        virtual bool getFirst( std::string &extName, CASIModuleIF **comp,
                          CASIPortIF **port ) const = 0;

        virtual bool getNext( const std::string &currentKey,
                         std::string &extName, CASIModuleIF **comp,
                           CASIPortIF **port ) const = 0;
        virtual bool getNext( std::string &extName, CASIModuleIF **comp,
                          CASIPortIF **ptr ) const = 0;
        virtual bool clear() = 0;
        virtual int  count() const = 0;
};
```

2.4 The clock interface classes

The ESL API provides interfaces for clocked components and for components that act as clock drivers. The simulation engine has one central clock that is the base to every other clock in a system.

2.4.1 The CASIClockMaster class

The class `CASIClockMaster` provides the interface for components that can act as clock drivers for other components.

This class can be used to generate clock dividers or other types of custom clock driving components (for example clock dividers).

Example 2-6 CASIClockMaster class

```
class CASIClockMaster : public CASIClockMasterIF
{
public:
    virtual CAInterface * ObtainInterface( if_name_t ifName, if_rev_t minRev,
                            if_rev_t * actualRev);
    CASIClockMaster();
    CASIClockMaster(CASIModuleIF * owner);
    virtual ~CASIClockMaster() ;

    // Functions to be implemented.
    virtual void registerClockSlave(CASIClockSlaveIF *) = 0;
    virtual void unregisterClockSlave(CASIClockSlaveIF *) = 0;
    virtual void registerClockSlave(CASIClockSlaveIF *slave,
          CASIPhase phase, uint32_t repetitions = 0,
          uint32_t ratio = 1, uint32_t offset = 0);
    // Disable a clock slave temporarily from the master
    virtual void disableClockSlave (CASIClockSlaveIF * slave);
    // Enable a clock slave on the master
    // (after it has been disabled with disableClockSlave)
    virtual void enableClockSlave (CASIClockSlaveIF * slave);

    // Functions with default implementations
    std::string getName() { return ""; }
    CASIInterfaceType getType() { return CASI_CLOCK_MASTER; }

    virtual void replaceClockSlave(CASIClockSlaveIF * toremove,
                        CASIClockSlaveIF *toadd) = 0;
    virtual void setPortInstanceName(const std::string& name) {
                        casiPortInstanceName = name; }
    virtual std::string getPortInstanceName() { return casiPortInstanceName; }
```

```
    virtual void setCASIOwner(CASIModuleIF * owner);
    virtual CASIModuleIF* getCASIOwner();
    virtual void enablePort(bool enable) { casiIsEnabled = enable; }
    virtual const bool isPortEnabled() { return casiIsEnabled; }

    virtual void setConnected(bool connected) {casiIsConnected = connected;}
    virtual const bool isConnected() {return casiIsConnected; }

    // Type definition for the communicate()/update() static scheduling
    typedef void (CASIClockSlaveIF::*CASIClockFuncType)();

    // Register the combinatorial communicate.
    template <class X> inline void registerCommunicate(
             CASIClockSlaveIF* component, void (X::*func)(),
             const std::string &name = "communicate")
         {
              registerRealCommunicate(component,
                       static_cast<CASIClockFuncType>(func), name);
         }
    // Register the update
    template <class X> inline void registerUpdate(CASIClockSlaveIF* component,
             void (X::*func)(), const std::string &name = "update")
        {
             registerRealUpdate(component,
                           static_cast<CASIClockFuncType>(func),name);
        }

     // add a edge from the source to the sink to represent dependency.
     // port -> func, func -> port, func -> func
     virtual void addDependency(CASIClockSlaveIF * component,
              const std::string& func, const CASIPortIF * port);
     virtual void addDependency(CASIClockSlaveIF * component,
              const CASIPortIF * port, const std::string& func);
     virtual void addDependency(CASIClockSlaveIF * component,
              const std::string& func1, const std::string & func2);
     virtual void schedule();

    // Prints a message to the standard output.
    virtual void pmessage( const std::string& msg,
              CASIMessageType type = CASI_MSG_INFO );
    virtual void pmessage( CASIMessageType type, const char *fmt, ... );

protected:
     void registerRealCommunicate(CASIClockSlaveIF* component,
            CASIClockFuncType func, const std::string &name);
     void registerRealUpdate(CASIClockSlaveIF* component,
            CASIClockFuncType func, const std::string &name);

    CASIClockSlaveIF *slave;
    std::vector<CASIClockSlaveIF*> slaves;
```



```
    CASIDependencyScheduler *pCommunicateScheduler;
    CASIDependencyScheduler *pUpdateScheduler;

    // casi function delegate component
    CASIDelegate *pClockedComponentsCommunicate;
    CASIDelegate *pClockedComponentsUpdate;
    // backup delegate
    CASIDelegate *pBackupClockedComponentsCommunicate;
    CASIDelegate *pBackupClockedComponentsUpdate;

private:
    std::string casiPortInstanceName;
    CASIModuleIF* casiOwner;
    bool casiIsEnabled;
    bool casiIsConnected;
};
```

This class can be used to generate clock dividers or other types of custom clock driving components.

The member functions are described in the following sections (some functions are described in the parent class, see the .h file for more detail):

- *CASIClockMaster::registerClockSlave()* on page 2-44
- *CASIClockMaster::unregisterClockSlave()* on page 2-45
- *CASIClockMaster::replaceClockSlave()* on page 2-46.
- *CASIClockMaster::pmessage()* on page 2-48
- *CASIClockMaster::disableClockSlave()* on page 2-46
- *CASIClockMaster::enableClockSlave()* on page 2-46
- *CASIClockMaster::registerRealCommunicate()* on page 2-46
- *CASIClockMaster::addDependency()* on page 2-48
- *CASIClockMaster::schedule()* on page 2-48
- *CASIPortIF::getName()* on page 2-38
- *CASIPortIF::getType()* on page 2-39
- *CASIPortIF::setCASIOwner()* on page 2-39
- *CASIPortIF::getCASIOwner()* on page 2-39
- *CASIPortIF::enablePort()* on page 2-39
- *CASIPortIF::isPortEnabled()* on page 2-39
- *CASIPortIF::enablePort()* on page 2-39
- *CASIPortIF::setConnected()* on page 2-40
- *CASIPortIF::isConnected()* on page 2-40.

2.4.2 CASIClockMaster::registerClockSlave()

When a clock slave is connected to its master, it can register itself using `registerClockSlave()` function for unconditional or conditional registration.

```
virtual void registerClockSlave(CASIClockSlaveIF *slave)=0

virtual void registerClockSlave(CASIClockSlaveIF *slave,
        CASIPhase phase, uint32_t repetitions = 0,
        uint32_t ratio = 1, uint32_t offset = 0) = 0
```

where:

`slave` is an instance of an object that implements the `CASIClockSlaveIF` interface (the base class for the `CASIClockSlave` class) and defines the basic cycle-accurate simulation execution unit. A CASI module can define zero, one or several clock slaves. Each clock slave represent the cycle-based behavior of the appropriate model part. The `communicate()` function implements the cycle-based communication with other components and `update()` implements the cycle-based updating of the internal resources of the component. A `CASIClockSlave` must be connected to an appropriate `CASIClockMaster` to receive the communicate or update calls.

`phase` specifies the phase to register the slave for:

- `CASI_PHASE_BOTH`
- `CASI_PHASE_COMMUNICATE`
- `CASI_PHASE_UPDATE`.

`repetitions` specifies the number of times the component is to be called. `0` means the component will be called indefinitely.

Note

This parameter has been deprecated and must not be used in new code. It has only been retained to support legacy code.

`ratio` specifies the clock ratio between the master clock and the local clock to be used. A ratio of `n` means that the component is be called once every `n` cycles. It must be greater than 0.

Note

This parameter has been deprecated and must not be used in new code. It has only been retained to support legacy code.

`offset` denotes the number of master clock cycles to elapse before the component is called for the first time. Default is 0.

Note

This parameter has been deprecated and must not be used in new code. It has only been retained to support legacy code.

Note

Conditional registration requires a more complex scheduling mechanism that can impact performance.

Conditional registration is only recommended for components that get called occasionally. In this case, the simulation performance is improved.

Note

The order in which the component `communicate()` functions are called cannot be specified by the standard `registerClockSlave()` function. Some systems, however, require that communication functions are called in a specific order to, for example, control a bus request and acknowledge sequence within the same cycle. For more information on specifying the order of `communicate()` functions, see Appendix A *Static Scheduling of Communication Functions*.

2.4.3 CASIClockMaster::unregisterClockSlave()

The `unregisterClockSlave()` function is used to remove the clock slave.

```
virtual void unregisterClockSlave(CASIClockSlaveIF *) =0
```

The unregistered clock slave no longer receive `communicate()` and `update()` calls in each cycle of this clock master.

This function is useful to optimize the simulation behavior by temporarily removing idle components from the simulation.

Note

This function cannot be used with static scheduling. See Appendix A *Static Scheduling of Communication Functions* for details on controlling clock functions.

2.4.4 CASIClockMaster::replaceClockSlave()

This function replaces the slave but keeps the conditional registration information.

```
virtual void replaceClockSlave(CASIClockSlaveIF *toremove,
            CASIClockSlaveIF *toadd) =0
```

The `replaceClockSlave()` function must be implemented for each clock master port to allow the replacement of clock slaves without the loss of conditional registration information.

2.4.5 CASIClockMaster::disableClockSlave()

This function disables a clock slave temporarily from the master. Use `enableClockSlave()` to re-enable the clock connection.

```
virtual void disableClockSlave (CASIClockSlaveIF * slave)
```

2.4.6 CASIClockMaster::enableClockSlave()

This function enables a clock slave on the master. It is used after the clock has been disabled with `disableClockSlave()`.

```
virtual void enableClockSlave (CASIClockSlaveIF * slave)
```

Note

This function is used with static scheduling instead of `unregisterClockSlave()`. See Appendix A *Static Scheduling of Communication Functions* for details on controlling clock functions.

2.4.7 CASIClockMaster::registerRealCommunicate()

Functions based on this template are used for static scheduling and to support combinatorial paths within the cycle-based simulation. `communicate` is used as the default name for the update function to enable use with legacy components.

The actual calling order for the specified communication function can be specified by using the `addDependency()` function (see *CASIClockMaster::addDependency()* on page 2-48).

```
template <class X> inline void registerCommunicate(CASIClockSlaveIF* component,
          void (X::*func)(), const std::string &name = "communicate")
{
   registerRealCommunicate(component,static_cast<CASIClockFuncType>(func),name);
}

template <class X> inline void registerCommunicate(CASIClockSlaveIF* component,
          void (X::*func)(), const std::string &name = "update")
{
   registerRealCommunicate(component,static_cast<CASIClockFuncType>(func),name);
}
```

Note

The order in which the component `communicate()` functions are called cannot be specified by the standard `registerClockSlave()` function. For more information on specifying the order of communication functions, see Appendix A *Static Scheduling of Communication Functions*.

2.4.8 CASIClockMaster::registerRealUpdate()

Functions based on this template are used for static scheduling and to support combinatorial paths within the cycle-based simulation. `update` is used as the default name for the update function to enable use with legacy components.

The actual calling order for the specified update function can be specified by using the `addDependency()` function (see *CASIClockMaster::addDependency()* on page 2-48).

```
template <class X> inline void registerUpdate(CASIClockSlaveIF* component,
             void (X::*func)(), const std::string &name = "update")
        {
             registerRealUpdate(component,
                           static_cast<CASIClockFuncType>(func),name);
        }
```

Note

The order in which the component `update()` functions are called cannot be specified by the standard `registerClockSlave()` function. For more information on specifying the order of functions, see Appendix A *Static Scheduling of Communication Functions*.

2.4.9 CASIClockMaster::addDependency()

These functions add an edge from the source to the sink to represent dependency. For example:

- To create a dependence edge between the communicate function `func()` and the port `port`, use the function:

  ```
  virtual void addDependency(CASIClockSlaveIF * component,
              const std::string& func, const CASIPortIF * port)
  ```

  The communication functions in the component connected to `port` will be called after the call to `func()`.

- To create a dependence edge between port and the communicate function func(), use the function:

  ```
  virtual void addDependency(CASIClockSlaveIF * component,
              const CASIPortIF * port, const std::string& func)
  ```

  The function `func()` will be called after the call to the communication functions in the component connected to `port`.

- To create a dependence edge between `func1()` and `func2()`, use the function:

  ```
  virtual void addDependency(CASIClockSlaveIF * component,
              const std::string& func1, const std::string & func2)
  ```

  The component communication function `func2()` will be called after `func1()`.

Note

If a port is used as a parameter in the call to `addDependency()`, the final dependency graph will be based on specified dependence edges in both the communication functions of the port owner and the dependence edges in the communication functions in the component that the port connects to. The communication functions in the connected component might also have dependencies on other functions in that component.

For more information on specifying the order of communication functions, see Appendix A *Static Scheduling of Communication Functions*.

2.4.10 CASIClockMaster::schedule()

This function schedules the calls to the attached clock ports.

```
virtual void schedule()
```

2.4.11 CASIClockMaster::pmessage()

This function prints a message to the standard output.


```
virtual void pmessage( const std::string& msg,
                CASIMessageType type = CASI_MSG_INFO )

virtual void pmessage( CASIMessageType type, const char *fmt, ... )
```

2.4.12 The CASIClockSlave class

This class provides a basic slave clock implementations. The slave clock interface is defined in `CASIClockSlaveIF`.

Note

`CASIModule` inherits from `CASIClockSlaveIF`, `CASIPortIF`, and `CASIModuleIF`.

The `communicate()` and `update()` functions must be implemented in the component class that inherits from `CASIClockSlaveIF`. The functions can, however, be implemented as a non-pure virtual function.

Example 2-7 CASIClockSlave definition

```
class CASIClockSlave : public CASIClockSlaveIF
{
public:
    virtual CAInterface * ObtainInterface( if_name_t ifName,
                        if_rev_t minRev, if_rev_t * actualRev);
    CASIClockSlave() : casiOwner(NULL), casiMaster(NULL), casiIsEnabled(true),
                        casiIsConnected(false) {}
    CASIClockSlave(CASIModuleIF* owner) : casiOwner(owner), casiMaster(NULL),
                        casiIsEnabled(true), casiIsConnected(false) {}
    virtual ~casi_clock_slave() {};

    // communicate and update must be implemented
    virtual void communicate() = 0;
    virtual void update() = 0;

    // Functions with default implementations.
    void casi_clocked(void) {};
    void casi_clocked (CASIPhase /* _phases */, uint32_t /* _numOfTimes */,
               uint32_t /* _ratio */, uint32_t /* _offset */) {}

    std::string getName() { return ""; }
    CASIInterfaceType getType() { return CASI_CLOCK_SLAVE; }

    virtual void connect(CASIClockMasterIF* master);
    virtual void disconnect(CASIClockMasterIF* master);
    virtual CASIClockMasterIF* getMaster() { return casiMaster; }

    virtual void setPortInstanceName(const std::string& name)
                            { casiPortInstanceName = name; }
    virtual std::string getPortInstanceName() { return casiPortInstanceName; }

    virtual void setCASIOwner(CASIModuleIF* owner) { casiOwner = owner; }
```



```
    virtual CASIModuleIF* getCASIOwner() { return casiOwner; }

    virtual void enablePort(bool enable) { casiIsEnabled = enable; }
    virtual const bool isPortEnabled() { return casiIsEnabled; }

    virtual void setConnected(bool connected) {casiIsConnected = connected;}
    virtual const bool isConnected() {return casiIsConnected; }

    virtual void pmessage( const std::string& msg,
            CASIMessageType type = CASI_MSG_INFO );
    virtual void pmessage( CASIMessageType type, const char *fmt, ... );
private:
    std::string casiPortInstanceName;
    CASIModuleIF* casiOwner;
    CASIClockMasterIF* casiMaster;
    bool casiIsEnabled;
    bool casiIsConnected;
};
```

2.5 The transaction interface classes

This section describes the classes that are used to implement the transaction master and transaction slave ports. This section has the following subsections:

2.5.1 Introduction to the transaction interface

The transaction interfaces are used for connections between components that consist of more than just a single signal. Memory transactions, for example, typically consist of the arbitration signals, the address and the data. Transactions usually involve bidirectional data transfer. That is, a read operation that passes an address from the master to the slave and returns the data value from the slave back to the master.

The transaction interfaces are used in pairs. A transaction-slave interface in one component can only be connected to a transaction-master interface in the other component. Multiple slaves can be connected to a single master. The master must decode the address to select the correct slave.

Transaction interfaces are not restricted to single reads and writes. Protocols can be implemented that allow burst, or block reads of multiple memory locations through a single transaction.

Transaction master ports are SystemC `sc_port` objects and use the corresponding transaction interface.

Master ports encapsulate the connection of a component to another component's slave port. The master ports provide standardized ways of accessing connected slave ports without having to know about the slave ports that are actually connected to them.

The slave ports behavior depends on the meaning and requirements of the resources corresponding to those slaves. The slave `read()` and `write()` functions must be customized to implement the behavior associated with the resource. However, because master ports only forward the requests to the connected slaves, master ports can have a generic implementation that can be directly used to instantiate master ports in the components. The `CASITransactionMaster` represents such a generic implementation.

2.5.2 The CASITransactionProperties structure

It is possible to define transaction properties for a more explicit definition of the semantics of a transaction. This simplifies implementing system level debugging features.

Example 2-8 CASITransactionProperties structure

```
struct CASITransactionProperties
{
    CASIVersion casiVersion;           // CASI_VERSION_2 or CASI_VERSION_3

    bool useMultiCycleInterface;      // use new drive/cancelTransaction functions ?
    uint8_t  addressBitwidth;         // address bitwidth for addressing of resources
    uint32_t mauBitwidth;             // minimal addressable unit
    uint32_t dataBitwidth;            // maximum bitwidth transferred in one cycle
    uint32_t dataBeats;               // Maximum data beats in a burst transaction (used only if
                                      // casiVersion >= 3.1).
    bool  isLittleEndian;             // alignment of MAUs
    bool  isOptional;                 // if true this port can be disabled
    bool  supportsAddressRegions;     // M/S can negotiate address mapping

    uint32_t numCtrlFields;           // # of ctrl elements used

    uint32_t numSlaveFlags;           // # of slave  flag elements being used
    uint32_t numMasterFlags;          // # of master flag elements being used
    uint32_t numTransactionSteps;     // # of transaction steps (maximum)
    uint32_t* validTimingTable;       // table providing transaction step of validity
    uint32_t protocolID;              // magic number of the protocol ID
                                       // The upper 16 bits = protocol implementation version
                                       // The lower 16 bits = actual protocol ID
    char protocolName[CASI_NAME_SIZE]  // The name of the protocol being used

    bool  supportsNotify;              // event based execution upon notify request
    bool  supportsBurst;               // burst transfer capability (true/false)
    bool  supportsSplit;               // split transfer capability (true/false)

    bool isAddrRegionForwarded         // true if addr region for this system port is actually
                                       // forwarded to a master port, false otherwise
```



```
    CASITransactionMasterIF * forwardAddrRegionToMasterPort;
                                    // master port of the same component to which
                                    // this salve port's addr region is forwarded

    CASITransactionDetails * details;  // Protocol extension, reserved for future functionality
};
```

The CASITransactionProperties structure can be used for both synchronous transactions (for example, using the read() and write() transaction member functions), as well as asynchronous transactions spanning over multiple clock cycles (for example, using the driveTransaction and cancelTransaction member functions).

The properties must be set from the constructor of the component, directly after creation of the port. The following functions are available in the transaction master and slave classes to access and modify the transaction properties information:

```
/* CASI 1.1: port properties */
virtual const CASITransactionProperties * getProperties();
virtual void setProperties(const CASITransactionProperties * prop);
```

The user calls these functions to set the properties of the transaction port and get the current properties. Port properties must not change after the reset stage.

2.5.3 CASITransactionProperties.validTimingTable

The valid timing table provides hints to, for example, a generic probe listening on a connection that uses the CASITransactionInfo multicycle interface.

Note

The sole purpose of the validTimingTable is for visualization tools. It is not intended to be used by the communicating parties (transaction master, transaction slave).

The valid timing table is a linear array of uint32_t. The information is recorded during the transaction step where certain items of the data structure become valid. Typically all items of the CASITransactionInfo data structure that characterize a transaction become valid at one point and stay valid from that point on (that might not always be the case).

There is also an assumption about which items are controlled by the master and which items are controlled by the transaction slave:

- For master controlled items the valid condition is:

  ```
  if ((validTimingTable[x] >= info.cts) && (info.status[info.cts] >=
  CASI_MASTER_READY)
  ```

- For slave controlled items the valid condition is:

  ```
  if ((validTimingTable[x] >= info.cts) && (info.status[info.cts] >=
  CASI_SLAVE_READY)
  ```

Usually it is only useful for a monitor to display valid information from `CASITransactionInfo`.

The index of the valid timing table corresponds to the items in the `CASITransactionInfo` data structure which again depends on the max number of beats, max number of transaction steps, max number of slave and master flags.

Example 2-9 Example of validTimingTable entries for an AXI port

```
validTimingTable[0] = AXI_STEP_ADDRESS; /* access */
validTimingTable[1] = AXI_STEP_ADDRESS; /* addr */
validTimingTable[2] = AXI_STEP_ADDRESS; /* dataSize */
validTimingTable[3] = AXI_STEP_ADDRESS; /* dataBeats */
validTimingTable[4] = AXI_STEP_DATA0; /* dataWr[0] */
validTimingTable[5] = AXI_STEP_DATA1; /* dataWr[1] */
...
validTimingTable[19] = AXI_STEP_DATA15; /* dataWr[15] */
validTimingTable[20] = AXI_STEP_DATA0; /* dataRd[0] */
...
validTimingTable[35] = AXI_STEP_DATA15; /* dataRd[15] */
validTimingTable[36] = AXI_STEP_ADDRESS; /* masterFlag[0]*/
...
validTimingTable[36 + n - 1] = AXI_STEP_ADDRESS; /* masterFlag[n] */
validTimingTable[36 + n - 1] = AXI_STEP_ADDRESS; /* slaveFlag[0] */
...
validTimingTable[36 + n + m - 1] = AXI_STEP_ADDRESS; /* slaveFlag[m] */
```

2.5.4 The CASITransactionIF interface

This class provides the interface for transaction-based communication. (See also *The CASITransactionSlave class* on page 2-66).

There are three access methods supported by the transaction interfaces:

Synchronous access functions

The `read()` and `write()` functions enable synchronous access between different components, where the `addr`, `value` and `ctrl` fields represent the address for the read or write, the value read or written and the control field for the read or write.

The `read()` and `write()` functions are expected to return immediately (in the same cycle where they were initiated), and return the status of the transaction. If they return `CASI_STATUS_OK`, the transaction finished successfully.

The read/write functions can implement also multi-cycled transactions. If, for example, in the first cycle they return `CASI_STATUS_WAIT`, then the initiating component calls the read/write function again in subsequent cycles, until it receives the `CASI_STATUS_OK` representing the end of this transaction.

The `readDbg()` and `writeDbg()` provide debug accesses and enable debuggers to, for example, read the desired information without advancing the simulation.

Asynchronous access functions

The asynchronous `readReq()` and `writeReq()` functions enable a communication model where the initiator master component provides a callback function pointer to the slave component.

When the slave component is ready to serve the transaction, it calls the callback function notifying the master that, for example, the data is ready.

Shared-memory asynchronous access functions

The shared-memory asynchronous functions provide a communication model where the initiating master component calls `driveTransaction()` providing to the slave a shared-memory data structure. This data structure is used throughout the transaction's life to communicate the information between the master and the slave components. After the first `driveTransaction()` function call, no other function calls are required through the transaction (unless a `cancelTransaction()` is called to cancel the respective transaction). The shared data structure is stored in CASITransactionInfo.

An optional notification callback from a slave to the connected master can be implemented through a `CASINotifyHandlerIF` object. The `notifyEvent()` function can be called by the slave to inform the master that the contents of the transaction info data structure has changed. This enables the master to react to the changes in the same cycle.

Note

For all of the three communication modes, the actual protocol define specific transaction parameters (such as the size of the data being transmitted and the meaning and size of the control field information) are defined by the specific protocol that the components use (for example AMBA AHB or AXI).

Note

Most of the functions in this class are pure virtual and must be implemented in either the `CASITransactionIF` class or the component slave port.

Example 2-10 CASITransactionIF class

```
class CASITransactionIF : public CASIPortIF, public sc_interface
{
public:
    CASITransactionIF() {}
    CASITransactionIF(std::string & name) {}
    virtual ~CASITransactionIF() {}
    virtual CASIStatus read(uint64_t addr, uint32_t* value, uint32_t* ctrl) = 0;

    virtual CASIStatus write(uint64_t addr, uint32_t* value, uint32_t* ctrl) = 0;
    virtual CASIStatus readDbg(uint64_t addr, uint32_t* value, uint32_t* ctrl) = 0;
    virtual CASIStatus writeDbg(uint64_t addr, uint32_t* value, uint32_t* ctrl) = 0;
    virtual CASIStatus readReq(uint64_t addr, uint32_t* value, uint32_t* ctrl,
                CASITransactionCallbackIF* callback) = 0;
    virtual CASIStatus writeReq(uint64_t addr, uint32_t* value, uint32_t* ctrl,
                CASITransactionCallbackIF* callback) = 0;

    // arbitration functions
    virtual CASIGrant requestAccess(uint64_t addr) = 0;
    virtual CASIGrant checkForGrant(uint64_t addr) = 0;

    // memory map functions
    virtual int getNumRegions() = 0;
    virtual void getAddressRegions(uint64_t* start, uint64_t* size, std::string * name) = 0;
    virtual void setAddressRegions(uint64_t* start, uint64_t* size, std::string * name) = 0;
    virtual CASIMemoryMapConstraints * getMappingConstraints() = 0;
```

```
    // shared memory transaction functions
    virtual void driveTransaction(CASITransactionInfo* info) = 0;
    virtual void cancelTransaction(CASITransactionInfo* info) = 0;
    virtual CASIStatus debugTransaction(CASITransactionInfo* info) = 0;

    // connection functions
    virtual void connect(CASITransactionMasterIF* iface) = 0;
    virtual void disconnect(CASITransactionMasterIF* iface) = 0;
    virtual CASITransactionMasterIF* getMaster() = 0;

    // Inspect transaction interface
    virtual const CASITransactionProperties * getProperties() = 0;
    virtual void setProperties(const CASITransactionProperties * prop) = 0;

    virtual std::string getName() = 0;
    virtual CASIInterfaceType getType() = 0;
    virtual std::string getPortInstanceName() = 0;
    virtual void setPortInstanceName(const std::string & name) = 0;
    virtual void setCASIOwner(CASIModuleIF * owner) = 0;
    virtual CASIModuleIF *getCASIOwner() = 0;
    virtual void enablePort(bool enable) = 0;
    virtual const bool isPortEnabled() = 0;
    virtual CASIStatus bypass(uint32_t msgSize, uint32_t* message, uint32_t rspSize,
                uint32_t* response) = 0;
};
```

The following subsections describe the functions:

- *CASITransactionIF::CASITransactionIF()* on page 2-59
- *CASITransactionIF::read()* on page 2-59
- *CASITransactionIF::write()* on page 2-60
- *CASITransactionIF::readDbg()* on page 2-60
- *CASITransactionIF::writeDbg()* on page 2-60
- *CASITransactionIF::readReq()* on page 2-60
- *CASITransactionIF::writeReq()* on page 2-61
- *CASITransactionIF::requestAccess()* on page 2-61
- *CASITransactionIF::checkForGrant()* on page 2-62
- *CASITransactionIF::getNumRegions()* on page 2-62
- *CASITransactionIF::getAddressRegions()* on page 2-63
- *CASITransactionIF::setAddressRegions()* on page 2-63
- *CASITransactionIF::getMappingConstraints()* on page 2-64
- *CASITransactionIF::driveTransaction()* on page 2-64
- *CASITransactionIF::cancelTransaction()* on page 2-64
- *CASITransactionIF::debugTransaction()* on page 2-64

- *CASITransactionIF::connect()* on page 2-65
- *CASITransactionIF::disconnect()* on page 2-65
- *CASITransactionIF::getMaster()* on page 2-65
- *CASITransactionIF::getProperties()* on page 2-65
- *CASITransactionIF::setProperties()* on page 2-65.
- *CASITransactionIF::bypass()* on page 2-65.

The following functions are inherited from `CASIPortIF`:

- *CASIPortIF::getName()* on page 2-38
- *CASIPortIF::getType()* on page 2-39
- *CASIPortIF::getPortInstanceName()* on page 2-39
- *CASIPortIF::setPortInstanceName()* on page 2-39
- *CASIPortIF::setCASIOwner()* on page 2-39
- *CASIPortIF::getCASIOwner()* on page 2-39
- *CASIPortIF::enablePort()* on page 2-39
- *CASIPortIF::isPortEnabled()* on page 2-39

CASITransactionIF::CASITransactionIF()

This function is the constructor for the interface.

```
CASITransactionIF() {}
CASITransactionIF(std::string & name) {}
```

CASITransactionIF::read()

This function performs synchronous read transaction operations.

```
virtual CASIStatus read(uint64_t addr, uint32_t* value, uint32_t* ctrl) =0
```

where:

`addr` is the transaction address.

`value` is an array of `uint32_t` for the value being read.

`ctrl` is an array of `uint32_t`, representing the control fields for the transaction. The actual meaning of the `ctrl` fields is protocol-dependent and is documented in the respective model or protocol documentation.

CASITransactionIF::write()

This function performs synchronous write transaction operations.

```
virtual CASIStatus write(uint64_t addr, uint32_t* value, uint32_t* ctrl) =0
```

where:

`addr` is the transaction address.

`value` is an array of `uint32_t`, representing the value being written.

`ctrl` is an array of `uint32_t`, representing the control fields for the transaction. The actual meaning of the `ctrl` fields is protocol-dependent and is documented in the respective model or protocol documentation.

CASITransactionIF::readDbg()

This function performs synchronous debug read transaction operations.

Similar to `read()`, except that the slave state must not change.

```
virtual CASIStatus readDbg(uint64_t addr, uint32_t* value, uint32_t* ctrl) =0
```

CASITransactionIF::writeDbg()

This function performs synchronous debug write transaction operations.

Similar to `write()`, except that the slave must not change state.

```
virtual CASIStatus writeDbg(uint64_t addr, uint32_t* value, uint32_t* ctrl) =0
```

CASITransactionIF::readReq()

This function performs asynchronous read transaction operations.

```
virtual CASIStatus readReq(uint64_t addr, uint32_t* value, uint32_t* ctrl,
        CASITransactionCallbackIF* callback) =0
```

where:

`addr` is the transaction address.

`value` is an array of `uint32_t`, representing the value being read.

`ctrl` is an array of `uint32_t`, representing the control fields for the transaction. The actual meaning of the `ctrl` fields is protocol-dependent and is documented in the respective model or protocol documentation.

`callback` is the callback object used by the slave to communicate the transaction progress to the master.

The definition of the callback function is:

```
virtual void CASITransactionCallbackIF::readAck (uint64_t address,
                    CASIStatus status) =0
```

CASITransactionIF::writeReq()

This function performs asynchronous write transaction operations.

```
virtual CASIStatus writeReq(uint64_t addr, uint32_t* value, uint32_t* ctrl,
        CASITransactionCallbackIF* callback) =0
```

where:

`addr` is the transaction address.

`value` is an array of `uint32_t`, representing the value being written.

`ctrl` is an array of `uint32_t`, representing the control fields for the transaction. The actual meaning of the `ctrl` fields is protocol-dependent and is documented in the respective model or protocol documentation.

`callback` is the callback object used by the slave to communicate the transaction progress to the master.

The definition of the callback function is:

```
virtual void CASITransactionCallbackIF::writeAck (uint64_t address,
                    CASIStatus status) =0
```

CASITransactionIF::requestAccess()

The arbitration functions offer support for one memory access per cycle with only a single cycle delay between the request and the actual transaction. For arbitrated bus accesses with one or more cycle delay:

1. Call `requestAccess()` for the desired location (address) in the first cycle
2. Call `checkForGrant()` to determine if the access was granted in the second cycle
3. In either the second or a following cycle, do the transaction if access was granted

Because there is no way to tell beforehand whether a transaction port will be connected to a simple transaction interface or a bus transaction interfaces, all transaction interfaces must implement `requestAccess()` and `checkForGrant()`.

The implementation for simple transaction interfaces, memory for example, can simply return CASI_GRANT_OK.

```
virtual CASIGrant requestAccess(uint64_t addr) =0
```

where:

addr is the request address.

The return value indicates the grant status:

CASI_GRANT_OK
: Bus was granted.

CASI_GRANT_DENIED
: Bus control was denied.

CASI_GRANT_ERROR
: An error occurred during bus arbitration.

CASI_GRANT_NOTSUPPORTED
: Bus operation not supported.

CASITransactionIF::checkForGrant()

This function checks whether access is still granted (see requestAccess()).

```
virtual CASIGrant checkForGrant(uint64_t addr) =0
```

where:

addr is the request address.

CASITransactionIF::getNumRegions()

This function returns the number of memory address regions supported by this interface.

The default behavior is to return 0 to indicate that the port covers the entire address space and indicate that no other connections can be made to the same master port.

```
virtual int getNumRegions() =0
```


CASITransactionIF::getAddressRegions()

This function returns the structure of the memory address regions supported by this interface. The address regions are returned as start value and block size in addition to a region name.

Note

More than one range can be specified. The number of expected regions is given by the function `getNumRegions()` and you must allocate the required memory for the parameters.

This function returns the default address mapping.

```
virtual void getAddressRegions(uint64_t* start, uint64_t* size,
        std::string * name) =0
```

where:

`start` is the memory address region start addresses array.

`size` is the memory address region sizes array.

`start` is the memory address region names array.

Note

Each parameter is an array. The size of the arrays is the value returned by the call to `getNumRegions()`.

CASITransactionIF::setAddressRegions()

This function sets the structure of the memory address regions for this interface.

```
virtual void setAddressRegions(uint64_t* start, uint64_t* size,
        std::string * name) =0
```

where:

`start` is the memory address region start addresses array.

`size` is the memory address region sizes array.

`start` is the memory address region names array.

Note

Each parameter is an array. The size of the arrays is the value returned by the call to `getNumRegions()`.

CASITransactionIF::getMappingConstraints()

This function returns the memory address regions constraints for this interface.

```
virtual CASIMemoryMapConstraints * getMappingConstraints() =0
```

CASITransactionIF::driveTransaction()

This function initiates a shared memory transaction.

The `driveTransaction()`, `cancelTransaction()`, and `debugTransaction()` functions are used for shared memory transactions. The interaction between the master and the slave is controlled through the shared `CASITransactionInfo`, therefore the transaction progress can only be observed by inspecting this shared data structure.

```
virtual void driveTransaction(CASITransactionInfo* info) =0
```

where:

`info` is the shared memory structure used to pass transaction status between master and slave.

CASITransactionIF::cancelTransaction()

This function cancels a shared memory transaction.

```
virtual void cancelTransaction(CASITransactionInfo* info) =0
```

where:

`info` is the shared memory structure used to pass transaction status between master and slave.

CASITransactionIF::debugTransaction()

This function initiates a shared memory transaction, but the slave state must not change.

```
virtual CASIStatus debugTransaction(CASITransactionInfo* info) =0
```

where:

`info` is the shared memory structure used to pass transaction status between master and slave.

CASITransactionIF::connect()

This function connects to a slave port.

The `connect()`, `disconnect()`, and `getMaster()` functions are used to specify the port connectivity. A slave port must notify its master if a relevant transaction event has happened through the `CASINotifyHandlerIF` object returned by the master's `getNotifyHandler()`. Thus the slave port must be able to identify its master.

```
virtual void connect(CASITransactionMasterIF* iface) =0
```

CASITransactionIF::disconnect()

This function disconnects a slave port.

```
virtual void disconnect(CASITransactionMasterIF* iface) =0
```

CASITransactionIF::getMaster()

This function retrieves the master connected to this port.

```
virtual CASITransactionMasterIF* getMaster() =0
```

CASITransactionIF::getProperties()

This function dynamically inspects the transaction interface custom properties and returns a description of the transaction properties of the port.

```
virtual const CASITransactionProperties * getProperties() =0
```

CASITransactionIF::setProperties()

This function dynamically sets the transaction interface custom properties.

```
virtual void setProperties(const CASITransactionProperties * prop) =0
```

where:

`prop` is a pointer to the new properties struct.

CASITransactionIF::bypass()

This function is used for debugging.

```
virtual CASIStatus bypass(uint32_t msgSize, uint32_t* message,
                          uint32_t rspSize, uint32_t* response) = 0;
```

CASITransactionIF::notifyEvent()

This function is the response to a call to `driveTransaction()`.

This function is called by the slave to inform the master that the transaction info structure has changed. This enables the master to process the changes in the same simulation cycle. The notify handler is typically set up in the reset phase to maximize efficiency.

Multiple calls to this function might occur while processing a single transaction.

```
virtual void notifyEvent(CASITransactionInfo* info) =0
```

where:

`info` is the shared memory structure used to pass transaction status between master and slave.

2.5.5 The CASITransactionSlave class

The `CASITransactionSlave` inherits the `CASITransactionIF` interface and can be used for any type of component that provides read and write functions to access its shared resources.

Note

Most of the functions in this class are pure virtual and must be implemented in the component slave port. Other functions however have a default behavior to simplify creations of components in common cases. See *The CASITransactionIF interface* on page 2-56 for the declaration of functions that are present in the parent class.

Example 2-11 CASITransactionSlave class

```
class CASITransactionSlave : public eslapi::CASITransactionIF
{
public:
   CASITransactionSlave(const std::string& name)
        : portName(name), casiPortInstanceName(""), casiOwner(NULL),
        casiIsEnabled(true), casiMaster(NULL)
        { memset(&casiProperties,0,sizeof(casiProperties)); }
   CASITransactionSlave(eslapi::CASIModuleIF * owner, const std::string& name)
        : portName(name), casiPortInstanceName(""), casiOwner(owner),
        casiIsEnabled(true), casiMaster(NULL)
        { memset(&casiProperties,0,sizeof(casiProperties)); }
   virtual ~CASITransactionSlave() {}
```



```
 // Functions to be implemented.
 // Synchronous access functions
virtual CASIStatus read(uint64_t addr, uint32_t* value, uint32_t* ctrl) = 0;
virtual CASIStatus write(uint64_t addr, uint32_t* value, uint32_t* ctrl) = 0;
virtual CASIStatus readDbg(uint64_t addr, uint32_t* value, uint32_t* ctrl);
virtual CASIStatus writeDbg(uint64_t addr, uint32_t* value, uint32_t* ctrl);

  // Asynchronous access functions
virtual CASIStatus readReq(uint64_t addr, uint32_t* value, uint32_t* ctrl,
         eslapi::CASITransactionCallbackIF* callback);
virtual CASIStatus writeReq(uint64_t addr, uint32_t* value, uint32_t* ctrl,
         eslapi::CASITransactionCallbackIF* callback);

 // Arbitration functions
virtual CASIGrant requestAccess(uint64_t addr);
virtual CASIGrant checkForGrant(uint64_t addr);

 // Memory map functions
virtual int getNumRegions();
virtual void getAddressRegions(uint64_t* start, uint64_t* size,
         std::string* name);
 // the address regions can be set from the outside
virtual void setAddressRegions(uint64_t* start, uint64_t* size,
         std::string* name);
virtual CASIMemoryMapConstraints* getMappingConstraints();
 // Shared-memory based asynchronous transaction functions
virtual void driveTransaction(eslapi::CASITransactionInfo* info);
virtual void cancelTransaction(eslapi::CASITransactionInfo* info);
virtual CASIStatus debugTransaction(eslapi::CASITransactionInfo* info);

 // Functions with default implementations.
 // CASIPortIF functions
std::string getName() { return portName; }
CASIInterfaceType getType() { return CASI_TRANSACTION_SLAVE; }

  // slaves now know their master
virtual void connect(CASITransactionMasterIF* master);
virtual void disconnect(CASITransactionMasterIF* master);
virtual eslapi::CASITransactionMasterIF* getMaster() { return casiMaster; }

 // instance names remembered by ports
virtual void setPortInstanceName(const std::string& name) {
         casiPortInstanceName = name; }
virtual std::string getPortInstanceName() { return casiPortInstanceName; }
 // every interface/port has an owner
virtual void setCASIOwner(eslapi::CASIModuleIF* owner) { casiOwner = owner; }
virtual eslapi::CASIModuleIF* getCASIOwner() { return casiOwner; }
  // ports can be enabled/disabled
virtual void enablePort(bool enable) { casiIsEnabled = enable; }
virtual const bool isPortEnabled() { return casiIsEnabled; }
```

```
    // message functions now also available in ports
    // Prints a message
   virtual void pmessage( const std::string& msg,
            CASIMessageType type = CASI_MSG_INFO );
   virtual void pmessage( CASIMessageType type, const char *fmt, ... );

    // port properties
   virtual const eslapi::CASITransactionProperties * getProperties()
            { return &casiProperties; }
   virtual void setProperties(const eslapi::CASITransactionProperties * prop)
            { casiProperties = *prop; }

    // auxiliary functions
   virtual CASIStatus bypass(uint32_t msgSize, uint32_t* message,
            uint32_t rspSize, uint32_t* response);

    // Return interface if requested
   virtual CAInterface * ObtainInterface(if_name_t ifName, if_rev_t minRev,
           if_rev_t *   actualRev)
   {
       if((strcmp(ifName,"eslapi.CASITransactionSlave2") == 0)
               && (minRev <= 0)){
           *actualRev = 0;
           return this;
      }
      return NULL;
   }

private:
    std::string portName;
    std::string casiPortInstanceName; // every port knows its instance name
    eslapi::CASIModuleIF * casiOwner; // every port knows its owner
    bool casiIsEnabled; // ports can be disabled
    eslapi::CASITransactionMasterIF* casiMaster; // slaves know their master

protected:
    eslapi::CASITransactionProperties casiProperties;
};
```


The read/write access functions are used for the actual transactions. There are three versions of these functions:

- The functions `read()` and `write()` are the standard methods for synchronous transactions.
- The `Dbg` versions of these functions are for debugger access only and no error checking is expected. It is, for example, possible to write to a ROM when doing so through the debugger.
- For asynchronous communication, the transaction slave interface provides the `readReq()` and `writeReq()` methods that have a callback interface as an additional parameter (see *The CASITransactionCallbackIF interface class* on page 2-73).

The function `getSlaves()` can be used to obtain a list of slaves connected in the form of an STL vector. The list is a read-only reference and therefore the return type is declared as const. See the transaction callback interface class for more details on asynchronous communication.

The parameters of the access functions are:

`address` is an unsigned 64bit value allowing large address sizes with only little computation overhead.

`parameter` is a pointer to a 32bit value that can be used for either a single 32bit value or an array of 32bit values. This allows for any size of data busses and is important for architectures with VLIW character.

`ctrl` can be used to qualify accesses. The data type (for example 8bit, 16bit, or 32bit access) can be passed here or any other value that provides additional information about the transaction.

Return values are:

`CASI_STATUS_OK` The memory access was successful.

`CASI_STATUS_WAIT`

The accessed memory is not available yet. The access must be retried in the following cycle.

`CASI_STATUS_NOACCESS`

The memory access is not permitted. For example, an attempt was made to write to a read-only memory.

`CASI_STATUS_NOMEMORY`

The memory accessed is not mapped to any component.

CASI_STATUS_NOTSUPPORTED

The connected slave did not implement the particular interface function

CASI_STATUS_ERROR

An undefined error has occurred during the memory access.

For arbitration purposes the TransactionSlave interface provides a requestAccess() and a checkForGrant() function. This makes it possible to have one memory access per cycle, with only a single cycle delay between the request and the actual transaction.

For arbitrated bus accesses with one or more cycle delay, the procedure would be:

- in the first cycle call requestAccess() for the desired location (address)
- in the second cycle call checkForGrant() to see if the access was granted
- in either the same or in the following cycle do the transaction if the access was granted

The function getNumRegions() returns the number of address regions that the slave port serves. The default behavior is to return 0. This return value indicates that the port covers the entire address space and does not allow any other connection to be made to the same master port. The user must overwrite the function if one or more address regions are specified.

The function getAddressRegions() can be overridden to return the appropriate address regions that the port wants to serve. The address regions are returned as start value and block size in addition to a region name.

Note

More than one range can be specified. The number of expected regions is given by the function getNumRegions() and it is expected that the user allocate the required memory for the parameters accordingly. This function simply returns the default address mapping. However the user can override those settings, unless the mapping constraints explicitly inhibit this.

From CASI version 1.1 and later, it is possible to set the memory map explicitly from outside. This enables centralized memory map management. The function setAddressRegions() is called to tell the slave port what address region it is mapped to.

The function getMappingConstraints() can be implemented to inform the simulation environment about supported memory regions:

- Some components can be used at any address.
- Others components might, however, be hard-coded for a specific address. The mapping constraints can reflect that to prevent the user from moving this component to another (not supported) memory region.

The CASIMemoryMapConstraints structure is defined as listed in Example 2-12:

Example 2-12 CASIMemoryMapConstraints structure

```
// CASI 1.1: new memory map info
struct CASIMemRegion
{
     uint64_t start;
     uint64_t size;
     std::string name;
};

class CASIMemoryMapConstraintsDetails;

struct CASIMemoryMapConstraints
{
     uint64_t minRegionSize;
     uint64_t minAddress;
     uint64_t maxAddress;
     uint32_t numSupportedRegions // min number of supported regions by this
                                  // slave at any given instant
     uint32_t maxNumSupportedRegions; // max number of supported regions by this

                                      // slave at any given point in time
     uint64_t alignmentBlockSize;  // Alignment requirement, the min block size
                                   // size where this slave's regions can be
                                    // mapped to (e.g., 1k for AHB)
     CASIMemoryMapConstraintsDetails *details; // Reserved

      //if this slave requires a FIXED set of regions, that CANNOT BE MOVED
     uint32_t numFixedRegions;   // if different then 0, this slave requires
                                 //a fixed set of regions that CANNOT BE MOVED
     CASIMemRegion *fixedRegionList; // only if numFixedRegions > 0
};
```

2.5.6 The CASITransactionMasterIF class

This interface facilitates master transaction port customization and communication:

- connect slaves to the master
- access the slaves using an address decoder in the master.

The predefined transaction master ports `sc_port<CASITransactionIF,0>` and `sc_port<CASITransactionIF,1>` inherit from this class and implement the functions defined in `CASITransactionIF` (see *The predefined sc_port< CASITransactionIF, 0> class* on page 2-87 and *The predefined sc_port<CASITransactionIF, 1> class* on page 2-83).

Example 2-13 CASITransactionMasterIF class

```
class CASITransactionMasterIF : public CASIPortIF
{
public:
    virtual ~CASITransactionMasterIF () {};

    // connection functions
    virtual void connect(CASITransactionIF* iface) = 0;
    virtual void disconnect(CASITransactionIF* iface) = 0;
    virtual const vector<CASITransactionIF*>& getSlaves() = 0;

    // customization functions
    virtual const CASITransactionProperties * getProperties() =0;
    virtual void setProperties (const CASITransactionProperties * prop) = 0;

    // notify handler support
    virtual void setNotifyHandler (CASINotifyHandlerIF * handler) = 0;
    virtual CASINotifyHandlerIF * getNotifyHandler (void) = 0;

    /// Common port functions
    virtual std::string getName() = 0;
    virtual CASIInterfaceType getType() = 0;
    virtual std::string getPortInstanceName() = 0;
    virtual void setPortInstanceName(const std::string & name) = 0;
    virtual void setCASIOwner(CASIModuleIF * owner) = 0;
    virtual CASIModuleIF * getCASIOwner() = 0;
    virtual void enablePort(bool enable) = 0;
    virtual const bool isPortEnabled() = 0;
};
```

The connect() function takes a CASITransactionIF pointer as a parameter and must register this pointer internally. It must check for the address regions the slave wants to occupy. This makes it possible to connect more than one slave to a master interface if the slaves do not occupy overlapping address regions.

The disconnect() function is used to remove the interface specified as the parameter from the list of connections.

Note

It is important to properly implement the disconnect() function as it is a requirement for debug probes that are dynamically connected and disconnected at runtime.

The function getSlaves() can be used to obtain a list of slaves connected in the form of an STL vector. The list is a read-only reference and therefore the return type is declared as const.

Note

A new notify handler that does not know the model internals might break proper transaction behavior. Wrap the original notify handler into the new handler and forward notifyEvent() calls to the original handler to preserve proper transaction behavior.

2.5.7 The CASITransactionCallbackIF interface class

A callback interface is provided for acknowledging read and write transactions and support asynchronous communication.

Example 2-14 CASITranactionCallbackIF class

```
class CASITransactionCallbackIF
{
public:
    virtual ~CASITransactionCallbackIF () {};
    /* Transaction Acknowledge functions */
    virtual void readAck(uint64_t address, CASIStatus status) = 0;
    virtual void writeAck(uint64_t address, CASIStatus status) = 0;
};
```

The functions readAck() and writeAck() are used to indicate the completion of a transaction. The transaction is identified by the address that has been requested. The status parameter contains the information whether the transaction has been completed successfully.

Figure 2-3 shows how asynchronous transactions are modeled:

- The master calls the `readReq()` function in the slave and passes all data necessary for the transaction and a callback interface pointer to the slave.
- The slave processes the transaction within one or more simulation cycles.
- Upon completion it calls the callback interface of the master to notify it of the transaction completion.
- The data is not passed again. The data containers initially passed by the master are used for storing the results.

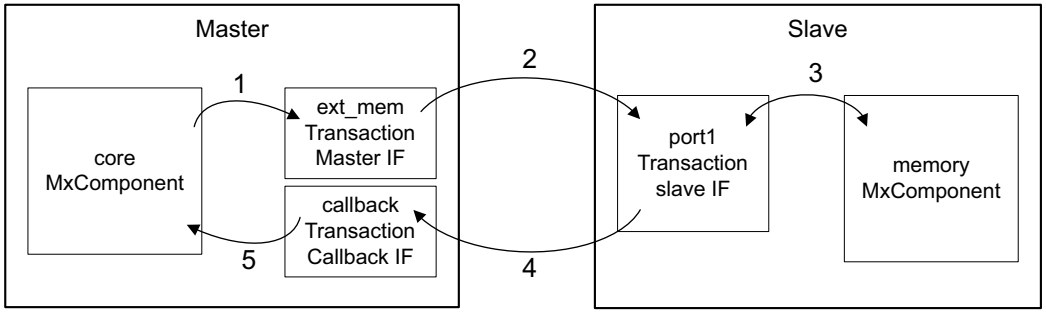

Figure 2-3 Asynchronous transactions using callbacks

CASITransactionCallbackIF::readAck()

This function is the answer for a `readReq()` call.

```
virtual void readAck(uint64_t address, CASIStatus status) =0
```

where:

`address` is the address of the `readReq` transaction call.

`status` is the status of the `readReq` transaction.

CASITransactionCallbackIF::writeAck()

This function is the answer for a `writeReq()` call.

```
virtual void writeAck(uint64_t address, CASIStatus status) =0
```

where:

`address` is the address of the `writeReq` transaction call.

`status` is the status of the `writeReq` transaction.

2.5.8 The multi-cycle transaction Interface

The CASI Transaction API supports transactions that have a life span of multiple cycles. The driveTransaction() method starts a transaction and `cancelTransaction()` aborts a single transaction. The communication between the master and the slave is based on shared resources during the course of a transaction. These shared resources are encapsulated in the structure `CASITransactionInfo`. Each transaction has a private instance of `CASITransactionInfo` that is instantiated by the initiating master port. The master port that calls the `driveTransaction()` function to initiate the transaction.

The following rules that ensure proper behavior while using the transaction shared variables from `CASITransactionInfo` for communication between a master and a slave port:

1. The transaction shared resources must only be owned exclusively by either the master or the slave port. The exception is the presence of a defined handover certificate where:
   - The current value of the shared resource indicates whether master or slave is allowed to do the next modification.
   - The ownership is controlled by another shared resource of the transaction that is exclusively owned by either a master or a slave port.
2. The transaction shared resources owned by a master are always modified during the communicate phase. Shared resources, however, that owned by a slave are always modified during the update phase.

The new transaction interface provides functions for initiating a transaction, canceling a transaction and performing a debug transaction. Each of these functions has a pointer to a data container of type `CASITransactionInfo` as a parameter:

```
virtual void driveTransaction(CASITransactionInfo* info) = 0
virtual void cancelTransaction(CASITransactionInfo* info) = 0
virtual CASIStatus debugTransaction(CASITransactionInfo* info) = 0
```

The functions must be defined in the slave and can be called by the master. The definition of the transaction info structure is described in detail in *The CASITransactionInfo structure* on page 2-76.

CASINotifyHandlerIF class

In addition to the `driveTransaction()`, `cancelTransaction()`, and `debugTransaction()` functions, an optional notification callback from a slave to the connected master was introduced in the CASI 1.1 API. The `notify` handler class is defined as shown in Example 2-15:

Example 2-15 CASINotifyHandlerIF class

```
class CASINotifyHandlerIF
{
    virtual ~CASINotifyHandlerIF () {};
    virtual void notifyEvent(CASITransactionInfo * info) = 0;
};
```

The `notifyEvent()` function can be called by the slave to inform the master that the contents of the transaction info data structure has changed. This gives the master the opportunity to react to the changes in the same cycle. ARM recommends setting the notify handler only once per simulation session to keep the simulation efficient. This can be done for example during the reset stage.

The *`info`* parameter is a pointer to the shared memory area used for communication.

2.5.9 The CASITransactionInfo structure

This structure (listed in Example 2-16) represents the data container throughout one transaction. It must be created by the master and passed to the slave.

For performance reasons, it is recommended to not allocate and delete the transaction info structure for each new transaction but instead reuse the existing structures for multiple transactions. If only one transaction can be processed at a time, only one single container is required for this interface for the entire simulation.

Example 2-16 CASITransactionInfo structure

```
struct CASITransactionInfo{
    /* Transaction Control */
  CASITransactionMasterIF* initiator; // initiator of the transaction
  uint32_t nts;                        // # of transaction steps to do
  uint32_t cts;                        // current transaction step
  CASITransactionStatus* status;      // status for each ctrl step
  CASINotify notify;                 // notify requested ? (CASI_NOTIFY_YES/NO)

    /* Pre-defined Transaction Elements */
```



```
  CASITransactionAccess access;       // data direction (read/write/rmw/…)
  uint64_t addr;                       // address of shared resource
  uint32_t dataSize;                   // dataSize in MAUs of shared resource
  uint32_t dataBeats;                  // # of data items for burst
  uint32_t* dataWr;                    // write data
  uint32_t* dataRd;                    // read data

    /* User-defined Transaction Elements */
  uint32_t* masterFlags;               // flags controlled by master port
  uint32_t* slaveFlags;                // flags controlled by slave port

public:
    // default constructor, no arguments
    CASITransactionInfo()
    // constructor with port as argument
    // => use this one to initialize the structure upon creation
    CASITransactionInfo(CASITransactionMasterIF *port)
    // Destructor always calls destruct function
    // only deletes private data, which was allocated in "initialize"
    ~CASITransactionInfo()
    // initialization function with port as argument
    // => use this one to initialize the structure e.g. from init function
    //    or at runtime
    void initialize(CASITransactionMasterIF *port);
    void initialize(const CASITransactionProperties *props);

    // The destruct function should not normally be called directly
    // delete should be called instead
    void destruct();

    // Use this to reset the structure before using it for a new transaction
    void reset();
    void setInitiator(CASITransactionMasterIF *port) { initiator = port};
    void clear(uint32_t undefVal = 0);
    bool saveData( CASIODataStream & data);
    bool restoreData( CASIIDataStream & data);

private:
        // Memory Management
       CASITransactionStatus * statusBuffer;
       uint32_t * dataWrBuffer;
       uint32_t * dataRdBuffer;
       uint32_t * masterFlagsBuffer;
       uint32_t * slaveFlagsBuffer;
       uint32_t numSteps, numData, numSFlags, numMFlags;

};
```

The CASITransactionInfo structure includes utility functions for allocation, initialization, and destruction. It also provides a reset function to be used when reusing a single container for multiple consecutive, but non-overlapping, transactions. By invalidating the master flags, all other data becomes invalid (this saves time compared to clearing the entire contents of the container).

If you are using the CASITransactionInfo(), driveTransaction(), cancelTransaction(), and debugTransaction() functions, the field casiVersion in CASITransactionProperties must be set to CASI_VERSION_3 and the field useMultiCycleInterface in CASITransactionProperties has to be set to true.

The following sections describe the contents of the CASITransactionInfo structure:

- *CASITransactionInfo.initiatingMaster*
- *CASITransactionInfo.nts*
- *CASITransactionInfo.cts* on page 2-79
- *CASITransactionInfo.status* on page 2-79
- *CASITransactionInfo.notify* on page 2-81
- *CASITransactionInfo.access* on page 2-81
- *CASITransactionInfo.addr* on page 2-81
- *CASITransactionInfo.dataSize* on page 2-82
- *CASITransactionInfo.dataBeats* on page 2-82
- *CASITransactionInfo.dataWr* on page 2-82
- *CASITransactionInfo.dataRd* on page 2-82
- *CASITransactionInfo.masterFlags* on page 2-83
- *CASITransactionInfo.slaveFlags* on page 2-83.

CASITransactionInfo.initiatingMaster

The pointer to the initiating master is providing access to the master port that is the controlling master that instantiated the transaction info data structure and initiated the transaction using the driveTransaction() function.

CASITransactionInfo.nts

The total *Number of Transaction Steps* (nts) indicates how many control steps are required for this overall transaction. This entry is controlled by the master port.

A transaction is considered completed if the simulation is in the last transaction step, and the status array contains CASI_SLAVE_READY for the last transaction step. This can be represented by the equation:

```
((cts == (nts - 1) && (status[cts] >= CASI_SLAVE_READY)).
```


CASITransactionInfo.cts

The *Current Transaction Step* (cts) is a positive integer value that indicates the transaction step that the described transaction is currently in. This data element is owned by the master port and is always initialized to 0 (0 indicates the first transaction step).

CASITransactionInfo.status

The status array describes the status of each transaction step. The maximum number of transaction steps is defined as part of the Transaction Port Properties. The status is a predefined enumeration type:

Example 2-17 CASITransactionStatus enumeration

```
enum CASITransactionStatus
{
    //status reported by the master
   CASI_MASTER_WAIT = 0,  // transaction step does not contain valid information
   CASI_MASTER_READY,       // transaction step contains valid info by master
        //status reported by the slave
   CASI_SLAVE_WAIT,         // transaction step does not contain valid response
   CASI_SLAVE_READY,        // transaction step contains valid response by slave
   CASI_SLAVE_READY_CANCEL, // slave acknowledges "cancelTransaction(…)"
   CASI_SLAVE_READY_SPLIT,  // slave breaks transaction, can be cont'd by master
   CASI_SLAVE_READY_RETRY,  // slave asks for restart of same transaction
   CASI_SLAVE_READY_ERROR   // slave reports transaction error
}
```

The status fields are under the control of both the connected master and slave. However, there is a defined hand-over from master to slave after CASI_MASTER_READY has been set by the master for the current transaction step:

- Initially, the transaction step status is initialized to CASI_MASTER_WAIT, indicating that the master has not provided any transaction specific information for this transaction step. This is a clear indication that the slave is not allowed to proceed in the current transaction step (master controlled wait).
- After the master has incremented the status to CASI_MASTER_READY, the master is responsible that all the information passed from master to slave in this transaction step is correctly set. From now on, the status for the current transaction step is under the sole control of the transaction slave.

The CASI_SLAVE_WAIT indicates to the master that it is not allowed to proceed with this transaction step (slave controlled wait) and that the information to be passed from the slave to the master is not available yet. Once the slave has incremented the status to CASI_SLAVE_READY or above the slave is responsible that all the information it has to pass back to the master is set.

This transaction control framework requires that the transaction's shared variables (used during a certain transaction step and owned by the master) must be valid on CASI_MASTER_READY. The shared variables owned by the slave, however, have to be valid on CASI_SLAVE_READY or above. This means that there is only a single point in time during a transaction step where data becomes valid. For instance, if a slave communicates two signals to the master and these become valid in two different cycles, they must be transferred in two separate transaction steps.

The transaction slave has the option to further qualify the CASI_SLAVE_READY to inform the master of further transaction control.

CASI_SLAVE_READY

Terminate current transaction step

CASI_SLAVE_READY_CANCEL

Terminate current transaction step and terminate the whole transaction (cancel has been accepted by slave)

CASI_SLAVE_READY_SPLIT

Terminate current transaction step and stop the transaction (close connection), however continue the transaction at a later point in time.

CASI_SLAVE_READY_RETRY

Terminate current transaction step and stop the transaction (close connection), however restart the transaction from scratch at a later point in time.

CASI_SLAVE_READY_ERROR

A transaction level control error occurred. Terminate current transaction step and the whole transaction (failure)

CASITransactionInfo.notify

A slave that modifies any shared resource, while this flag is set is required to call the `notifyEvent()` method implemented by the master port `NotifyEventHandler`. The notification request flag is a Boolean variable that is owned by the master port of a forwarding slave. This is an advanced option to allow synchronization between multiple slaves that have chained or dependant responses even though their order of execution is not defined.

CASITransactionInfo.access

The access data element of the `CASITransactionInfo` structure is an enumeration type value that defines the different possible accesses. This entry is owned by the master. The master determines the access encapsulated in this data structure. One of the main side effects of this parameter is deciding whether the data is passed by `dataRd` or `dataWr`.

Example 2-18 CASITransaction Access enumeration

```
enum CASITransactionAccess
{
   CASI_ACCESS_IDLE = 0  // dummy access
   CASI_ACCESS_READ = 1, // read access  (data is provided by Slave)
   CASI_ACCESS_WRITE,    // write access (data is provided by Master)
   CASI_ACCESS_RMW,      // atomic sequence of read followed by write
   CASI_ACCESS_SPLIT,    // transaction will be stopped and continued
                          // as another transaction
   CASI_ACCESS_USER      // protocol specific access type; see resp. masterFlag
}
```

The value `CASI_ACCESS_USER` can be used if other protocol-specific access codes are required. In this case more specific information has to be provided by a masterFlags entry.

CASITransactionInfo.addr

The address data element of the `CASITransactionInfo` structure is an unsigned 64 bit value that specifies the address of the resource (resource identification) to be accessed during this transaction. The number of least significant bytes being used for the transaction is limited by the API to 64 and is specified by the transaction property named `address_bitwidth`. It is the responsibility of the master to pad all unused MSBs with 0. The `mau_size` transaction property defines, the minimal addressable unit, dictating the number of bits that are specified between address increments.

CASITransactionInfo.dataSize

The size of data element of the CASITransactionInfo structure is an unsigned eight-bit value that specifies the size of a single data element to be transferred based on the unit mau_size defined in the transaction properties.

For example, if mau_size is 4 and dataSize is 3, the number of bits valid in data is 12.

This element of the structure is owned by the master.

CASITransactionInfo.dataBeats

The number of data beats is an unsigned eight-bit value that specifies the number of data transfers to be executed during this transaction. This capability is also referred to as burst transaction. In case the transaction property indicates that supportsBurst is false, this entry is irrelevant and the total number of MAUs is solely defined by dataSize.

Every data beat uses its own transaction step because the data becomes valid during different cycles and might have an arbitrary number of wait-states in between.

Note

dataRd and dataWr are reused during each beat. The data for the previous beat is lost. Of course this data can be stored and traced outside of the TransactionInfo structure if necessary. This element of the structure is owned by the master.

CASITransactionInfo.dataWr

The data write element of the CASITransactionInfo structure is a pointer to an array of 32-bit unsigned values that are set by the master during write accesses (or during the write of a read modify write sequence). This data is transferred to the slave to modify a resource identified by the address given in addr.

CASITransactionInfo.dataRd

The data read element of the CASITransactionInfo structure is a pointer to an array of 32-bit unsigned values that are set by the slave during read accesses (or during the read of a read modify write sequence). This data is transferred to the master that has requested to read the resource identified by the address given in addr.

CASITransactionInfo.masterFlags

The masterFlags element is a pointer to an array of user defined unsigned 32-bit data elements that are owned by the master. The number of entries in this array is defined as part of the transaction properties numMasterFlags. The content of this array is user-defined and therefore only the raw numbers can be displayed by generic monitors.

CASITransactionInfo.slaveFlags

The slaveFlags element is a pointer to an array of user defined unsigned 32-bit data elements that are owned by the slave. The number of entries in this array is defined as part of the transaction properties numSlaveFlags. The content of this array is user-defined and therefore only the raw numbers can be displayed by generic monitors.

2.5.10 The predefined sc_port<CASITransactionIF, 1> class

This class is a predefined port class that is used for all transaction master ports that connect to a single slave. The templated constructors are listed in Example 2-19:

Example 2-19 sc_port<CASITransactionIF, 1> class

```
class sc_port<eslapi::CASITransactionIF, 1> : public eslapi::CASITransactionMasterIF
{
public:
    sc_port<eslapi::CASITransactionIF,1> (eslapi::CASIModuleIF * _owner, const std::string& name,
         eslapi::CASITransactionProperties *prop=NULL)
         : slave(NULL), portName(name), casiPortInstanceName(""),casiOwner(_owner), casiIsEnabled(true)
         { memset(&casiProperties,0,sizeof(casiProperties)); if(prop!=NULL) casiProperties = *prop; }

    sc_port<eslapi::CASITransactionIF,1>(const std::string& name,
           eslapi::CASITransactionProperties *prop=NULL)
          : slave(NULL), portName(name), casiPortInstanceName(""), casiOwner(NULL), casiIsEnabled(true)
          { memset(&casiProperties,0,sizeof(casiProperties)); if(prop!=NULL) casiProperties = *prop; }

   sc_port<eslapi::CASITransactionIF,1>() : slave(NULL), portName("default name 0"),
        casiPortInstanceName(""), casiOwner(NULL), casiIsEnabled(true)
      {memset(&casiProperties,0,sizeof(casiProperties));}

  explicit sc_port<eslapi::CASITransactionIF,1>(const char *name)
     : slave(NULL), portName(name), casiPortInstanceName(""), casiOwner(NULL), casiIsEnabled(true)
     { memset(&casiProperties,0,sizeof(casiProperties)); }

  explicit sc_port<eslapi::CASITransactionIF,1>( eslapi::CASITransactionIF& interface_ )
     : slave(NULL), portName("default name 0"), casiPortInstanceName(""),
     casiOwner(NULL),  casiIsEnabled(true)
     { memset(&casiProperties,0,sizeof(casiProperties));}
```



```
sc_port<eslapi::CASITransactionIF,1>( const char* name_, eslapi::CASITransactionIF& interface_ )
      : slave(NULL), portName(name_), casiPortInstanceName(""), casiOwner(NULL), casiIsEnabled(true)
      { memset(&casiProperties,0,sizeof(casiProperties));}

sc_port<eslapi::CASITransactionIF,1>( sc_port<eslapi::CASITransactionIF,1>& parent_ )
      : CASITransactionMasterIF(), slave(NULL), portName("default name 0"), casiPortInstanceName(""),
      casiOwner(NULL), casiIsEnabled(true)
      { memset(&casiProperties,0,sizeof(casiProperties));}
sc_port<eslapi::CASITransactionIF,1>( const char* name_,
      sc_port<eslapi::CASITransactionIF,1>& parent_ )
      : slave(NULL), portName(name_), casiPortInstanceName(""), casiOwner(NULL), casiIsEnabled(true)
      { memset(&casiProperties,0,sizeof(casiProperties));}

virtual ~sc_port<CASIGenericTransactionIFCASITransactionIF,1>() {}

  // Configuration functions
std::string getName() { return portName; }
eslapi::CASIInterfaceType getType() { return eslapi::CASI_TRANSACTION_MASTER; }
virtual const eslapi::CASITransactionProperties * getProperties()
       { return &eslapi::casiProperties; }
virtual void setProperties(const eslapi::CASITransactionProperties * prop)
       { eslapi::casiProperties = *prop; }
bool supportsAddressRegions()     {return eslapi::casiProperties.supportsAddressRegions; }
virtual void setPortInstanceName(const std::string& name) { casiPortInstanceName = name; }
virtual std::string getPortInstanceName() { return casiPortInstanceName; }
virtual void setCASIOwner(eslapi::CASIModuleIF* owner) { casiOwner = owner; }
virtual eslapi::CASIModuleIF* getCASIOwner() { return casiOwner; }
virtual void enablePort(bool enable) { casiIsEnabled = enable; }
virtual const bool isPortEnabled() { return casiIsEnabled; }
virtual void setNotifyHandler(CASINotifyHandlerIF *handler) { notifyHandler=handler; }
virtual eslapi::CASINotifyHandlerIF * getNotifyHandler(void) { return notifyHandler; }
  // Prints a message
virtual void pmessage( const std::string& msg,
     eslapi::CASIMessageType type = eslapi::CASI_MSG_INFO );
virtual void pmessage( eslapi::CASIMessageType type, const char *fmt, ... );

  // Connectivity functions
virtual void connect(eslapi::CASITransactionIF* iface);
virtual void disconnect(eslapi::CASITransactionIF* iface);
virtual void replace(eslapi::CASITransactionIF* old_iface, eslapi::CASITransactionIF* new_iface);
virtual const std::vector<eslapi::CASITransactionIF*>& getSlaves();

  // Behavioral functions
  // These are convenience functions which will operate on the unique slave.
  // Synchronous read transaction operation.
eslapi::CASIStatus read(uint64_t addr, uint32_t* value, uint32_t* ctrl)
   { return slave->read(addr, value, ctrl); }
  // Synchronous write transaction operation.
eslapi::CASIStatus write(uint64_t addr, uint32_t* value, uint32_t* ctrl)
```



```
   { return slave->write(addr, value, ctrl); }
  // Synchronous debug read transaction operation.
eslapi::CASIStatus readDbg(uint64_t addr, uint32_t* value, uint32_t* ctrl)
   { return slave->readDbg(addr, value, ctrl); }
  // Synchronous debug write transaction operation.
eslapi::CASIStatus writeDbg(uint64_t addr, uint32_t* value, uint32_t* ctrl)
   { return slave->writeDbg(addr, value, ctrl); }
  // Aynchronous read transaction operation.
eslapi::CASIStatus readReq( uint64_t addr, uint32_t* value, uint32_t* ctrl,
   eslapi::CASITransactionCallbackIF* callback )
  { return slave->readReq( addr, value, ctrl, callback ); }
  // Asynchronous write transaction operation.
eslapi::CASIStatus writeReq( uint64_t addr, uint32_t* value, uint32_t* ctrl,
   eslapi::CASITransactionCallbackIF* callback )
   { return slave->writeReq( addr, value, ctrl, callback ); }

  // Requests bus access
eslapi::CASIGrant requestAccess(uint64_t addr) { return slave->requestAccess( addr ); }
  // Checks whether access is still granted.
eslapi::CASIGrant checkForGrant(uint64_t addr) {return slave->checkForGrant( addr ); }

  // Returns the number of memory address regions supported.
int getNumRegions()
      {
          if(slave != NULL) return slave->getNumRegions();
          return 0;
      }
  // Returns the structure of the memory address regions supported.
void getAddressRegions(uint64_t* start, uint64_t* size, std::string* name)
   { slave->getAddressRegions(start, size, name); }
void setAddressRegions(uint64_t* start, uint64_t* size, std::string* name)
   { slave->setAddressRegions(start, size, name); }
  // Initiate a shared memory transaction

virtual eslapi::CASIMemoryMapConstraints* getMappingConstraints()
   { return slave->getMappingConstraints(); }

virtual void driveTransaction(eslapi::CASITransactionInfo* info)
   { slave->driveTransaction(info); }
  // Cancels a shared memory transaction
virtual void cancelTransaction(eslapi::CASITransactionInfo* info)
   { slave->cancelTransaction(info); }
  // Initiate a shared memory transaction
virtual eslapi::CASIStatus debugTransaction(eslapi::CASITransactionInfo* info)
          { return slave->debugTransaction(info); }
  // convenience connection operator
void operator () ( eslapi::CASITransactionIF& interface_ );
virtual const char* kind() const { return kind_string; }

  // Return interface if requested
```



```
  virtual eslapi::CAInterface * ObtainInterface(if_name_t ifName, if_rev_t minRev,
         if_rev_t *   actualRev)
         {
             if((strcmp(ifName,"eslapi.sc_port<CASITransactionIF,1>2") == 0) && (minRev <= 0))
             {
                  *actualRev = 0;
                   return this;
             }
             return NULL;
         }


  typedef sc_port_base  base_type;
  typedef sc_port_b<eslapi::CASITransactionIF> this_type;

private:
  void bind( eslapi::CASITransactionIF& interface_ );
  //void operator() ( eslapi::CASITransactionIF& interface_ );
  void bind( this_type& parent_ );
  void operator() ( this_type& parent_ );
    virtual sc_interface* get_interface() {return NULL;};
    virtual const sc_interface* get_interface() const {return NULL;};
    static const char* const kind_string;

protected:
    eslapi::CASITransactionIF *slave;
    std::string portName;
    std::string casiPortInstanceName;    // every port knows its instance name
    eslapi::CASIModuleIF * casiOwner;    // every port knows its owner
    bool casiIsEnabled;                  // ports can be disabled
    eslapi::CASINotifyHandlerIF *notifyHandler; // the notify handler of this master transaction port
    std::vector<eslapi::CASITransactionIF*> slaves;
    eslapi::CASITransactionProperties casiProperties;
};
```

Only one transaction slave can be connected to the transaction master port. To connect multiple transaction slaves to a master port use:

```
sc_port<eslapi::CASITransactionIF,0>
```

The `read()`, `write()`, `readDbg()`, `writeDbg()`, and related functions forward the requests to the slave connected to this master.

See *The CASITransactionIF interface* on page 2-56 for a description of the member functions.

2.5.11 The predefined sc_port< CASITransactionIF, 0> class

This class is a predefined port class that is used for transaction master ports that behave like buses and support connection to more than one slave. The second argument of the template (that is, the value of 0 here) specifies that an unlimited number of slave ports can be connected to this master port.

Example 2-20 sc_port<CASITransactionIF,0> class

```
class sc_port<eslapi::CASITransactionIF,0> :     public eslapi::CASITransactionMasterIF
{
public:
    sc_port<eslapi::CASITransactionIF,0>(eslapi::CASIModuleIF * owner, const std::string& name,
           uint64_t blocksize, uint64_t memsize,
           eslapi::CASITransactionProperties *prop=NULL);
    sc_port<eslapi::CASITransactionIF,0>(const std::string& name, uint64_t blocksize,
           uint64_t memsize, eslapi::CASITransactionProperties *prop=NULL);
    sc_port<eslapi::CASITransactionIF,0>(const std::string& name) : portName(name),
           casiPortInstanceName(""), casiOwner(NULL), casiIsEnabled(true), slave(NULL)
           { memset(&casiProperties,0,sizeof(casiProperties)); }
    sc_port<eslapi::CASITransactionIF,0>() : portName("default name 0"),
           casiPortInstanceName(""), casiOwner(NULL), casiIsEnabled(true), slave(NULL)
           {memset(&casiProperties,0,sizeof(casiProperties));}

    explicit sc_port<eslapi::CASITransactionIF,0>(const char *name): portName(name),
           casiPortInstanceName(""), casiOwner(NULL), casiIsEnabled(true), slave(NULL)
           { memset(&casiProperties,0,sizeof(casiProperties)); }

    explicit sc_port<eslapi::CASITransactionIF,0>( eslapi::CASITransactionIF& interface_ )
          : portName("default name 0"), casiPortInstanceName(""), casiOwner(NULL),
          casiIsEnabled(true), slave(NULL)
          { memset(&casiProperties,0,sizeof(casiProperties));}

    sc_port<eslapi::CASITransactionIF,0>( const char* name_,
          eslapi::CASITransactionIF& interface_ ) : portName(name_), casiPortInstanceName(""),
          casiOwner(NULL), casiIsEnabled(true), slave(NULL)
          { memset(&casiProperties,0,sizeof(casiProperties));}
    sc_port<eslapi::CASITransactionIF,0>( sc_port<eslapi::CASITransactionIF,0>& parent_ )
          : CASITransactionMasterIF(), portName("default name 0"), casiPortInstanceName(""),
          casiOwner(NULL), casiIsEnabled(true), slave(NULL)
          { memset(&casiProperties,0,sizeof(casiProperties));}
    sc_port<eslapi::CASITransactionIF,0>( const char* name_,
          sc_port<eslapi::CASITransactionIF,0>& parent_ )
          : portName(name_), casiPortInstanceName(""), casiOwner(NULL),
          casiIsEnabled(true), slave(NULL)
          { memset(&casiProperties,0,sizeof(casiProperties));}
    virtual ~sc_port<CASITransactionIF,0>();
```



```
// Configuration functions
std::string getName() { return portName; }
eslapi::CASIInterfaceType getType() { return eslapi::CASI_TRANSACTION_MASTER; }
virtual const eslapi::CASITransactionProperties * getProperties() { return &casiProperties; }
virtual void setProperties(const eslapi::CASITransactionProperties * prop)
        { casiProperties = *prop; }
bool supportsAddressRegions() { return casiProperties.supportsAddressRegions; }
virtual void setPortInstanceName(const std::string& name) { casiPortInstanceName = name; }
virtual std::string getPortInstanceName()   { return casiPortInstanceName; }
virtual void setCASIOwner(eslapi::CASIModuleIF* owner)  { casiOwner = owner; }
virtual eslapi::CASIModuleIF* getCASIOwner() { return casiOwner; }
virtual void enablePort(bool enable) { casiIsEnabled = enable; }
virtual const bool isPortEnabled() { return casiIsEnabled; }
virtual void setNotifyHandler(eslapi::CASINotifyHandlerIF *handler) { notifyHandler=handler; }
virtual eslapi::CASINotifyHandlerIF * getNotifyHandler(void) { return notifyHandler; }

// Prints a message
virtual void pmessage( const std::string& msg,
        eslapi::CASIMessageType type = eslapi::CASI_MSG_INFO );
virtual void pmessage(eslapi::CASIMessageType type, const char *fmt, ... );

// Connectivity functions
virtual void connect(eslapi::CASITransactionIF* iface);
virtual void disconnect(eslapi::CASITransactionIF* iface);
virtual void replace(eslapi::CASITransactionIF* old_iface, eslapi::CASITransactionIF* new_iface);
virtual const std::vector<eslapi::CASITransactionIF*>& getSlaves();
eslapi::CASITransactionIF *getSlave() { return slave; }

// Behavioral functions
// These will be redirected to the appropriate slave based on the address decoding algorithm.
// Synchronous read transaction operation.
virtual eslapi::CASIStatus read(uint64_t addr, uint32_t* value,
      uint32_t* ctrl);
// Synchronous write transaction operation.
virtual eslapi::CASIStatus write(uint64_t addr, uint32_t* value,
     uint32_t* ctrl);
// Synchronous debug read transaction operation.
virtual eslapi::CASIStatus readDbg(uint64_t addr, uint32_t* value,
     uint32_t* ctrl);
// Synchronous debug write transaction operation.
virtual eslapi::CASIStatus writeDbg(uint64_t addr, uint32_t* value,
     uint32_t* ctrl);
// Aynchronous read transaction operation.
virtual eslapi::CASIStatus readReq(uint64_t addr, uint32_t * value,
     uint32_t* ctrl, eslapi::CASITransactionCallbackIF * callback)
     { return eslapi::CASI_STATUS_NOTSUPPORTED; }
// Asynchronous write transaction operation.
virtual eslapi::CASIStatus writeReq(uint64_t addr, uint32_t * value,
     uint32_t* ctrl, eslapi::CASITransactionCallbackIF * callback)
     { return eslapi::CASI_STATUS_NOTSUPPORTED; }
```



```
    // Requests bus access
    virtual eslapi::CASIGrant requestAccess( uint64_t addr );
    // Checks whether access is still granted.
    virtual eslapi::CASIGrant checkForGrant( uint64_t addr );
    // Initiate a shared memory transaction
    virtual void driveTransaction(eslapi::CASITransactionInfo* info);
    // Cancels a shared memory transaction
    virtual void cancelTransaction(eslapi::CASITransactionInfo* info);
    // Initiate a shared memory transaction
    virtual eslapi::CASIStatus debugTransaction(eslapi::CASITransactionInfo* info);

    // convenience connection operator
    void operator () ( CASITransactionIF& interface_ );
    int size() const;
    eslapi::CASITransactionIF* operator -> ();
    const eslapi::CASITransactionIF* operator -> () const;
    eslapi::CASITransactionIF* operator [] ( int index_ );
    const eslapi::CASITransactionIF* operator [] ( int index_ ) const;

    virtual const char* kind() const { return kind_string; }

    // SC typedefs
    typedef sc_port_base base_type;
    typedef sc_port_b<eslapi::CASITransactionIF> this_type;

private:
    bool modifyMemoryMaps(eslapi::CASIMMI *cammi, eslapi::CASITransactionIF* slaveIface,
         bool addNotRemoveMapping);
    void bind( eslapi::CASITransactionIF& interface_ );
    void bind( this_type& parent_ );
    void operator () ( this_type& parent_ );
    virtual sc_interface* get_interface() {return NULL;};
    virtual const sc_interface* get_interface() const {return NULL;};
    static const char* const kind_string;

protected:
    uint64_t minBlockSize;
    std::string portName;
    std::string casiPortInstanceName;       // every port knows its instance name
    eslapi::CASIModuleIF* casiOwner;        // every port knows its owner
    bool casiIsEnabled;                     // ports can be disabled

    // the notify handler of this master transaction port
    eslapi::CASINotifyHandlerIF *notifyHandler;

    std::vector<eslapi::CASITransactionIF*> slaves;
    eslapi::CASITransactionProperties casiProperties;
    // Should keep a list of TSlaves
    eslapi::CASITransactionIF ** memTable;
    eslapi::CASITransactionIF * defaultSlavePointer;
```



```
    uint64_t shiftVal;
    uint64_t numBlocks;
    eslapi::CASITransactionIF *slave;
};
```

More than one transaction slave can be connected to a bus master port, provided that their address regions do not overlap. The bus master port is configurable to support different types of memory organizations and sizes.

The constructor takes two parameters in addition to the port-name:

- `block` denotes the size of one memory block

  Note

  The block size must not be zero.

- `space` specifies the size of the entire supported memory range. The number of admissible memory blocks is therefore given by the ratio of space and block.

See *The CASITransactionIF interface* on page 2-56 for a description of the member functions.

2.5.12 AXI and AHB transactions

The `AXITransactionInfo` and `AHBTransactionInfo` classes extend the `CASITransactionInfo` class with additional functionality specific to the AXI and AHB buses. See the `AXI_Transaction.h` and `AHB_Transaction.h` files for more details of these classes.

2.6 The signal interface classes

The signal interfaces are used for connections between components that are similar to hardware signal connections. Multiple signal-slaves can be connected to a signal-master that drives the signal in all connected slaves (fan-out).

This section contains the following subsections:

- *The CASISignalProperties structure*
- *The CASISignalIF Interface* on page 2-92
- *CASISignalIF::driveSignal()* on page 2-93
- *CASISignalIF::readSignal()* on page 2-93
- *The CASISignalSlave class* on page 2-94
- *The CASISignalMasterIF class* on page 2-96
- *The predefined sc_port<CASISignalIF, 1> class* on page 2-97.

2.6.1 The CASISignalProperties structure

It is possible to define signal properties, the bit width of the signal, and whether the signal port is optional.

Example 2-21 Defining port properties

```
struct CASISignalProperties
{
   bool isOptional;
   uint32_t bitwidth;
};
```

The following functions are offered by all the signal interfaces:

- `virtual const CASISignalProperties getProperties();`
- `virtual void setProperties(const CASISignalProperties prop);`

The properties must be set in the constructor, directly after creation of the port.

2.6.2 The CASISignalIF Interface

The CASISignalIF class is the interface that is used for signal-based communication:

- CASISignalSlave inherits from CASISignalIF and provides the base for signal slave ports (see *The CASISignalSlave class* on page 2-94).

  The CASISignalSlave class provides a default implementation for most of the interface functions.

- sc_port<CASISignalIF, 1> uses the templated sc_port class to provide the interface in a signal master port (see *The predefined sc_port<CASISignalIF, 1> class* on page 2-97).

The signal interface enables the implementation of signal-level communication by providing the driveSignal() function to allow driving a signal. The value parameter represents the value being transmitted and the extValue allows for extra value fields if more then 32 bits are required to be transmitted.

Note

The readSignal() functions are provided for convenience only. Signal communication is single-directional from the master to the slave and done through driveSignal().

Example 2-22 CASISignalIF interface

```
class CASISignalIF : public CASIPortIF, public sc_interface
{
public:
    CASISignalIF () {}
    CASISignalIF (std::string & name) {}
    virtual ~CASISignalIF()     {};

    virtual void driveSignal(uint32_t value, uint32_t* extValue) = 0;
    virtual uint32_t readSignal() = 0;
    virtual void readSignal(uint32_t* value, uint32_t* extValue) = 0;
    virtual void connect(CASISignalMasterIF* iface) = 0;
    virtual void disconnect(CASISignalMasterIF* iface) = 0;

    virtual CASISignalMasterIF* getMaster() = 0;
    virtual const CASISignalProperties * getProperties() = 0;
    virtual void setProperties (const CASISignalProperties * prop) = 0;
    virtual std::string getName() = 0;
    virtual CASIInterfaceType getType() = 0;
    virtual std::string getPortInstanceName() = 0;
    virtual void setPortInstanceName (const std::string & name) = 0;
    virtual void setCASIOwner(CASIModuleIF * owner) = 0;
```



```
    virtual CASIModuleIF * getCASIOwner() = 0;
    virtual void enablePort(bool enable) = 0;
    virtual const bool isPortEnabled() = 0;
    virtual CASIStatus bypass(uint32_t msgSize, uint32_t* message,
             uint32_t rspSize, uint32_t* response) = 0;
};
```

The signal interface can be used for simple signals (0 or 1), but it can also be used for up to signals of larger bit widths, because the value is passed as a 32-bit unsigned integer and a pointer to bits beyond 32.

2.6.3 CASISignalIF::driveSignal()

This function enables master-to-slave signal communication.

```
virtual void driveSignal(uint32_t value, uint32_t* extValue) = 0
```

where:

`value` is the 32 bit signal value.

`extValue` is a pointer to an array of 32 bit values. The size and meaning of the array are dependent on the model.

Note

The `driveSignal()` function is provided for convenience only in the signal slave port. The signal communication is meant to be single-directional, from the master to the slave.

2.6.4 CASISignalIF::readSignal()

This function enables slave-to-master signal communication.

```
virtual void readSignal(uint32_t* value, uint32_t* extValue) = 0
```

where:

`value` is the 32 bit signal value.

`extValue` is a pointer to an array of 32 bit values. The size and meaning of the array are dependent on the model.

The shorter form returns a 32-bit value from the slave:

```
virtual uint32_t readSignal() = 0
```

Note

The `readSignal()` functions are provided for convenience only in the signal master port. The signal communication is meant to be single-directional, from the master to the slave, through `driveSignal()`.

2.6.5 The CASISignalSlave class

The `CASISignalSlave` class inherits from `CASISignalIF` and provides the basic signal slave implementation. The class that implements the actual signal slave port must inherit from this class and implement at least the `driveSignal()` and `readSignal()` functions.

Example 2-23 CASISignalSlave class

```
class CASISignalSlave : public CASISignalIF
{
public:
    CASISignalSlave(CASIModule * owner, const std::string& name)
        : portName(name), casiPortInstanceName(""), casiOwner(owner),
        casiIsEnabled(true), casiMaster(NULL)
        {memset(&casiProperties,0,sizeof(casiProperties));}
   CASISignalSlave(const std::string& name)
        : portName(name), casiPortInstanceName(""), casiIsEnabled(true),
        casiMaster(NULL), casiOwner(NULL)
        {memset(&casiProperties,0,sizeof(casiProperties));}
    virtual ~CASISignalSlave() {}

    // Functions to be implemented.
    virtual void driveSignal(uint32_t value, uint32_t* extValue) = 0;
    virtual uint32_t readSignal() = 0;
    virtual void readSignal(uint32_t* value, uint32_t* extValue) {};

    // Functions with default implementations.
    std::string getName() { return portName; }
    CASIInterfaceType getType() { return CASI_SIGNAL_SLAVE; }
    virtual void connect(CASISignalMasterIF*);
    virtual void disconnect(CASISignalMasterIF*);
    virtual CASISignalMasterIF* getMaster() { return casiMaster; }
    virtual void setPortInstanceName(const std::string& name)
            { casiPortInstanceName = name; }
    virtual std::string getPortInstanceName() { return casiPortInstanceName; }
    virtual void setCASIOwner(CASIModuleIF* owner) { casiOwner = owner; }
    virtual CASIModuleIF* getCASIOwner() { return casiOwner; }
    virtual void enablePort(bool enable) { casiIsEnabled = enable; }
    virtual const bool isPortEnabled() { return casiIsEnabled; }
```



```
    virtual void pmessage( const std::string& msg,
            CASIMessageType type = CASI_MSG_INFO );
    virtual void pmessage( CASIMessageType type, const char *fmt, ... );
    virtual const CASISignalProperties * getProperties()
            { return &casiProperties; }
    virtual void setProperties(const CASISignalProperties * prop)
            { casiProperties = *prop; }
    virtual CASIStatus bypass(uint32_t msgSize, uint32_t* message,
            uint32_t rspSize, uint32_t* response);

    // Return interface if requested
    virtual CAInterface * ObtainInterface(if_name_t ifName,
            if_rev_t minRev, if_rev_t *   actualRev)
    {
        if((strcmp(ifName,"eslapi.CASISignalSlave2") == 0) && (minRev <= 0))
        {
            *actualRev = 0;
             return this;
        }
        return NULL;
    }

private:
    std::string portName;
    std::string casiPortInstanceName;
    bool casiIsEnabled;

protected:
    CASISignalProperties casiProperties;
    CASISignalMasterIF* casiMaster;
    CASIModuleIF * casiOwner;
};
```

2.6.6 The CASISignalMasterIF class

This interface enables master signal port customization. The predefined sc_port<CASISignalIF,1> and sc_port<CASISignalIF,1> ports descend from this class.

Example 2-24 CASISignalMasterIF class

```
class CASISignalMasterIF : public CASIPortIF
{
public:
    virtual ~CASISignalMasterIF() {}
    // These functions are used to specify the port connectivity.
    virtual void connect(CASISignalIF* iface) = 0;
    virtual void disconnect(CASISignalIF* iface) = 0;
    virtual const std::vector<CASISignalIF*>& getSlaves() = 0;

    // Customize interface functions.
    virtual const CASISignalProperties * getProperties() = 0;
    virtual void setProperties(const CASISignalProperties * prop) = 0;

    // Common port functions
    virtual std::string getName() = 0;
    virtual CASIInterfaceType getType() = 0;
    virtual std::string getPortInstanceName() = 0;
    virtual void setPortInstanceName(const std::string & name) = 0;
    virtual void setCASIOwner(CASIModuleIF * owner) = 0;
    virtual CASIModuleIF * getCASIOwner() = 0;
    virtual void enablePort(bool enable) = 0;
    virtual const bool isPortEnabled() = 0;
};
```

The connect() function takes an CASISignalIF pointer as a parameter and must register this pointer internally. In fan-out is allowed, multiple slaves can be connected to a master, otherwise only a single slave is connected to each master.

The disconnect() function is used to remove the interface specified as the parameter from the list of connections.

2.6.7 The predefined sc_port<CASISignalIF, 1> class

sc_port<CASISignalIF, 1> inherits from CASIPortIF and implements the functions in CASISignalIF to provide a signal master port.

Caution

The sc_port<CASISignalIF, 1> class is deprecated.

Use sc_port<CASISignalIF, 0> instead. It supports multiple masters.

Master ports encapsulate the connection of a component to another component's slave port. The master ports provide standardized ways of accessing connected slave ports without the requirement to know about the slave ports that are actually connected to them.

The slave ports behavior depends on the meaning and requirements of the resources corresponding to those slaves. For this reason, the user must implement this behavior in the slave read/write functions. However, because master ports only forward the requests to the connected slaves, master ports can have a generic implementation that can be directly used to instantiate master ports in the components. The CASITransactionMaster represents such a generic implementation.

This class is a predefined port class that is used for all signal master ports.

Example 2-25 sc_port<CASISignalIF, 1> class

```
template <> class sc_port<CASISignalIF, 1> : public CASIPortIF
{
public:
    sc_port<CASISignalIF,1>(CASIModule * owner, const std::string& name)
            : portName(name), casiPortInstanceName(""), casiOwner(owner),
            casiIsEnabled(true), slave(NULL)
            { memset(&casiProperties,0,sizeof(casiProperties)); }
    virtual ~sc_port<CASISignalIF,1>() {}

    // Configuration functions
    std::string getName() { return portName; }
    CASIInterfaceType getType() { return CASI_SIGNAL_MASTER; }
    virtual void setPortInstanceName(const std::string& name)
            { casiPortInstanceName = name; }
    virtual std::string getPortInstanceName() { return casiPortInstanceName; }
    virtual void setCASIOwner(CASIModuleIF* owner) { casiOwner = owner; }
    virtual CASIModuleIF* getCASIOwner() { return casiOwner; }
    virtual void enablePort(bool enable) { casiIsEnabled = enable; }
    virtual const bool isPortEnabled() { return casiIsEnabled; }
    // Prints a message
```

```
    virtual void pmessage( const std::string& msg,
            CASIMessageType type = CASI_MSG_INFO );
    virtual void pmessage( CASIMessageType type, const char *fmt, ... );
    virtual const CASISignalProperties * getProperties()
            { return &casiProperties; }
    virtual void setProperties(const CASISignalProperties * prop)
            { casiProperties = *prop; }
    // Connectivity functions
    virtual void connect(CASISignalIF* iface);
    virtual void disconnect(CASISignalIF* iface);
    virtual void replace(CASISignalIF* old_iface, CASISignalIF* new_iface);
    virtual const std::vector<CASISignalIF*>& getSlaves();

    // Behavioral functions
    // These will be forwarded to all connected slaves
    // Master-to-slave signal communication.
    virtual void driveSignal(uint32_t value, uint32_t* extValue);
    // Slave-to-master signal communication.
    virtual uint32_t readSignal();
    // Slave-to-master signal communication.
    virtual void readSignal(uint32_t *value, uint32_t *extValue);

    // convenience connection operator
    void operator()( CASISignalIF& interface_ );
    virtual const char* kind() const
            { return kind_string; }
    typedef sc_port_base base_type;
    typedef sc_port_b<CASISignalIF> this_type;

private:
    void bind( CASISignalIF& interface_ );
    //void operator() ( CASISignalIF& interface_ );
    void bind( this_type& parent_ );
    void operator() ( this_type& parent_ );
    int size() const;
    CASISignalIF* operator -> ();
    const CASISignalIF* operator -> () const;
    CASISignalIF* operator [] ( int index_ );
    const CASISignalIF* operator [] ( int index_ ) const;
    virtual sc_interface* get_interface() {return NULL;};
    virtual const sc_interface* get_interface() const {return NULL;};

private:
    // Data Section
    static const char* const kind_string;
    std::string portName;
    std::string casiPortInstanceName;
    CASIModuleIF* casiOwner;
    bool casiIsEnabled;
    casi_signal_if *slave;
```



```
    std::vector<CASISignalIF*> slaves;
    CASISignalProperties casiProperties;
};
```

The signal master port supports fan-out, meaning that any number of signal slaves can be connected to this port. If the signal changes, all signal slaves are notified by calling their `driveSignal()` function.

When the user calls the `driveSignal/readSignal` functions, the `drive/read` action is forwarded to the signal slave ports connected to this master port.

2.7 The component factory class CASIFactory

The factory class in the ESL API provides a standardized mechanism for dynamic component creation and enables instantiating and creating a component without the need for access to its constructor.

Example 2-26 CASIFactory class

```
class CASIFactory
{
public:
    CASIFactory(const std::string& name);
    virtual ~CASIFactory() {};
    virtual CASIModuleIF *createInstance(CASIModuleIF *parent,
            const std::string& instance_name) = 0;
};
```

Note

Typically the component factory not does require any changes. The factory class is not required if you are using the models in a pure SystemC environment.

Only two functions must be implemented in the `CASIFactory` class:

- *CASIFactory::CASIFactory()*
- *CASIFactory::createInstance()* on page 2-101.

2.7.1 CASIFactory::CASIFactory()

The constructor simply calls the default constructor, passing the name of the component as a parameter. It is important that this name is equivalent to the name that is returned by the `getName()` function of the corresponding component. ARM recommends the declaration of the constant that contains the name of the model as a string:

```
#define MODEL_NAME "My_Model";
MyModelFactory::MyModelFactory() : CASIFactory(MODEL_NAME) {}
```


2.7.2 CASIFactory::createInstance()

The `createInstance()` function is called by the simulation backend to instantiate the component. It must simply call the `new` operator of the component class and return the pointer to this created instance. A pointer to the parent component is provided as a parameter and this must be passed on to the component's constructor:

```
CASIModuleIF * MyModelFactory::createInstance(CASIModuleIF *parent,
            const std::string& instance)
{
    return new MyModel(parent, instance);
}
```

The constructor must be implemented calling the parent's constructor and must pass the name of the system as a parameter:

```
MyModelFactory::MyModelFactory() : CASIFactory("MyModel") {}
```

Use the `createInstance()` function to create an instance of the component by passing a pointer to the parent component as shown in Example 2-27:

Example 2-27 Creating an instance of a component

```
CASIModule * MyModelFactory::createInstance(CASIModuleIF *parent,
            const std::string& instance_name)
{
    return new MyModel(parent, const std::string& instance_name);
}
```

2.8 The save/restore interface CASISaveRestore

The ESL API supports saving of state information to allow resuming the simulation from a given point. The state information is stored to an `CASIODataStream`.

The data in file that is the destination of the `CASIODataStream` is stored as a binary stream of encoded information that is not dependent on the operation system, CPU, or byte order of the host computer. A stream that is written by a PC under Windows can therefore be read by a Sun SPARC running Solaris (assuming the models are also available on both platforms). Every component in that system must support this interface to enable saving of state for an entire system.

2.8.1 Enabling save/restore support

For a component to support the advanced save and restore feature:

1. The component must be derived from `CASISaveRestore`:

   ```
   class MyModel : public CASIModule, public CASISaveRestore {
   ```

2. The `getProperty()` function must return `yes` for the `CASI_PROP_SAVE_RESTORE` property. This informs the simulation environment about the save/restore capability of the component.

3. The Save/Restore functionality of the component now exists but has not been initialized. If the initialization method is not called, it is not possible to save the component's state. To initialize the Save/Restore functionality, call `initStream(this)` from the component's constructor.

Example 2-28 Initialization of save/restore functionality

```
// Constructor for MyModel
MyModel::MyModel( … )
{
    // Create Component
        ...

    // Call method inherited from CASISaveRestore
    // required to enable saving and restoring state information
    initStream( this );}
```

4. Two pure virtual methods inherited from `CASISaveRestore` must be implemented. These methods must implement the details of saving and restoring of the component's state information.

Example 2-29 saveData and restoreData

```
// Implementation of CASISaveRestore interface to save state
bool MyModel::saveData( CASIODataStream &data )
{
    // return save was successful
    return true;
}

// Implementation of CASISaveRestore interface to restore state
bool MyModel::restoreData( CASIODataStream &data )
{
    // return restore was successful
    return true;
}
```

The above two methods are valid for a component that has no state information to save, but is required to support this feature so more complex systems that include the model can use Save/Restore.

It is not unusual for components to not have any state to save. For example, a Fan Out component (FOUT) that distributes a signal to several components, completes its task in one cycle and has no state that must be saved. But for more complex components, there might be a lot of information that must be remembered.

2.8.2 MyComponent::saveData

Saving your components state is straightforward. An `CASIODataStream` is passed into the method. The stream can be written to in a similar fashion to regular streams (inside of this method you can only write to the stream).

Example 2-30 depicts a simple example that saves several values and some character strings. See CASISaveRestore.h for details of the CASIODataStream class.

Example 2-30 Passing the CASIODataStream parameter

```
bool MyComponent::saveData( CASIODataStream &data )
{
    float version = 1.29;
    data << version;                 // store version number
    data << internalCycleCount;      // store 64bit cycle count
    for( int i = 0; i < NUM_REGISTERS; i++ ) // loop through all
        data << registerValues[i];   // registers saving all
    data << stringData;             // store char* array
    // no errors
    return true;
}
```

Note

The order that items are written to the stream is the same as the order they must be read back. When writing to the stream, CASIODataStream manages the overhead of ensuring endian order.

Byte order

```
int byteOrder()    const;
```

The byteOrder() function returns the current byte order setting and is defined from:

```
enum ByteOrder { BigEndian, LittleEndian };
```

This is the order that all data is written out as and is independent of the order used natively by the host platform.

```
SetByteOrder( int byteOrder )
```

Sets the byte order to *byteOrder*. This is the order that all data is written out as.

The *byteOrder* parameter can be either:

- CASIODataStream::BigEndian
- CASIODataStream::LittleEndian.

By default this is set to big endian. Changing this is not recommended, as the reader must also then be changed.

Operators

`CASIODataStream &operator<<( int8_t i )`

Writes a signed 8-bit integer to the `CASIODataStream` object and returns a reference to the object.

`CASIODataStream &operator<<( uint8_t i )`

Writes an unsigned 8-bit integer to the `CASIODataStream` object and returns a reference to the object.

`CASIODataStream &operator<<( int16_t i )`

Writes a signed 16-bit integer to the `CASIODataStream` object and returns a reference to the object.

`CASIODataStream &operator<<( uint16_t i )`

Writes an unsigned 16-bit integer to the `CASIODataStream` object and returns a reference to the object.

`CASIODataStream &operator<<( int32_t i )`

Writes a signed 32-bit integer to the `CASIODataStream` object and returns a reference to the object.

`CASIODataStream &operator<<( uint32_t i )`

Write an unsigned 32-bit integer to the `CASIODataStream` object and returns a reference to the object.

`CASIODataStream &operator<<( int64_t i )`

Writes a signed 64-bit integer to the `CASIODataStream` object and returns a reference to the object.

`CASIODataStream &operator<<( uint64_t i )`

Writes an unsigned 64-bit integer to the `CASIODataStream` object and returns a reference to the object.

`CASIODataStream &operator<<( float f )`

Writes a 32-bit floating-point number to the `CASIODataStream` object (using the standard IEEE754 format) and returns a reference to the object.

`CASIODataStream &operator<<( double f )`

Writes a 64-bit floating-point number to the `CASIODataStream` object (using the standard IEEE754 format) and returns a reference to the object.

`CASIODataStream &operator<<( const char *std::string )`

Writes a `NULL` terminated string to the `CASIODataStream` object and returns a reference to the object.

MyComponent::writeBytes()

`CASIODataStream &writeBytes( const char *buffer, unsigned int length )`

Writes `length` number of characters from the `buffer` to the `CASIODataStream` object and returns a reference to the object.

`CASIODataStream &writeRawBytes( const char *buffer, unsigned int length )`

As above except this method does not serialize the data forcing the reader to know the exact length of the data being written. It is advised that you do not use this method except for exceptional circumstances.

2.8.3 MyComponent::restoreData

Restoring the state of your component is almost as straightforward as writing it out. An `CASIIDataStream` is passed into the method and this can be read from in a similar fashion to regular streams.

Inside of this method you can only read from the stream. But, you now have the added requirement that you read in the data exactly how it was written. The following depicts the same as Example 2-30 on page 2-104, but adds restoring the several values and character strings.

Example 2-31 Restoring state

```
bool MyComponent::restoreData( CASIIDataStream &data )
{
    float currentVersion = 1.29;
    float version;
    // if version numbers don't match then return an error
    // this will abort the restoring of state in the whole system.
    data >> version;                    // Read in version of state data
    if ( version != currentVersion )
            return false;

    data >> internalCycleCount;         // restore 64bit cycle count
    for( int i = 0; i < NUM_REGISTERS; i++ )// loop through all
             data >> registerValues[i];          // registers restoring all

    // The stream will allocate memory for stringData using new
    // we must delete[] it
```



```
    char *stringData;
    data >> stringData;                // restore char* array
    setData( stringData );            // do something with data
    delete[] stringData;             // finished with stringData

    // no errors
    return true;
}
```

The CASIIDataStream object manage the byte order for the written data and no user involvement is required. The reading order, however, must match the order it was written out. See CASISaveRestore.h for a description of the data stream object.

The code in Example 2-31 on page 2-106 also returns an error if the saved state returns a wrong version. Implementing the version number in the model is desirable because as the model develops, the state information might:

- stay the same or change
- be ordered differently
- be represented in a different way.

This allows the model version reading the state data to reject an older or newer version of the state information or possibly import state information from an older model.

MyComponent::byteOrder()

`CASIIDataStream APIint byteOrder()  const`

Returns the current byte order setting either BigEndian or LittleEndian defined as:

`enum ByteOrder { BigEndian, LittleEndian }`

This is the order that all data is read in as independent to the host platform.

`SetByteOrder( int byteOrder )`

Sets the byte order to byteOrder. This is the order that all data s read in as.

The byteOrder parameter can be either:

- CASIODataStream::BigEndian
- CASIODataStream::LittleEndian.

By default this is set to big endian. Changing this is not recommended, as the writer must also then be changed.

MyComponent::&operator

`CASIIDataStream &operator>>( int8_t i )`

Reads a signed 8-bit integer from the `CASIIDataStream` object and returns a reference to the object.

`CASIIDataStream &operator>>( uint8_t i )`

Reads an unsigned 8-bit integer from the `CASIIDataStream` object and returns a reference to the object.

`CASIIDataStream &operator>>( int16_t i )`

Reads a signed 16-bit integer from the `CASIIDataStream` and returns a reference to the object.

`CASIIDataStream &operator>>( uint16_t i )`

Reads an unsigned 16-bit integer from the `CASIIDataStream` object and returns a reference to the object.

`CASIIDataStream &operator>>( int32_t i )`

Reads a signed 32-bit integer from the `CASIIDataStream` object and returns a reference to the object.

`CASIIDataStream &operator>>( uint32_t i )`

Reads an unsigned 32-bit integer from the `CASIIDataStream` object and returns a reference to the object.

`CASIIDataStream &operator>>( int64_t i )`

Reads a signed 64-bit integer from the `CASIIDataStream` object and returns a reference to the object.

`CASIIDataStream &operator>>( uint64_t i )`

Reads an unsigned 64-bit integer from the `CASIIDataStream` object and returns a reference to the object.

`CASIIDataStream &operator>>( float f )`

Reads a 32-bit floating-point number from the `CASIIDataStream` object (using the standard IEEE754 format) and returns a reference to the object.

`CASIIDataStream &operator>>( double f )`

Reads a 64-bit floating-point number from the `CASIIDataStream` object (using the standard IEEE754 format) and returns a reference to the object.

`CASIIDataStream &operator>>( const char *std::string )`

Reads a NULL terminated string from the `CASIIDataStream` object and returns a reference to the object.

MyComponent::readBytes()

`CASIIDataStream &readBytes( char *&buffer, unsigned int &length )`

Reads into `buffer` from the `CASIIDataStream` object and returns a reference to the object.

The `buffer` is allocated using `new`. Use the `delete[]` operator to destroy it. If the length of the stored data is zero or buffer cannot be allocated, then buffer is set to `NULL`.

The `length` parameter is set to the number of bytes actually read into `buffer`.

`CASIIDataStream &readRawBytes( char *buffer, unsigned int length )`

Use this method to read in data that was written with `writeRawBytes(…)`. This method reads in data that has not been serialized. This means that the caller must know the exact length of the data they wish to read and the buffer must be pre-allocated. Reading more data than was actually written with `writeRawBytes(…)` can cause subsequent reads to be erroneous. The raw methods are not therefore recommended for general use.

2.9 Integrating CASI models into OSCI SystemC

This section describes the steps required to setup the simulation in the `sc_main()` function for the OSCI SystemC environment.

1. Example 2-32 shows an example of instantiating the master and slave modules in the OSCI SystemC `sc_main()` function:

Example 2-32 Instantiating the CASI models in OSCI SystemC

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

   //Instantiate the modules
   Master_casi *MasterComp = new Master_casi (“Master”);
   Slave_casi *SlaveComp = new Slave_casi (“Slave”);

   …
```

2. Add the code in Example 2-33 to connect the models to the CASI scheduler:

Example 2-33 Connecting the models to the scheduler

```
   // create master clock
   CASIClockDriverRoot master_clk ("master_clk", 1, SC_NS);
   MasterComp->connect (& master_clk);
   SlaveComp->connect (& master_clk);
```

3. Add the code in Example 2-34 to connect the models to each other:

Example 2-34 Interconnecting the models

```
   //Connect the ports
   (* MasterComp->p_TM_TMaster) (* SlaveComp->p_TS_TSlave);
   (* SlaveComp->p_SM_SMaster) (* MasterComp->p_SS_SSlave);

   //Call all interconnect functions
   MasterComp->interconnect();
   SlaveComp->interconnect();
```


Note

The interconnect() functions of the models must be called *after* connecting the CASI ports of the respective models. The `interconnect()` functions rely on the connections being already established.

4. Example 2-35 shows how to initialize the component parameters and call the CASI simulation stages:

Example 2-35 Calling the CASI simulation stages

```
//Setting the parameters
MasterComp->setParameter("No of Transaction","10");
MasterComp->setParameter("Data Value","0xff");
SlaveComp->setParameter("Enable Debug Messages", "true");

//Call all init functions
MasterComp->init();
SlaveComp->init();
//Interconnecting the models

…

//Call resets
MasterComp->reset(CASI_RESET_HARD,NULL);
SlaveComp->reset(CASI_RESET_HARD,NULL);
```

5. Example 2-36 shows the code that starts the simulation:

Example 2-36 Running the simulation

```
…

//Start the simulation
sc_start(100);
```

6. Example 2-37 shows the code that terminates the simulation:

Example 2-37 Terminating the simulation

```
   …

   //Terminate the simulation
   MasterComp->terminate();
   SlaveComp->terminate();
```

7. Example 2-38 shows flushing the output buffers and exiting `sc_main()`:

Example 2-38 Exiting sc_main()

```
    fflush (stdout);
    fflush (stderr);

    return 0;
}
```

Chapter 3
The Cycle Accurate Debug Interface

This chapter describes the *Cycle Accurate Debug Interface* (CADI) that provides debug access to memory values, registers values, or disassembly of code in a component. It contains the following sections:

Note

The examples and descriptions in this chapter are specific to components that are running in a CASI environment only. The usage of CADI in non-CASI environments might be different.

3.1 Introduction

The Cycle Accurate Debug Interface can be used to:

- Display the contents of registers and memory within the simulation environment for any type of component.
- Enable interaction with an existing debugger. This simplifies integrating a core model with established user base for an existing debugger or where there is a limited range of debuggers available for the architecture.

Figure 3-1 on page 3-3 shows the CADI class hierarchy.

The CADI functionality is exposed through the `CADI` and `CADICallBackObj` interfaces:

- CADI handles the requests from the outside world into the target. `CADI` objects are implemented by the models and can be obtained through the `getCADI()` method of the `CASIModule` interface.
- The `CADICallBackObj` handles the requests made by the target towards the outside world. `CADICallBackObj` objects must be implemented by the system builder and registered with the target.

  The `CADICallbackObj` interfaces are the mechanism through which the low-level simulation commands are issued by the CADI target.

The `CADICallbackObj` is also used for semihosting requests. A program running on a target can issue console operations. Instead of requiring the simulation of a full operating system, CADI offers the option to forward the console operations to the host operating system through a semihosting mechanism.

Most of the functionality available through `CADICallbackObj` can be obtained by polling the state of the target model each cycle through the regular CADI interface. However, it is more efficient to have target make the semihosting calls as required rather than having the overhead of a large number of polling calls.

The callback methods can be called asynchronously at any time during simulation. It is recommended that the callback handlers do as little processing as possible and, for example, only set flags for later processing. All of the callback processing must be completed synchronously.

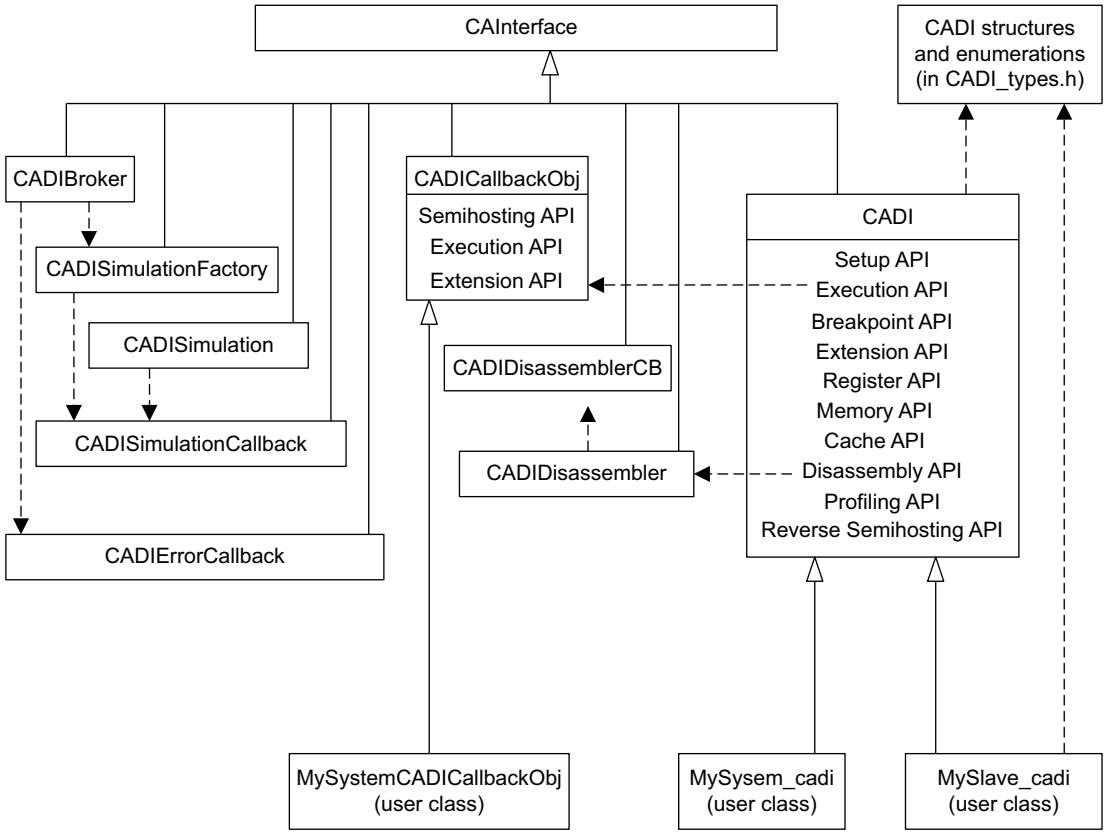

Figure 3-1 CADI class overview

Note

The top-level functions are pure virtual and the lowest level functions implement the component-specific behavior.

There are several conceptually distinct parts of the CADI interface:

CADI
- Setup
- Execution
- Breakpoints
- Extension
- Register
- Memory
- Cache
- Disassembly
- Profiling
- Reverse semihosting.

CADICallbackObj
- Semihosting
- Execution
- Extension.

A given CADI target might only implement a subset of the CADI interface methods, according to the associated CASI model semantics. For instance, a target for a memory model only implements the Memory API and does not implement the Register API or the Disassembly API. For API implementation details for the CADI targets of a specific model, see the model documentation.

The Breakpoint and Execution APIs might not be implemented by all the core models. If not, these parts of the CADI interface must be overridden by the user with appropriate functionality. This is the only case where user extensions of the CADI interface are required and useful.

Note

See the `CADITypes.h` file for definitions of enumerations and data structures that are used with the CADI interface.

3.1.1 Simulation control and the CADI interface

The CADI is intended for use in conjunction with the Cycle-Accurate Simulation Interface (CASI) to allow inspection and modification of the internal state of SystemC models through an externally attached debugger. The models run under the supervision of a simulation host that is in charge of managing the SystemC cycle-based simulation loop, calling the communicate/update methods of the loop in each cycle.Each debuggable CASI model in a system exposes a CADI interface (target) that can be accessed using the `getCADI()` method of the `CASIModule` interface. A CASI model and the associated CADI target are considered fully constructed and initialized after the call to the `init()` function of the `CASIModule` interface. For more information on simulation control and startup, see *Accessing the debug interface from sc_main()* on page 3-99.

An example CADI interaction architecture is shown in Figure 3-2. Typical CADI interaction architectures might be more complex, depending on the actual requirements of the models, simulation host and debugger.

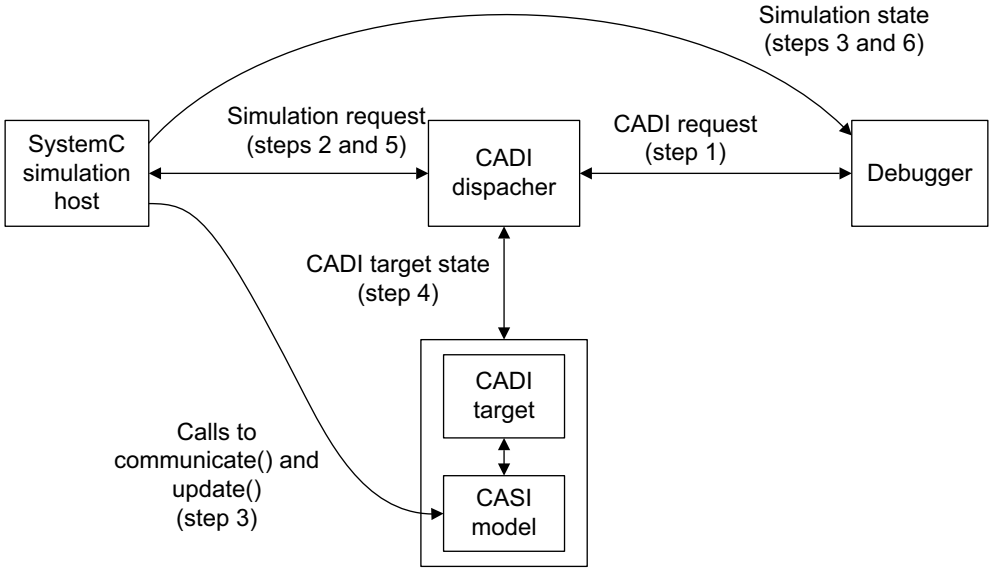

Figure 3-2 CADI and CASI interaction

In Figure 3-2 on page 3-5, a CADI enabled CASI model is integrated in a system containing a SystemC simulation host and an external debugger. The simulation and debug sequence is:

1. The debugger issues an execution request `run()`.
2. The CADI dispatcher translates this into a start simulation request made to the simulation host
3. The simulation host starts running the simulation loop and makes a call to `communicate()` and `update()` of the model for each cycle. It also signals the debugger that the simulation has started.
4. The simulation proceeds until the CADI target detects that a breakpoint was hit and issues a mode change callback to the CADI Dispatcher.
5. When the current simulation cycle is finished, the CADI dispatcher requests the simulation host to stop the simulation loop.
6. The simulation host pauses the simulation and it signals the debugger that the simulation has stopped.

3.1.2 Registering the CADI interface with the component

To make the CADI interface available, the following changes must be made in the component:

1. declare a pointer to your CADI interface as a private member in your component
2. instantiate the CADI interface from the `init()` function of the component
3. implement the function `getCADI()`, returning a pointer to the CASI interface

4. add the property CASI_PROP_CADI_SUPPORT to the getProperty() function. The function must return yes for this option.

Example 3-1 CADISlave example

```
class CADISlave : public CADI
{
public:
    CADISlave(Slave_casi* c);
    virtual ~CADISlave();

public:
    // Register access functions
    . . .
    // Memory access functions
    . . .

private:
    Slave_casi*                target;

    // Register related info
    CADIRegInfo_t*             regInfo;
    CADIRegGroup_t*            regGroup;

    // Memory related info
    CADIMemSpaceInfo_t*        memSpaceInfo;
    CADIMemBlockInfo_t*        memBlockInfo;
};
```

3.1.3 CADI API overview

This section describes the interfaces in the CADI class. It contains the following sections:

- *Setup API* on page 3-8
- *Breakpoint API* on page 3-8
- *Execution API* on page 3-8
- *Register API* on page 3-8
- *Memory API* on page 3-9
- *Cache API* on page 3-10
- *Profiling API* on page 3-10
- *Disassembly API* on page 3-10.

Note

For details of the structures, enumerations, and defines used with the CADI interface, see the `CADITypes.h` file.

Note

Unsupported functions must return `CADI_STATUS_CmdNotSupported` if called.

Setup API

The interface API controls the interaction between the host, the debugger, and the CADI target. Use this API to:

- control the CADI related memory management
- inspect the actual properties of a given CADI object
- if necessary, register any `CADICallbackObj` callbacks.

Breakpoint API

The breakpoint API enables defining breakpoints in the target model that refer to:

- the address, or range of addresses, of an instruction
- the content of a memory location
- the content of a register
- temporary breakpoints for *run to* debugger behavior.

Execution API

The execution API enables a debugger to:

- control the execution using various asynchronous execution commands
- control the executed program.

It can, for example, manage the asynchronous commands of loading or resetting a program.

Register API

The register API exposes the internal state of the registers of a model for inspection and modification.

If a model has a large number of registers, the registers can be grouped to simplify navigating through the registers.

Models must expose their internal performance counters (for example, Instr Cache Reads, Instr Cache Misses) as registers to be accessible through this interface.

Memory API

The memory API exposes the internal state of the memory of a model for inspection and modification. Memory is exposed through *address spaces* (memory spaces) that represent separately addressable units.

For core models, the memory exposed through the API is not memory contained in the model, but rather memory accessed by the model. The actual reading and writing into such memory is done through the `readDbg()` and `writeDbg()` methods of the respective transaction master ports.

Some core models, however, do contain their own physical memory and expose this memory as a separate address space.

The requirement for multiple address spaces is because of different processor models:

- Harvard architectures can require two separate address spaces
- DSP cores might require up to three address spaces.
- There also exist cores that access different address spaces depending on internal execution flags, for instance distinguishing between secure memory and non-secure memory.

In these cases, each memory transaction master port is usually associated with a separate address space.

A core can have multiple memory transaction master ports. but still define a single unified memory space.

Memory models typically expose a single address space corresponding to their physical memory and other models typically do not expose any memory.

For an address space corresponding to the memory accessed by a (core) model there is currently no relation with the actual structure of the physical memory mapped to the transaction master ports of that core.

Data stored in an address space is organized according to the endianness specified by the flags of that particular memory space. This can be little endian or big endian, with the *invariance* defining the number of bytes in a unit.

Data can also be organized using a model-specific endianness. In these cases, documentation accompanying the model must provide specific details.

The total numbers of bytes in a memory word is defined by `bitsPerMau` / 8. The bytes are divided in groups of invariance bytes. These groups are then arranged in little endian or big endian order. For instance, for `invariance` of 2 and `bitsPerMau` of 64, a little endian word is represented as `b0 b1 b2 b3 b4 b5 b6 b7`, whereas a big endian word is represented as `b6 b7 b4 b5 b2 b3 b0 b1`.

Each address space can be further subdivided in memory blocks. Memory blocks contain additional information pertaining to the intended usage of the memory. This information can be used as hints for memory data presentation dedicated for human consumption, but it has no effect on the actual simulation.

Cache API

These functions enable access to cache memories in the target. Use the `CADIGetCacheInfo()` function to return the cache information for the target. The `CADICacheRead()` and `CADICacheWrite()` functions are used to directly access the cache memory contents.

Profiling API

`CADIProfileSetup()` enables processor execution and memory profiling for a processor. The API includes functions to record or process profile data for executable code, PC values executed, and register or memory accesses.

Disassembly API

The disassembly API enables interpreting the binary instruction codes of a program loaded on a core model into human-readable assembly instructions. The host requests the disassembly string corresponding to the instruction loaded at a specific address. As a convenience, the functionality might also access in history (batch) mode. For programs containing debug information, the relation between a specific instruction and the source code can be reestablished.

3.1.4 CADI Callback API overview

This section describes the interfaces in the CADICallback API. It contains the following sections:

- *Semihosting API*
- *Execution API*
- *Extension API.*

The `CADICallback` class provide an abstract interface that must be defined by the user. For more details, see the documentation in the header files for the actual function calls and structures.

Semihosting API

Programs running on a core model might have to perform file I/O. As an alternative to simulating a full fledged OS, models using the CADI interface can perform the file I/O using the host operating system.

The low level C file I/O calls of the simulated program are forwarded to the host through the registered `CADICallbackObj`. A simple host implementation can simply forward these calls to its own low-level C file I/O interface. However, the host has full freedom to redirect these calls as it sees fit, for instance to redirect writes to the descriptor 2 (stderr) to a special log file.

The stream ids used by this API correspond to the low level C file descriptors and obey the same conventions. 0, 1, 2 are used for stdin, stdout and stderr, while ids greater 2 are used to identify explicitly opened streams.

There is no semihosting support for other types of I/O.

Execution API

The Execution API:

- drives low level simulation signals (simulation mode change)
- informs an external debugger about other important simulation events.

Extension API

This functionality is reserved for future use.

3.2 Defining a CADI interface

A CADI interface can be easily created manually with the following steps:

1. Define a new class derived from `CADI`.
2. If your component is a processor, see the description of the disassembler classes in the header files. Disassembler support is typically not required for user components that are not processors.
3. For register support:
   - customize the register parameters
   - implement the register access functions.
4. For memory support:
   - customize the memory parameters
   - implement the memory access functions.
5. Instantiate this class from within your component class

Note

You are not required to implement functions that are not required by your component. For example, if your component does not have cache or other memory, you do not have to use the cache or memory functions.

See the following subsections for more detail on the CADI classes and methods:

Starting a simulation

- *The CADISimulationFactory class* on page 3-73
- *CADIBroker* on page 3-68

Setting up the CADI interface

These functions enable configuring the CADI interface and connection to the debugger:

- *The component CADI class declaration* on page 3-17
- *The CADI class constructor* on page 3-18
- *CADI::CADIXfaceGetFeatures()* on page 3-19
- *CADI::CADIXfaceGetError()* on page 3-19
- *CADI::CADIXfaceAddCallback()* on page 3-20
- *CADI::CADIXfaceRemoveCallback()* on page 3-21
- *CADI::CADIXfaceBypass()* on page 3-21
- *CADI::CADIGetTargetInfo()* on page 3-22
- *CADI::CADIGetParameters()* on page 3-23
- *CADI::CADISetParameters()* on page 3-23
- *CADI::CADIGetParameterValues()* on page 3-22
- *CADI::CADIGetTargetInfo()* on page 3-22
- *CADI::CADIGetTargetInfo()* on page 3-22.

Defining registers

To define the registers, fill in the `reg_info` and `group_info` structures that are in the top part of the `CADI`*`Template`*`.cpp` file and define the read/write access functions:

- *CADI::CADIRegGetGroups()* on page 3-23
- *CADI::CADIRegGetMap()* on page 3-24
- *CADI::CADIRegGetCompound()* on page 3-25
- *CADI::CADIRegWrite()* on page 3-26
- *CADI::CADIRegRead()* on page 3-26.

Defining memory spaces

To define memory spaces, fill in the `mem_info` structure in the `CADITemplate.cpp` file and define the read/write access functions.

- *CADI::CADIMemGetSpaces()* on page 3-28
- *CADI::CADIMemGetBlocks()* on page 3-28
- *CADI::CADIMemRead()* on page 3-29
- *CADI::CADIMemWrite()* on page 3-30
- *CADI::CADIMemGetOverlays()* on page 3-31
- *CADI::VirtualToPhysical()* on page 3-31
- *CADI::PhysicalToVirtual()* on page 3-32.

Adding cache support

This following debug functions relate to caches.

- *CADI::CADIGetCacheInfo()* on page 3-32
- *CADI::CADICacheRead()* on page 3-32
- *CADI::CADICacheWrite ()* on page 3-33.

Controlling application execution

The Execution APIs modify the execution state of the target:

- *CADIExecMode_t structure* on page 3-91
- *CADI::CADIExecGetModes()* on page 3-34
- *CADI::CADIExecGetResetLevels()* on page 3-34
- *CADI::CADIExecSetMode()* on page 3-35
- *CADI::CADIExecGetMode()* on page 3-35
- *CADI::CADIExecSingleStep()* on page 3-35
- *CADI::CADIExecReset()* on page 3-37
- *CADI::CADIExecContinue()* on page 3-37
- *CADI::CADIExecStop()* on page 3-37
- *CADI::CADIExecGetExceptions()* on page 3-38
- *CADI::CADIExecAssertException()* on page 3-38
- *CADI::CADIExecGetPipeStages()* on page 3-39
- *CADI::CADIExecGetPipeStageFields()* on page 3-39
- *CADI::CADIExecLoadApplication()* on page 3-40
- *CADI::CADIExecUnLoadApplication()* on page 3-40
- *CADI::CADIExecGetLoadedApplication()* on page 3-40
- *CADI::CADIExecSetApplication()* on page 3-41
- *CADI::CADIGetInstructionCount()* on page 3-42
- *CADI::CADIGetCycleCount()* on page 3-42.

Setting breakpoints

Use the breakpoint API to enable the debugger to set and respond to breakpoints:

- *CADI::CADIBptGetList()* on page 3-43
- *CADI::CADIBptRead()* on page 3-43
- *CADI::CADIBptSet()* on page 3-44
- *CADI::CADIBptClear()* on page 3-44
- *CADI::CADIBptConfigure()* on page 3-44.

Adding disassembly support

If the component supports disassembly, the disassembly API can be used to display the disassembly during a simulation:

- *The CADIDisassembler class* on page 3-45
- *CADI::CADIgetDisassembler()* on page 3-20
- *CADIDisassembler::getType()* on page 3-46
- *CADIDisassembler::getModeCount()* on page 3-46
- *CADIDisassembler::getModeNames()* on page 3-46
- *CADIDisassembler::getCurrentMode()* on page 3-47.
- *CADIDisassembler::getSourceReferenceForAddress()* on page 3-47.
- *CADIDisassembler::getSourceReferenceForAddress()* on page 3-47.
- *CADIDisassembler::getAddressForSourceReference()* on page 3-47.
- *CADIDisassembler::getDisassembly()* on page 3-48
- *CADIDisassembler::GetInstructionType()* on page 3-48
- *CADIDisassembler::ObtainInterface()* on page 3-49 (provided for legacy code).

Adding debug profiling support

These APIs are used to access execution and memory profiling for a processor.

- *CADIProfiling::CADIProfileSetup()* on page 3-51
- *CADIProfiling::CADIProfileControl ()* on page 3-52
- *CADIProfiling::CADIProfileTraceControl ()* on page 3-52
- *CADIProfiling::CADIProfileGetExecution()* on page 3-53
- *CADIProfiling::CADIProfileGetMemory()* on page 3-53
- *CADIProfiling::CADIProfileGetTrace()* on page 3-54

- *CADIProfiling::CADIProfileGetRegAccesses()* on page 3-55
- *CADIProfiling::CADIProfileSetRegAccesses()* on page 3-56
- *CADIProfiling::CADIProfileGetMemAccesses()* on page 3-56
- *CADIProfiling::CADIProfileSetMemAccesses()* on page 3-57
- *CADIProfiling::CADIProfileGetAddrExecutionFrequency()* on page 3-57
- *CADIProfiling::CADIProfileSetAddrExecutionFrequency()* on page 3-58
- *CADIProfiling::CADIGetNumberOfInstructions()* on page 3-58
- *CADIProfiling::CADIProfileInitInstructionResultArray ()* on page 3-58
- *CADIProfiling::CADIProfileSetInstructionExecutionFrequency()* on page 3-60
- *CADIProfiling::CADIProfileRegisterResourceAccessCallBack()* on page 3-60
- *CADIProfiling::CADIProfileUnregisterResourceAccessCallBack()* on page 3-60
- *CADIProfiling::CADIProfileRegisterCallBack()* on page 3-60
- *CADIProfiling::CADIProfileUnregisterCallBack()* on page 3-61
- *CADIProfilingCallbacks::profileResourceAccess()* on page 3-61
- *CADIProfilingCallbacks::profileRegisterHazard()* on page 3-61.

Note

This API is for debug profiling (tracing program execution for example). It is not related to the *Cycle Accurate Profiling Interface* (CAPI).

Using the semihosting API

These functions enable the debugger to access files located on the host:

- *CADICallbackObj::appliOpen()* on page 3-62
- *CADICallbackObj::appliInput ()* on page 3-62
- *CADICallbackObj::appliOutput ()* on page 3-63
- *CADICallbackObj::appliClose()* on page 3-63
- *CADICallbackObj::doString()* on page 3-63
- *CADI::CADICaptureSemihosting()* on page 3-63
- *CADI::CADIConsoleGetChannels()* on page 3-64

- *CADI::CADIConsoleNotifyInput()* on page 3-64
- *CADI::CADISemiHostingGetInputChannels()* on page 3-64
- *CADI::CADISemiHostingSendInput()* on page 3-65.

Controlling execution

These functions enable execution control from the callback object:

- *CADICallbackObj::modeChange()* on page 3-65
- *CADICallbackObj::reset()* on page 3-65
- *CADICallbackObj::cycleTick()* on page 3-65 (deprecated)
- *CADICallbackObj::killInterface()* on page 3-66 (deprecated).

Extension functions

This API is reserved for future use. It is intended to provide a mechanism for extending the existing callback functionality:

- *CADICallbackObj::bypass()* on page 3-66
- *CADICallbackObj::lookupSymbol()* on page 3-66
- *CADICallbackObj::refresh()* on page 3-66.

3.2.1 The component CADI class declaration

The class declaration can typically be left as is in the template with the exception of:

- adding any private data members
- changing the parameter in the constructor to the class name of the component.

Figure 3-1 on page 3-3 shows the contents of the header file for a typical generated class (the example has the name `CASI`*`Template`*, the actual name depends on the name of your component).

Note

If your component is a processor, see also the functions that are available in `CADIDisassembler` for controlling and monitoring program execution.

Example 3-2 Header file for a typical CADI component class

```
class CADIMyComponent : public CADI
{
public:
    CADIMyComponent(MyComponentClass* c); // Change names accordingly
    virtual ~CADIMyComponent();
```

```
public:
    // Register access functions
    CADIReturn_t   CADIRegGetGroups(uint32_t groupIndex
        , uint32_t desiredNumOfRegGroups, uint32_t* actualNumOfRegGroups
        , CADIRegGroup_t* reg);
    CADIReturn_t   CADIRegGetMap( uint32_t groupID, uint32_t regIndex
        , uint32_t registerSlots, uint32_t* registerCount, CADIRegInfo_t* reg);
    CADIReturn_t   CADIRegWrite( uint32_t regCount, CADIReg_t* reg
        , uint32_t* numRegsWritten, uint8_t doSideEffects);
    CADIReturn_t   CADIRegRead(uint32_t regCount, CADIReg_t* reg
        , uint32_t* numRegsRead, uint8_t doSideEffects);
    // Memory access functions
    CADIReturn_t   CADIMemGetSpaces( uint32_t spaceIndex, uint32_t memSpaceSlots
        , uint32_t* memSpaceCount, CADIMemSpaceInfo_t* memSpace);
    CADIReturn_t   CADIMemGetBlocks( uint32_t memorySpace, uint32_t blockIndex
        , uint32_t memBlockSlots, uint32_t* memBlockCount
        , CADIMemBlockInfo_t* memBlock);
    CADIReturn_t   CADIMemWrite( CADIAddrComplete_t startAddress
        , uint32_t unitsToWrite, uint32_t unitSizeInBytes, const uint8_t *data
        , uint32_t *actualNumOfUnitsWritten, uint8_t doSideEffects);
    CADIReturn_t   CADIMemRead( CADIAddrComplete_t startAddress
        , uint32_t unitsToRead, uint32_t unitSizeInBytes, uint8_t *data
        , uint32_t *actualNumOfUnitsRead, uint8_t doSideEffects);

    // Access to disassembly class (if available)
    CADIDisassembler* CADIGetDisassembler(void);

private:
    // Pointer to your own component class, change name to match component
    MyComponent_casi *              target;

    // Register related info
    CADIRegInfo_t*                  regInfo;
    CADIRegGroup_t*                 regGroup;

    // Memory related info
    CADIMemSpaceInfo_t*             memSpaceInfo;
    CADIMemBlockInfo_t*             memBlockInfo;
};
```

3.2.2 The CADI class constructor

You can define in the constructor the number of registers you have and the property of your memory spaces.

One change that must be made for every CADI interface is to use the proper component class in the constructor's parameter list to store a local pointer to the component class. This enables accessing the resources of this component when the read/write functions for registers or memory are called.

3.2.3 CADI::CADIXfaceGetFeatures()

The debugger for a target must call this function when it attaches to a target. This function is typically called once per target. The debugger can, however, call it more often if required. This call determines the features supported by the target by updating the passed `features` parameter.

```
virtual CADIReturn_t CADI::CADIXfaceGetFeatures (
            CADITargetFeatures_t * features) =0
```

The caller allocates and de-allocates memory for the *`features`* parameter. For details on the CADITargetFeatures_t structure, see *CADITargetFeatures_t structure* on page 3-78.

3.2.4 CADI::CADIXfaceGetError()

If an error is detected, this routine is called to get the error message.

```
virtual CADIReturn_t CADI::CADIXfaceGetError (uint32_t maxMessageLength,
        uint32_t * actualMessageLength, char * errorMessage) =0
```

where:

`maxMessageLength`

is the max length of `errorMessage` array. The target must not fill more than this number of characters in the array.

`actualMessageLength`

is the max length of `errorMessage` array. The target must set this to the actual number of chars written into the `errorMessage` buffer.

`errorMessage` is the actual error message text. The target writes the text into this character buffer. The length of this buffer is exactly `maxMessageLength`.

3.2.5 CADI::CADIgetDisassembler()

The returns the disassembler for a target.

```
virtual CADIDisassembler * CADI::CADIgetDisassembler (void) =0
```

3.2.6 CADI::CADIXfaceAddCallback()

A debugger connected to the target must call this to establish a callback object that handles asynchronous information from the target. The callback routines must not make calls to the target. It is possible for a debugger to receive a callback while in the middle of a call (for example, receiving a `modeChange` callback from within a `CADIExecStop` call).

Callbacks from a target into the debugger typically come from a different thread (called the simulation thread) than the calls from the debugger into the target (called the GUI thread or debugger thread).

This is the debugger interface that is exposed to a CADI object.

```
virtual CADIReturn_t CADI::CADIXfaceAddCallback (CADICallbackObj * callbackObj,
        char enable[CADI_CB_Count]) =0
```

where:

`callbackObj` is a pointer to the object whose member functions are called as callbacks.

`enable` the elements of this array enable or disable specific callbacks. The caller must always check if the callbacks are enabled. The callbacks must not be called if they are disabled.

The indexes in the array must be based on the list in `CADICallbackType_t`. The length of the array is `CADI_CB_Count`.

3.2.7 CADI::CADIXfaceRemoveCallback()

A debugger must call this to remove any callback objects it has added.

```
virtual CADIReturn_t CADI::CADIXfaceRemoveCallback (
        CADICallbackObj * callbackObj) =0
```

where:

`callbackObj` is a pointer to the callback object. The target must not use this object after this call.

3.2.8 CADI::CADIXfaceBypass()

Targets can have specialized commands that can be requested by the debugger. This command enables the debugger to pass a string containing one of these commands to a target. The target must silently ignore all commands issued through this mechanism and on return set `response` to an empty string and use `CADI_STATUS_UnknownCommand` as the return value.

```
virtual CADIReturn_t CADI::CADIXfaceBypass (uint32_t commandLength,
        const char * command, uint32_t maxResponseLength, char * response) =0
```

where:

`commandLength`
: is the length of (including the terminating `NULL`) command. This helps networked versions of the interface know how much space to allocate for command.

`command`
: is the entire command with all arguments.

`maxResponseLength`
: is the length of the response array. The target must truncate the response to fit it into the array.

`response`
: is the response from the target. This string might or might not be zero terminated. It might also contain binary data for certain commands.

3.2.9 CADI::CADIGetParameterInfo()

Get parameter info class for a specific parameter name.

```
virtual CADIReturn_t CADIGetParameterInfo(const char *parameterName,
                  CADIParameterInfo_t *param) =0
```

where:

`parameterName`

is the name of the parameter to be retrieved. This is the local name in the model, not the global hierarchical name.

`param` points to a single `CADIParameterInfo_t` buffer that must be pre-initialized by the caller and filled with data by the callee.

3.2.10 CADI::CADIGetTargetInfo()

Return target information for this model. The values for the return parameters are set by the model.

```
virtual CADIReturn_t CADI::CADIGetTargetInfo (CADITargetInfo_t targetInfo) =0
```

where:

`targetInfo` is set to point to the `CADITargetInfo_t` structure.

3.2.11 CADI::CADIGetParameterValues()

Return the current parameter values.

```
virtual CADIReturn_t CADI::CADIGetParameterValues(uint32_t parameterCount,
            uint32_t *actualNumOfParamsRead,
            CADIParameterValue_t *paramValuesOut) =0
```

where:

`parameterCount`

is the length of array `paramValuesOut`.

`actualNumOfParamsRead`

is the number of valid entries in `paramValuesOut`. This must be initialized to 0 by the caller.

If an error code is returned and `actualNumOfParamsRead` is greater than 0, the first `actualNumOfParams` entries are valid and caused no error. The entry `paramValuesOut[actualNumOfParamsRead]` caused the error.

`paramValuesOut`

is an output buffer that will hold the parameter values.

3.2.12 CADI::CADIGetParameters()

Get list of supported parameters and parameter details.

```
virtual CADIReturn_t CADI::CADIGetParameters (uint32_t startIndex,
        uint32_t desiredNumOfParams, uint32_t * actualNumOfParams,
        CADIParameterInfo_t * params) =0
```

3.2.13 CADI::CADISetParameters()

Set parameter values.

```
virtual CADIReturn_t CADI::CADISetParameters (uint32_t parameterCount,
        CADIParameterValue_t * parameters,
        CADIFactoryErrorMessage_t * error) =0
```

3.2.14 CADI::CADIRegGetGroups()

This call is used to retrieve register groups from the target.

```
virtual CADIReturn_t CADI::CADIRegGetGroups (uint32_t groupIndex,
        uint32_t desiredNumOfRegGroups, uint32_t * actualNumOfRegGroups,
        CADIRegGroup_t * reg) =0
```

where:

`groupIndex` is the index into the target's list of register groups. It is *not* the group ID's.

`desiredNumOfRegGroups`

is the size of the `reg[]` buffer provided by the caller.

`actualNumOfRegGroups`

on return, is the number of groups returned by the target. If this is less than the number requested, the debugger calls this function again with a different `groupIndex`. Any value set on input is ignored.

`reg` is the register group information. The array is allocated (and deallocated, if applicable) by the caller and filled by the target. The amount of space allocated must be enough to hold the number of groups desired. If the desired count is greater than the targets total number of register groups, the target must return all groups.

3.2.15 CADI::CADIRegGetMap()

The debugger for the target must call this once after connecting to the target. All registers must be reported even if they are part of a compound register. All register numbers must be unique both for registers in the same group and register numbers in other groups.

```
virtual CADIReturn_t CADI::CADIRegGetMap (uint32_t groupID,
        uint32_t startRegisterIndex, uint32_t desiredNumOfRegisters,
        uint32_t * actualNumOfRegisters, CADIRegInfo_t * reg) =0
```

where:

`groupID` identifies the ID of the group whose map is requested. If the value is `CADI_REG_ALLGROUPS`, all registers are returned.

`startRegisterIndex`

is the index into the target's list of registers. It is *not* register numbers.

`desiredNumOfRegisters`

is the total number of registers desired by the caller. The caller must allocate a buffer size that is enough to hold the requested number of registers.

`actualNumOfRegisters`

on return, is the number of registers returned by the target. Any value set on input is ignored.

`reg` is the register information. The array is allocated (and deallocated, if applicable) by the caller to be filled by the target. The amount of space allocated must be enough to hold the number of registers requested. If the count is greater than the targets number of registers, the target must return all the registers.

CADIRegInfo_t

The `CADIRegInfo_t` structure is defined as:

```
struct CADIRegInfo_t
{
  char       name[CADI_NAME_SIZE];
  char       description[CADI_DESCRIPTION];
  uint32_t    regNumber;
  uint32_t    bitsWide;
  int32_t     hasSideEffects;
  CADIRegDetails_t  details;
  CADIRegDisplay_t  display;        //Default is "HEX".
```



```
  CADIRegSymbols_t  symbols;        // For type "symbolic" only.
  CADIRegFloatFormat_t  fpFormat;   // For type "float" only.
  uint32_t    lsbOffset;  // Offset of the least significant bit
                          // relative to bit 0 in the parent
                          // register (or 0 if there is no parent).
  enum { CADI_REGINFO_NO_DWARF_INDEX = 0xffffffff };
  uint32_t   dwarfIndex; // DWARF register index
                          // (CADI_REGINFO_NO_DWARF_INDEX if
                          // register has no DWARF index).
  bool       isProfiled; // Profiling info is available for reg
  bool       isPipeStageField; // Is pipe stage field, also true
                          // for pc and contentInfoRegisterId in
                          // CADIPipeStage_t.
  uint32_t   threadID;   // Thread identifiers. If zero, then not
                          // assigned to a thread.
  CADIRegAccessAttribute_t attribute; // Register access
                         // attributes.
};
```

3.2.16 CADI::CADIRegGetCompound()

This call gets the information about a compound register (as reported by a call to `CADIRegGetMap()`.

```
virtual CADIReturn_t CADI::CADIRegGetCompound (uint32_t reg,
        uint32_t componentIndex, uint32_t desiredNumOfComponents,
        uint32_t * actualNumOfcomponents, uint32_t * components) =0
```

where:

`reg` is the register number.

`componentIndex`

is the index into the this register's (reg) component array.

`desiredNumOfComponents`

is the total number of child registers desired by the caller, starting at `componentRegIndex`.

`actualNumOfcomponents`

on return, is the number of components returned by the target. Any value set on input is ignored.

`components` on return, is the list of component registers. The array is allocated (and deallocated, if applicable) by the caller to be filled by the target. The amount of space allocated must be big enough to hold the number requested. If a target has written less than `regCount` registers it returns the

number of registers successfully written in this field and returns an error code that applies to the register that caused the error (`reg[numOfRegsWritten]`). Any value set on input is ignored.

3.2.17 CADI::CADIRegWrite()

Implementing this function to enable writing registers from debug windows. The implementation of this function is optional.

```
virtual CADIReturn_t CADI::CADIRegWrite (uint32_t regCount, CADIReg_t * reg,
        uint32_t * numOfRegsWritten, uint8_t doSideEffects) =0
```

where:

`regCount` is the number of registers in the reg array.

`reg` is an array of structures each holding the number (as gathered from the `CADIRegGetMap` call) and value of an individual register. The number of bytes allocated for each register is available from the `CADIRegGetMap` call.

`numOfRegsWritten`

on return, is the number of registers that are actually written. Any value set on input is ignored.

`doSideEffects`

indicates whether operation incurs side-effects.

3.2.18 CADI::CADIRegRead()

This function reads register values from the component. This function must be implemented.

```
virtual CADIReturn_t CADI::CADIRegRead (uint32_t regCount, CADIReg_t * reg,
        uint32_t * numRegsRead, uint8_t doSideEffects) =0
```

where:

`regCount` is the number of registers in the reg array.

`reg` is an array of structures each holding the number (as gathered from the `CADIRegGetMap` call) and value of an individual register. The number of bytes allocated for each register is available from the `CADIRegGetMap` call.

`numRegsRead` on return, is the number of registers actually read. It can be less than the number of registers in `reg`. If the value is less than `regCount`, the value specifies the number of registers successfully read and the function returns an error code. Any value set on input is ignored.

`doSideEffects`

indicates whether side effects occur as a result of the operation.

The code in Example 3-3 shows how to set up the CADI access functions in `sc_main()` and test reading a register value:

Example 3-3 CADI register access

```
CADI * cadi1 = s1->getCADI ();
uint32_t actual =0;
CADIRegGroup_t regGroups [2];
cadi1->CADIRegGetGroups (0, 2, & actual, regGroups);
CADIRegInfo_t regs [2];
actual =0;
cadi1->CADIRegGetMap (regGroups [0].groupID, 0, 2, & actual, regs);
CADIReg_t reg;
memset (& reg, 0, sizeof (CADIReg_t));
reg.regNumber = regs [1].regNumber;
actual =0;
cadi1->CADIRegRead (1, & reg, & actual, 0);
printf ("CADI reg 0x%x\n", reg.bytes [0]);
```

3.2.19 CADI::CADIGetPC ()

Returns the PC of the instruction that will be executed next from an ISA perspective.

Reserved for future use.

```
virtual uint64_t CADI::CADIGetPC () =0
virtual uint64_t CADI::CADIGetPC (bool * is_virtual) =0
```

3.2.20 CADI::CADIGetCommittedPCs()

The function returns the number of program counters in the current cycle. This can be used with multi-issue processors.

```
virtual CADIReturn_t CADIGetCommitedPCs (int startIndex, int desiredCount,
                            int * actualCount, uint64_t * pcs =0
```

where:

`startIndex` is the index into the buffer of PCs present in the target.

`desiredCount` is the desired number of PCs.

`actualCount` is the total number of PCs returned by the target.

`pcs` is a list of PCs. The array is allocated (and deallocated, if applicable) by the caller to be filled by the target. This space must be big enough to hold the desired number of spaces.

3.2.21 CADI::CADIMemGetSpaces()

The debugger for the target must call this once after connecting to the target but before accessing any memory. The function identifies the number of independent address spaces available on the target. Use different memory spaces to separate distinct memory areas with overlapping address values (like program and data memory in a Harvard architecture).

```
virtual CADIReturn_t CADI::CADIMemGetSpaces (uint32_t startMemSpaceIndex,
        uint32_t desiredNumOfMemSpaces, uint32_t * actualNumOfMemSpaces,
        CADIMemSpaceInfo_t * memSpaces) =0
```

where:

`startMemSpaceIndex`

is the index into the buffer of memory spaces present in the target.

`desiredNumOfMemSpaces`

is the desired number of memory spaces.

`actualNumOfMemSpaces`

is the total number of memory spaces returned by the target.

`memSpaces` is a list of memory spaces. The array is allocated (and deallocated, if applicable) by the caller to be filled by the target. This space must be big enough to hold the desired number of spaces.

3.2.22 CADI::CADIMemGetBlocks()

The debugger for the target must call this once for each memory space (as indicated by calling the `CADIMemGetSpaces()` function) before accessing memory in that space. This must return the layout of the memory in a single address space. No two blocks with the same parent can overlap. This call returns existing memory blocks only. The caller can assume that any memory that is not in a block is a gap or invalid memory.

```
virtual CADIReturn_t CADI::CADIMemGetBlocks (uint32_t memorySpace,
        uint32_t memBlockIndex, uint32_t desiredNumOfMemBlocks,
        uint32_t * actualNumOfMemBlocks, CADIMemBlockInfo_t * memBlocks) =0
```


where:

`memorySpace` is the memory space for which the caller wants a block list.

`memBlockIndex`

is the index into the target's buffer of memory blocks for the specified memory space.

`desiredNumOfMemBlocks`

is the desired number of memory blocks.

`actualNumOfMemBlocks`

is the is the total number of blocks returned by the target. It might be less than the number requested.

`memBlocks` is a buffer that must be big enough to hold the desired number of `CADIMemBlockInfo_t` structures. Space is allocated (and deallocated, if applicable) by the caller.

3.2.23 CADI::CADIMemRead()

The function reads memory values from the component. This function must be implemented to support the display of memory contents.

```
virtual CADIReturn_t CADI::CADIMemRead (CADIAddrComplete_t startAddress,
        uint32_t unitsToRead, uint32_t unitSizeInBytes, uint8_t * data,
        uint32_t * actualNumOfUnitsRead, uint8_t doSideEffects) =0
```

where:

`startAddress` is the starting address to begin reading from. If startAddress.overlay is `CADI_NO_OVERLAY`, it refers to the current overlay.

`unitsToRead` This is the number of units of size `unitSizeInBytes` to read.

`unitSizeInBytes`

is the unit size of the addresses specified in bytes.

`data` is the data buffer that was allocated by the caller and must be big enough to hold the requested number of addresses. The target data is encoded in little endian format.

`actualNumOfUnitsRead`

is the number of units actually read. It can be less than the number of units requested.

doSideEffects

indicates that any side effects are associated with accessing this memory.

Note

If an error occurs, CADIMemRead() must return the error position in actualNumOfUnits*. Data is assumed valid up to this position.

3.2.24 CADI::CADIMemWrite()

This function writes values to the memory in the target. If support is required for writing to memory from debug windows, CADIMemWrite() must be implemented.

```
virtual CADIReturn_t CADI::CADIMemWrite (CADIAddrComplete_t startAddress,
        uint32_t unitsToWrite, uint32_t unitSizeInBytes, const uint8_t * data,
        uint32_t * actualNumOfUnitsWritten, uint8_t doSideEffects) =0
```

where:

startAddress is the starting address to begin writing from. If startAddress.overlay is CADI_NO_OVERLAY, it refers to the current overlay.

unitsToWrite

is the number of units of size unitSizeInBytes to write.

unitSizeInBytes

is the unit size of the addresses specified in bytes.

data is the data buffer holding the values to be written. This contains target data, encoded in little endian format.

actualNumOfUnitsWritten

is the number of units actually written to target.

doSideEffects

indicates whether operation incurs any side effects associated with accessing this memory.

Note

On error, CADIMemWrite() must return the error position in actualNumOfUnits*. Data is assumed valid up to this position.

If the write spans a gap in the memory space, the target must stop writing a the beginning of the gap and return the number of successful writes in numUnitsWritten.

3.2.25 CADI::CADIMemGetOverlays()

The debugger calls this function to get the list of active overlays. This would typically be done when a breakpoint is hit. When overlays are implemented, an overlay ID must be stored in the symbol table and in the target software. The symbol table must store the starting address, memory space, and byte count for each overlay. This enables the ID to be sent to the host when an overlay occurs.

```
virtual CADIReturn_t CADI::CADIMemGetOverlays (uint32_t activeOverlayIndex,
      uint32_t desiredNumOfActiveOverlays, uint32_t* actualNumOfActiveOverlays,
      CADIOverlayId_t * overlays) =0
```

where:

`activeOverlayIndex`

is the start index into the target's buffer of overlays.

`desiredNumOfActiveOverlays`

is the desired number of overlays.

`actualNumOfActiveOverlays`

is the number of overlay structures returned by the target.

`overlays` is the list of overlays that are currently memory resident (that is, swapped-in). The array is allocated (and deallocated, if applicable) by the caller and filled by the target.

3.2.26 CADI::VirtualToPhysical()

This function translates the virtual address passed as a parameter to a physical address that is the return value.

```
virtual CADIAddrComplete_t CADI::VirtualToPhysical (CADIAddrComplete_t vaddr) =0
```

where:

`vaddr` is the virtual address that is to be converted.

3.2.27 CADI::PhysicalToVirtual()

This function translates the physical address passed as a parameter to a virtual address that is the return value.

```
virtual CADIAddrComplete_t CADI::PhysicalToVirtual (CADIAddrComplete_t paddr) =0
```

where:

`paddr` is the physical address that is to be converted.

3.2.28 CADI::CADIGetCacheInfo()

This call gets the cache information for a memory space.

```
virtual CADIReturn_t CADI::CADIGetCacheInfo (uint32_t memSpaceID,
        CADICacheInfo_t * cacheInfo) =0
```

where:

`memSpaceID` is the memory space.

`cacheInfo` is the cache information.

3.2.29 CADI::CADICacheRead()

This function performs a cache read.

```
virtual CADIReturn_t CADI::CADICacheRead (CADIAddr_t addr, uint32_t linesToRead,
        uint8_t * data, uint8_t * tags, bool * is_dirty, bool * is_valid,
        uint32_t * numLinesRead, bool doSideEffects) =0
```

where:

`addr` is the address to be read, including memspace-id.

`linesToRead`
is the number of cache lines to read.

`data` is a byte array of size (`cache_lines * line_size`). The array is encoded in little endian format.

`tags` is a byte array of size (`cache_lines * tagsbits/8`).

`is_dirty` is the status (one per line).

`is_valid` is the status (one per line).

`numLinesRead`
is the number of cache lines actually read.

3.2.30 CADI::CADICacheWrite ()

This function performs a cache write.

```
virtual CADIReturn_t CADI::CADICacheWrite (CADIAddr_t addr,
      uint32_t linesToWrite, const uint8_t * data, const uint8_t * tags,
      const bool * is_dirty, const bool * is_valid, uint32_t * numLinesWritten,
      bool doSideEffects) =0
```

where:

`addr` is the address to be written, including memspace-id.

`linesToWrite`
: is the number of cache lines to write.

`data` is a byte array of size (`cache_lines * line_size`). The array is encoded in little endian format.

`tags` is a byte array of size (`cache_lines * tagsbits/8`).

`is_dirty` is status (one per line).

`is_valid` is status (one per line).

`numLinesWritten`
: is the number of cache lines actually written.

`doSideEffects`
: selects write through if `true`.

3.2.31 CADI execution modes

The execution APIs modify the execution state of the target.

These functions typically return before the target completes the requested action. For example, a run or even a single step returns before the target stops. The debugger is notified by the callback about the completion of the request.

The exec mode calls enable extensions to the typical execution modes of *run* and *breakpoint*. If a target does not have other modes, these calls are redundant and are typically not used.

The execution mode array requires that certain modes have specific positions in the array (see *CADI_EXECMODE_t enumeration* on page 3-90).

3.2.32 CADI::CADIExecGetModes()

Many processors have more than just a run state and a breakpoint state. This call allows the debugger to determine what these states are.

```
virtual CADIReturn_t CADI::CADIExecGetModes (uint32_t startModeIndex,
        uint32_t desiredNumOfModes, uint32_t * actualNumOfModes,
        CADIExecMode_t * execModes) =0
```

where:

`startModeIndex`

is the index into the target's buffer of execution modes.

`desiredNumOfModes`

is the requested number of modes This call can be repeated with index + count as the index.

`actualNumOfModes`

is the number of modes returned by the target.

`execModes` is the space. The caller allocates (and, if applicable, deallocates) space. The number of elements must be the same as `actualNumOfModes`. The mode values are listed in *CADI_EXECMODE_t enumeration* on page 3-90.

3.2.33 CADI::CADIExecGetResetLevels()

Many targets have more than just one reset level. This call allows the debugger to determine what these levels are.

```
virtual CADIReturn_t CADI::CADIExecGetResetLevels (uint32_t
startResetLevelIndex,
        uint32_t desiredNumOfResetLevels, uint32_t * actualNumOfResetLevels,
        CADIResetLevel_t * resetLevels) =0
```

where:

`startResetLevelIndex`

is the index into the target's buffer of reset levels.

`desiredNumOfResetLevels`

is the number of levels desired by the caller. It is the number of reset levels desired by the caller.

`actualNumOfResetLevels`
: is the total number returned.

`resetLevels`
: is the caller allocated space. The number of elements must be the same as the `actualNumOfResetLevels`. These must be in the order of most severe (at reset level zero) to least severe.

3.2.34 CADI::CADIExecSetMode()

This sets the execution modes for the target. This call returns immediately, possibly before the target execution mode has been reached. The mode values are listed in *CADI_EXECMODE_t enumeration* on page 3-90

```
virtual CADIReturn_t CADI::CADIExecSetMode (uint32_t execMode) =0
```

This call is, for a subset of the execution modes, redundant with other APIs:

- A call to `CADIExecSetMode(CADI_EXECMODE_Run)` is equivalent to a call to `CADIExecContinue()`.
- A call to `CADIExecSetMode(CADI_EXECMODE_Stop)` is equivalent to a call to `CADIExecStop()`.

Note

`execMode` must be less than the value `nrExecModes` received by `CADIXfaceGetFeatures()`.

3.2.35 CADI::CADIExecGetMode()

This call enables the debugger to determine the execution state of the target.

```
virtual CADIReturn_t CADI::CADIExecGetMode (uint32_t * execMode) =0
```

Note

`execMode` corresponds to the value `nrExecModes` received by `CADIXfaceGetFeatures()`.

3.2.36 CADI::CADIExecSingleStep()

This function returns immediately and a separate notification informs the debugger that the execution state has changed. Typically this call results in the `modeChange()` callback (if enabled) for `CADI_EXECMODE_Run` followed by `CADI_EXECMODE_Stop`.

```
virtual CADIReturn_t CADI::CADIExecSingleStep (uint32_t instructionCount,
        int8_t stepCycle, int8_t stepOver) =0
```

where:

`instructionCount`
: is the number of instructions requested. Some targets can not step a specific number of instructions safely (into a delay slot, for example). The target can step more instructions so that it stops at a safe place.

`stepCycle`
: specifies (for targets that have exposed multiple pipe stages) whether the step merely clocks the device (`stepCycle` == yes) or flushes the pipe (`stepCycle` == no).

  For other kinds of targets, this argument is ignored (`stepCycle` = no is assumed).

`stepOver`
: allows the target to handle stepping over a call. It is especially useful on an emulator with no available breakpoints and the target must step until the call returns or a breakpoint is hit.

Note

Because this call returns immediately, the return value indicates whether the target believes that it can perform the operation and not whether the operation was completed successfully.

3.2.37 CADI::CADIExecReset()

Upon receipt of this call, the target:

- resets its execution related internal state
- resets its registers to their initial state
- does not change breakpoints or callbacks.

This call provides a simulation level reset.

```
virtual CADIReturn_t CADI::CADIExecReset (uint32_t resetLevel) =0
```

Note

resetLevel must be one of the numbers provided in the `resetLevels` array received by `CADIExecGetResetLevels()`.

3.2.38 CADI::CADIExecContinue()

This function returns immediately and a separate notification from the `modeChange(RUN)` callback informs the debugger when the execution state changes. The simulation runs asynchronously in a separate thread

```
virtual CADIReturn_t CADI::CADIExecContinue (void) =0
```

Note

Because this call returns immediately, the return value indicates whether the target believes that it can perform the operation and not whether the operation was completed successfully.

3.2.39 CADI::CADIExecStop()

This causes the target's execution to stop. The function returns immediately and the target might still running when the function returns. A debugger must wait for a `modeChange(STOP)` callback to ensure that the simulation has ended.

```
virtual CADIReturn_t CADI::CADIExecStop (void) =0
```

3.2.40 CADI::CADIExecGetExceptions()

This gets the list of the target's exception vectors.

```
virtual CADIReturn_t CADI::CADIExecGetExceptions (uint32_t startExceptionIndex,
        uint32_t desiredNumOfExceptions, uint32_t * actualNumOfExceptions,
        CADIException_t * exceptions) =0
```

where:

`startExceptionIndex`

is the index into the targets list of exceptions.

`desiredNumOfExceptions`

is the number of slots in the exception array. The target must not fill more than this number of characters in the array.

`actualNumOfExceptions`

is the total number of returned exceptions. If this is less than desired count, the call can be repeated with a different set of parameters.

`exceptions` is list of exceptions. The array is allocated (and deallocated, if applicable) by the caller to be filled by the target. This buffer must be big enough to hold the desired count of exceptions.

3.2.41 CADI::CADIExecAssertException()

Raise an exception.

```
virtual CADIReturn_t CADI::CADIExecAssertException (uint32_t exception,
        CADIExceptionAction_t action) =0
```

The definition of `CADIExceptionAction_t` is:

```
typedef enum CADIExceptionAction_t
{
  CADI_EXCEPTION_Raise, // For targets that can raise an exception ...
  CADI_EXCEPTION_Lower, // ... and leave it raised
  CADI_EXCEPTION_Pulse,
  CADI_EXCEPTION_ENUM_MAX,
} CADIExceptionAction_t;
```


3.2.42 CADI::CADIExecGetPipeStages()

This is used to expose the pipe simulated inside of a cycle-accurate simulation.

```
virtual CADIReturn_t CADI::CADIExecGetPipeStages (uint32_t startPipeStageIndex,
        uint32_t desiredNumOfPipeStages, uint32_t * actualNumOfPipeStages,
        CADIPipeStage_t * pipeStages) =0
```

where:

`startPipeStageIndex`

is the index into the target's list of pipe stages.

`actualNumOfPipeStages`

is the number of stages actually returned to the caller.

`pipeStages` is the list of pipestages in order of execution for a single instruction. `pipestage[0]` must contain the first stage executed for any single instruction. The array is allocated (and deallocated, if applicable) by the caller to be filled by the target.

`desiredNumOfPipeStages`

is the number of spaces available to fill. The target must not fill more than this number of elements in the pipestage array.

3.2.43 CADI::CADIExecGetPipeStageFields()

Reserved for future use.

```
virtual CADIReturn_t CADI::CADIExecGetPipeStageFields (
        uint32_t startPipeStageFieldIndex, uint32_t desiredNumOfPipeStageFields,
        uint32_t * actualNumOfPipeStageFields,
        CADIPipeStageField_t * pipeStageFields) =0
```

3.2.44 CADI::CADIExecLoadApplication()

This is used to load an application file to a processor in the simulation.

```
virtual CADIReturn_t CADI::CADIExecLoadApplication(const char *filename,
                     bool loadData, bool verbose, const char *parameters) =0
```

where:

`filename`

is the name of the application file.

`loadData`

loads data and symbols if true, if false only load symbols. The target decides whether or not it can load symbols.

`verbose` prints verbose messages while loading a file if true. The target decides whether or not it output messages.

`parameters`

if not NULL, this is the command line parameters to be passed to the loaded application.

3.2.45 CADI::CADIExecUnLoadApplication()

This is used to symbol information of a specific image that was loaded previously.

```
virtual CADIReturn_t CADI::CADIExecUnloadApplication(const char *filename) =0
```

where:

`filename`

is the same as was specified for `CADIExecLoadImage()`.

3.2.46 CADI::CADIExecGetLoadedApplication()

This gets a list of image filenames that are currently loaded in the target.

```
virtual CADIReturn_t CADI::CADIExecGetLoadedApplications(uint32_t startIndex,
                uint32_t desiredNumberOfApplications,
                uint32_t *actualNumberOfApplicatiosReturnedOut,
                char *filenamesOut, uint32_t filenameLength,
                char *parametersOut, uint32_t parametersLength) =0
```


where:

`startIndex`

is the starting index in the list of filenames.

`desiredNumberOfApplications`

is the desired number of applications (filename + parameters).

`actualNumberOfApplicatiosReturnedOut`

is the number of applications (filenames + parameters) that are valid in `filenamesOut` and `parametersOut`.

`filenamesOut`

is a buffer of length `[desiredNumberOfFilenames * filenamLength]`, the N^{th} filename returned starts at offset `N*filenameLength`.

`filenameLength`

is the maximum length of a single filename including terminating 0, filenames which are longer are truncated. All returned filenames must always be 0 terminated. If one of the returned filenames has the length `filenameLength-1` then `filenameLength` was too short and must be redone. The target decides whether or not it can keep information of more than one file.

`parametersOut`

is a buffer of length `[desiredNumberOfApplications * parametersLength]`, the N^{th} parameter returned starts at offset `N*parametersLength`. The target decides whether or not it can keep information of more than one file.

`parametersLength`

is the maximum length of a single parameters string including terminating 0, parameters which are longer are truncated. All returned parameters must always be 0 terminated. If one of the returned parameters has the length `parametersLength-1` then `parametersLength` was too short and must be redone. The target decides whether or not it can keep information of more than one file.

3.2.47 CADI::CADIExecSetApplication()

Reserved for future use.

```
virtual CADIReturn_t CADI::CADIExecSetApplication (
        const std::string & fileName) =0
```

3.2.48 CADI::CADIGetInstructionCount()

This method gets the current instruction count of the specific target that this interface is connected to.

```
virtual CADIReturn_t CADI::CADIGetInstructionCount (
        uint64_t & instructionCount) =0
```

where:

`instructionCount`

is the returned instruction count.

3.2.49 CADI::CADIGetCycleCount()

Gets the current cycle count.

```
virtual CADIReturn_t CADI::CADIGetCycleCount (uint64_t & cycleCount,
        bool systemCycles) =0
```

where:

`cycleCount`

is the returned cycle count. This must be pre-initialized by the caller and assigned by the callee.

`systemCycles`

if true, the method returns the system cycle count. If false, the method returns return the target specific cycle count.

3.2.50 CADI::CADIBptGetList()

If the debugger attaches to a target that already has breakpoints set, this enables the debugger to identify the breakpoints.

```
virtual CADIReturn_t CADI::CADIBptGetList (uint32_t startIndex,
        uint32_t desiredNumOfBpts, uint32_t * actualNumOfBpts,
        CADIBptDescription_t * breakpoints) =0
```

where:

`startIndex` is the index into the target's buffer of breakpoints.

`desiredNumOfBpts`

is the desired number of breakpoints.

`actualNumOfBpts`

is the number of breakpoints that are returned in the buffer.

`breakpoints`

is the array of breakpoints that are returned in the buffer that was allocated by the caller.

The elements are of type `CADIBptDescription_t` and defined as:

```
typedef struct CADIBptDescription_t
{
  CADIBptNumber_t     bptNumber; // The breakpoint number.
  CADIBptRequest_t    bptInfo;   // The breakpoint information
                                 // (address, condition, etc.).
} CADIBptDescription_t;
```

3.2.51 CADI::CADIBptRead()

Read a breakpoint for a specific breakpoint ID. This can be used to retrieve the current `ignoreCount` of a specific breakpoint.

```
virtual CADIReturn_t CADIBptRead(CADIBptNumber_t breakpointId,
               CADIBptRequest_t *requestOut) = 0;
```

where:

`brealpointId` is the breakpoint ID of the breakpoint to be read.

`requestOut` is the buffer for a single breakpoint.

3.2.52 CADI::CADIBptSet()

This sets a, possibly complex, code breakpoint in the target.

```
virtual CADIReturn_t CADI::CADIBptSet (CADIBptRequest_t * request,
            CADIBptNumber_t * breakpoint) =0
```

where:

`request` is the requested breakpoint.

`breakpoint` is the resulting breakpoint (zero if the breakpoint was not set).

The `CADIBptNumber_t` is defined as `uint32_t`.

3.2.53 CADI::CADIBptClear()

This function removes a breakpoint from the target.

```
virtual CADIReturn_t CADI::CADIBptClear (CADIBptNumber_t breakpointId) =0
```

where:

`breakpointId`
: is the requested breakpoint.

3.2.54 CADI::CADIBptConfigure()

This function enables or disables a breakpoint on the target. This only applies if the target supports enabling and disabling of hardware breakpoints. Otherwise, this type of breakpoint management must be done in the host.

```
virtual CADIReturn_t CADI::CADIBptConfigure (CADIBptNumber_t breakpointId,
        CADIBptConfigure_t configuration) =0
```

where:

`breakpointId`
: is the requested breakpoint.

`configuration`
: is the requested configuration.

3.3 The CADIDisassembler class

If the component supports disassembly, the disassembly API can be used to display the disassembly during a simulation.

The CADI class provides the function CADIGetDisassembler(). It is called by the simulation environment to obtain a pointer to the disassembly API.

Note

A program memory space must exist to use the disassembly feature.

Example 3-4 CADIDisassembler class

```
class CADIDisassembler  : public CAInterface
{
public:
    static if_name_t IFNAME() { return "eslapi.CADIDisassembler2"; }
    static if_rev_t IFREVISION() { return 0; }

    // Two types: distinguish standard and history type
    virtual CADIDisassemblerType getType() const =0;

    // mode handling functions
    virtual uint32_t getModeCount() const =0;
    virtual void getModeNames(CADIDisassemblerCB *callback) =0;
    virtual uint32_t getCurrentMode() =0;

    // set disassembly callback for this object
    virtual void registerModeChangeCB(CADIDisasmModeChangeCB* cb);

   virtual CADIDisassemblerStatus GetSourceReferenceForAddress(
        CADIDisassemblerCB *callback, const CADIAddr_t &address) =0;

    virtual CADIDisassemblerStatus getAddressForSourceReference(
        const char *sourceFile, uint32_t sourceLine, CADIAddr_t &address) =0

    //    function for standard type disassembly
    virtual CADIDisassemblerStatus getDisassembly(CADIDisassemblerCB *callback,
        const CADIAddr_t &address, CADIAddr_t &nextAddr, const uint32_t mode,
        uint32_t desiredCount = 1) =0;

    // Query if an instruction is a call instruction
    virtual CADIDisassemblerStatus GetInstructionType(const CADIAddr_t &address,
        CADIDisassemblerInstructionType &insn_type) =0;
```

```
    // A default minimum implementation, to provide backwards-compatibility.
    virtual CAInterface * ObtainInterface(if_name_t ifName, if_rev_t minRev,
         if_rev_t *   actualRev)
        {
            if((strcmp(ifName,IFNAME()) == 0) && (minRev <= IFREVISION()))
           {
                if (actualRev) // make sure this is not a NULL pointer
                {
                    *actualRev = IFREVISION();
                }
               return this;
            }
            if((strcmp(ifName, CAInterface::IFNAME()) == 0) &&
                minRev <= CAInterface::IFREVISION())
            {
                if (actualRev != NULL)
                {
                    *actualRev = CAInterface::IFREVISION();
                }
                return this;
            }
            return NULL;
        }

};
```

3.3.1 CADIDisassembler::getType()

The return value indicates whether the type is standard or historical.

```
virtual CADIDisassemblerType CADIDisassembler::getType () const =0
```

3.3.2 CADIDisassembler::getModeCount()

The return value from this function indicates support for multiple modes (for example 32bit or 16bit mode). Valid modes start at 1. Mode 0 indicates no modes or *don't care*.

```
virtual uint32_t CADIDisassembler::getModeCount () =0
```

3.3.3 CADIDisassembler::getModeNames()

This function returns the name of all modes. A call is triggered to `CADIDisassemblerCB::ReceiveModeName()` once for every mode.

```
virtual std::string CADIDisassembler::getModeNames (
     CADIDisassemblerCB *callback) =0
```


3.3.4 CADIDisassembler::getCurrentMode()

The return value indicates the current execution mode. If modes are not supported by this target, the return value is 0. If modes are supported, the return value is between 0 and the value returned by `GetModeCount()`.

```
virtual uint32_t CADIDisassembler::getCurrentMode() = 0
```

3.3.5 CADIDisassembler::getSourceReferenceForAddress()

The return value indicates source-level information. It triggers a call to `CADIDisassemblerCB::ReceiveSourceReference()`.

```
virtual CADIDisassemblerStatus CADIDisassembler::GetSourceReferenceForAddress (
       CADIDisassemblerCB *callback, const CADIAddr_t &address) = 0
```

where:

`address` is the requested address.

`callback` is the callback function to retrieve the disassembly.

3.3.6 CADIDisassembler::getAddressForSourceReference()

The return value indicates the first address corresponding to that generated for the given source line.

```
virtual CADIDisassemblerStatus CADIDisassembler::getAddressForSourceReference(
            const char *sourceFile, uint32_t sourceLine,
            CADIAddr_t &address) = 0
```

where:

`sourceLine` is the source line number.

`sourceFile` is the source file name is returned as the first element of the 1-sized array.

`address` is set to the address corresponding to the source line.

3.3.7 CADIDisassembler::getDisassembly()

Function enables standard type disassembly.

```
virtual CADIDisassemblerStatus CADIDisassembler::getDisassembly (
            CADIDisassemblerCB *callback, const CADIAddr_t &address,
            CADIAddr_t &nextAddr, const uint32_t mode,
            uint32_t desiredCount = 1) = 0
```

where:

`address` passes the address of the instruction to disassemble and to return the address of the next valid instruction. Mandatory also if return value is `CADIDISASM_NO_INSTRUCTION` or `CADIDISASM_ILLEGAL_ADDRESS`.

`nextAddr` returns the address of the next instruction. This must be used if the return value is `CADIDISASM_NO_INSTRUCTION` or `CADIDISASM_ILLEGAL_ADDRESS`.

`nextAddr` must be a hint to the next address that might result in successful disassembly.

`callback` is the callback function to retrieve the disassembly.

`mode` contains the execution mode. If 0, use the current execution mode.

`desiredCount` can be used to disassemble a sequence of instructions. Up to *`desiredCount`* calls are made to `CADIDisassemblerCB::ReceiveDisassembly()`.

The first instruction is the instruction pointed to by *`address`*. The sequence of disassembled instructions stops if there is an error (no instruction or illegal address) occurs while attempting to disassemble an instruction

3.3.8 CADIDisassembler::GetInstructionType()

This method determines whether the instruction is a call instruction.

```
virtual CADIDisassemblerStatus GetInstructionType(const CADIAddr_t &address,
                  CADIDisassemblerInstructionType &insn_type) = 0
```

where:

`address` is used to pass the address of the instruction to check if it is a call.

`insn_type` is true if the instruction is a call instruction.

3.3.9 CADIDisassembler::ObtainInterface()

This is a default minimum implementation, to provide backwards-compatibility with legacy code. This implementation assumes that there will be no other interfaces implemented on the component providing `CADIDisassembler`.

```
virtual CAInterface * ObtainInterface(if_name_t ifName, if_rev_t minRev,
                        if_rev_t *   actualRev)
```

See `CADIDisassembler.h` for implementation details.

3.3.10 The CADIDisassemblerCB class

This callback class must be implemented by the disassembly frontend.

```
class CADI_WEXP CADIDisassemblerCB : public CAInterface
{
public:
    // Return the CAInterface name for this interface.
    static if_name_t IFNAME() { return "eslapi.CADIDisassemblerCB2"; }
    static if_rev_t IFREVISION() { return 0; }
    virtual void ReceiveModeName(uint32_t mode, const char *modename) =0;
    virtual void ReceiveSourceReference(const CADIAddr_t &addr, const char
            *sourceFile, uint32_t sourceLine) =0;
     virtual void ReceiveDisassembly(const CADIAddr_t &addr,
     const char *opcodes, const char *disassembly) =0;
};
```

3.3.11 CADIDisassemblerCB::ReceiveModeName()

This callback is triggered by `CADIDisassembler::GetModeNames()`.

```
virtual void ReceiveModeName(uint32_t mode, const char *modename) =0) = 0
```

where:

`mode` is the required mode.

`modename` returns the mode name string.

3.3.12 CADIDisassemblerCB::ReceiveSourceReference()

This callback is triggered by `CADIDisassembler::GetSourceReferenceForAddress(...)`.

```
virtual ReceiveSourceReference(const CADIAddr_t &addr, const char *sourceFile,
        uint32_t sourceLine) = 0
```

where:

`addr` is the address in the code.

`sourceFile` is the source code text.

`sourceline` is the line in the source that corresponds to the code at `addr`.

3.3.13 CADIDisassemblerCB::ReceiveDissassembly()

This callback is triggered by `CADIDisassembler::GetDisassembly(...))`.

```
virtual void ReceiveDisassembly(const CADIAddr_t &addr, const char *opcodes,
         const char *disassembly) = 0
```

where:

`addr` is the address in the code.

`opcodes` is the opcode text for the disassembly instruction.

`disassembly` is the text for the disassembly.

3.4 The CADIProfiling class

This class enables you to record and monitor profile information related to the debugging session.

3.4.1 CADIProfiling::CADIProfileSetup()

This informs the target of the memory regions that are to be added. This function must be called once before any number of calls to either of the following:

- `CADIProfileControl(CADI_PROF_CNTL_Start)`
- `CADIProfileControl(CADI_PROF_CNTL_Stop)`.

```
virtual CADIReturn_t CADIProfiling::CADIProfileSetup (CADIProfileType_t type,
              uint32_t regionCount, CADIProfileRegion_t * region) =0
```

where:

`type` is the type of profiling (execution addresses or data access) to which these regions apply:

- `CADI_PROF_TYPE_Execution`
- `CADI_PROF_TYPE_Memory` is used with `CADIProfileGetMemory()`
- `CADI_PROF_TYPE_Trace` is used with `CADIProfileGetTrace()`.

`regionCount` is the number of regions.

`region` contains the memory areas being added. The caller allocates all memory.

The return value must be `CADI_STATUS_IllegalArgument` if any of the following are true:

- any region spans unpopulated memory
- any region spans illegal memory
- any region overlaps another region
- the address space of a region is not consistent with the profiling type.

The definition of `CADIProfileRegion_t` is:

```
typedef struct CADIProfileRegion_t
{
  int               addressesAreValid;
  CADIOverlayId_t   overlay;
  CADIMemSpace_t    memorySpace;
  CADIAddrSimple_t  start;
  CADIAddrSimple_t  finish;
} CADIProfileRegion_t;
```

3.4.2 CADIProfiling::CADIProfileControl ()

This starts, stops, or resets profiling by passing a member of the CADIProfileControl_t enumeration.

```
virtual CADIReturn_t CADIProfiling::CADIProfileControl(
        CADIProfileControl_t control) =0
```

where:

control — defines profiling behavior. The CADIProfileControl_t enumeration values are:

- CADI_PROF_CNTL_Start
- CADI_PROF_CNTL_Stop
- CADI_PROF_CNTL_Reset (stop and then restart immediately).

Note

Starting profiling resets any information that was saved. Stopping profiling does not reset recorded information.

3.4.3 CADIProfiling::CADIProfileTraceControl ()

This starts, stops, and resets recording the execution trace.

```
virtual CADIReturn_t CADIProfiling::CADIProfileTraceControl (
        CADITraceBufferControl_t bufferArg, CADITraceControl_t control,
        CADITraceOverlayControl_t overlay) =0
```

where:

bufferArg — is the buffer control.

control — is the action to take when the buffer fills. The action might only be valid for some types of trace. The enumerated values are:

- CADI_TRACE_CNTL_StartContinuous
- CADI_TRACE_CNTL_StartDiscontinuity
- CADI_TRACE_CNTL_Stop.

overlay — selects overlay:

- If CADI_TRACE_OVERLAY_Memory, overlay events must be included in the trace output at the expense of not being able to see inside the trace manager.

- If CADI_TRACE_OVERLAY_Manager, the trace data must include the code in the overlay manager code at the expense of not knowing the details of memory regions that are overlaid.

3.4.4 CADIProfiling::CADIProfileGetExecution()

This gets the results of a profiling session for executable code.

If called before profiling is stopped or before a legal set of regions have been established, this call must return CADI_STATUS_GeneralError.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetExecution (
        CADIProfileResultType_t * type, uint32_t regIndex, uint32_t regionSlots,
        uint32_t * regionCount, CADIProfileResults_t * region) =0
```

where:

type — indicates whether percentage statistics or an absolute count is being returned.

regIndex — is the index into the target's buffer.

regionSlots — is the number of spaces requested to be filled. The target shall not fill more than this number of elements in the region array.

regionCount — is the actual number of regions setup by CADIProfileSetup plus one. The additional count indicates the other category.

region — corresponds to the regions setup by CADIProfileSetup. The array is allocated (and deallocated, if applicable) by the caller and filled by the target.

3.4.5 CADIProfiling::CADIProfileGetMemory()

This gets the results of a profiling session for memory accesses. If called before profiling is stopped or before a legal set of profiling regions has been established, the return value must be CADI_STATUS_GeneralError.

CADIProfileGetMemory() is similar to CADIProfileGetExecution() and is provided to enable future versions to separately modify the call signatures of the two functions.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetMemory (
        CADIProfileResultType_t * type, uint32_t regIndex, uint32_t regionSlots,
        uint32_t * regionCount, CADIProfileResults_t * region) =0
```

where:

`type` tells the caller whether percentage statistics or an absolute count is being returned.

`regIndex` is the index into the target's buffer.

`regionSlots` is the number of spaces requested to be filled. The target shall not fill more than this number of elements in the region array.

`regionCount` is the actual number of regions setup by `CADIProfileSetup` plus one. The additional count indicates the `other` category.

`region` corresponds to the regions setup by `CADIProfileSetup`. The array is allocated (and deallocated, if applicable) by the caller and filled by the target.

3.4.6 CADIProfiling::CADIProfileGetTrace()

This gets the results of a trace session. The `block` parameter contains the PC values that have been executed by the target.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetTrace (uint32_t blockIndex,
        uint32_t blockSlots, uint32_t * blockCount, CADITraceBlock_t * block) =0
```

where:

`blockIndex` is the start index of the trace block.

`blockCount` is the number of samples being returned.

`block` is the list of executed addresses and overlay events in time sequential order. The blocks in the array must be sorted by time executed and `block[0]` must contain the most recently executed address or event. If multiple program memory spaces exist (and execution uses multiple spaces during execution), separate blocks must exist for each memory space. The `block` array is allocated (and deallocated, if applicable) by the caller and filled in by the target.

`blockSlots` is the number of spaces available to fill. The target must not fill more than this number of elements in the `block` array.

CASITraceBlock_t and CASIAddr_t are defined as:

```
typedef struct CASITraceBlock_t
{
  CASITraceBlockType_t blockType;
  union
  {
    CASIAddr_t       address;
    CASIOverlayId_t  overlay;  //uint32_t
  } u;
} CASITraceBlock_t;


typedef struct CASIAddr_t
{
  CASIMemSpace_t   space; // Numeric designation of the memory space (uint32_t)
  CASIAddrSimple_t addr;// The actual memory address (uint32_t)
} CASIAddr_t;
```

3.4.7 CADIProfiling::CADIProfileGetRegAccesses()

Reads the number of read/write accesses for *numberOfRegs* registers, starting with register index *startReg*.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetRegAccesses (
        uint32_t startRegID, uint32_t numberOfRegs,
        CADIRegProfileResults_t * reg, uint32_t & actualNumberOfRegs) =0
```

where:

`startRegID` is the index of the first register.

`NumberOfRegs` is the number of registers the profiling data is requested for.

`reg` on return, this contains the results.

Note

reg must point to an array of objects of type `CADIResourceProfileResults_t` with size *numberOfRegs*.

`actualNumberOfRegs`

on return, this contains the number of registers the profiling data was actually read for.

3.4.8 CADIProfiling::CADIProfileSetRegAccesses()

Writes the number of read/write accesses for *numberOfRegs* registers according to values saved in *reg*, starting with register index *startReg*.

```
virtual CADIReturn_t CADIProfiling::CADIProfileSetRegAccesses (
        uint32_t startRegID, uint32_t numberOfRegs,
        CADIRegProfileResults_t * reg, uint32_t & actualNumberOfRegs) =0
```

where:

`startRegID` is the index of the first register.

`NumberOfRegs` is the number of registers the profiling data.

`reg` contains the results on return.

Note

reg must point to an array of objects of type `CADIResourceProfileResults_t` with size *numberOfRegs*.

`actualNumberOfRegs`

contains the number of updated registers.

3.4.9 CADIProfiling::CADIProfileGetMemAccesses()

Reads the number of read/write accesses for *numberOfRegs* memory units.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetMemAccesses (
        CADIAddrComplete_t startAddress, uint32_t numberOfUnits,
        CADIMemProfileResults_t * mem, uint32_t & actualNumberOfUnits) =0
```

where:

`startAddress` is the starting address for the memory units.

`NumberOfUnits` is the number of memory units.

`mem` contains the results on return.

Note

mem must point to an array of objects of type `CADIResourceProfileResults_t` with size *numberOfUnits*.

`actualNumberOfUnits`

contains the number of memory units for which data was collected.

3.4.10 CADIProfiling::CADIProfileSetMemAccesses()

Writes the number of read/write accesses for *numberOfUnits* memory units according to values saved in *mem*.

```
virtual CADIReturn_t CADIProfiling::CADIProfileSetMemAccesses (
        CADIAddrComplete_t startAddress, uint32_t numberOfUnits,
        CADIMemProfileResults_t * mem, uint32_t & actualNumberOfUnits) =0
```

where:

`startAddress` is the starting address for the memory units.

`NumberOfUnits`

is the number of memory units.

`mem` contains the values to use for the update.

Note

mem must point to an array of objects of type `CADIResourceProfileResults_t` with size *numberOfUnits*.

`actualNumberOfUnits`

contains the number of memory units for which data was collected.

3.4.11 CADIProfiling::CADIProfileGetAddrExecutionFrequency()

Reads the execution frequency for *numberOfAddr* disassembly addresses.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetAddrExecutionFrequency (
        uint64_t startAddr, uint32_t numberOfAddr, uint64_t * freq,
        uint32_t & actualNumberOfAddr) =0
```

where:

`startAddr` is the starting address for the memory units.

`numberOfAddr`

is the number of memory units.

`freq` contains the results.

Note

freq must point to an array of `uint32_t` with size *numberOfAddr*.

`actualNumberOfAddr`

contains the number of addresses for which the frequency was read.

3.4.12 CADIProfiling::CADIProfileSetAddrExecutionFrequency()

Writes the execution frequency for `numberOfAddr` disassembly addresses according to values saved in `freq`.

```
virtual CADIReturn_t CADIProfiling::CADIProfileSetAddrExecutionFrequency (
        uint64_t startAddr, uint32_t numberOfAddr, uint64_t * freq,
        uint32_t & actualNumberOfAddr) =0
```

where:

`startAddr` is the starting address for the memory units.

`numberOfAddr`

is the number of memory units.

`freq` contains the values to use to update the disassembly addresses.

Note

`freq` must point to an array of `uint32_t` with size *`numberOfAddr`*.

`actualNumberOfAddr`

contains the number of addresses updated.

3.4.13 CADIProfiling::CADIGetNumberOfInstructions()

Returns number of instructions of the target.

```
virtual uint32_t CADIProfiling::CADIGetNumberOfInstructions () =0
```

3.4.14 CADIProfiling::CADIProfileInitInstructionResultArray ()

This method prepares given array instructions by setting FID, name and `pathToInstructionInLISASource`.

```
virtual CADIReturn_t CADIProfiling::CADIProfileInitInstructionResultArray (
        uint32_t numberOfInstructions,
        CADIInstructionProfileResults_t * instructions,
        uint32_t & actualNumberOfInstructions) =0
```


where:

`numberOfInstructions`

is the desired number of array entries to be prepared.

`instructions` contains the values to use to update.

`actualNumberOfInstructions`

is the number of array entries actually prepared.

The definition of `CADIInstructionProfileResults_t` is:

```
typedef struct CADIInstructionProfileResults_t
{
  uint32_tFID;
  char        name[CADI_DESCRIPTION];
  char        pathToInstructionInLISASource[1024];
  uint64_texecutionCount;
} CADIInstructionProfileResults_t;
```

3.4.15 CADIProfiling::CADIProfileGetInstructionExecutionFrequency()

Reads the execution counts for *`numberOfInstructions`* instructions by setting the appropriate *`executionCount`* entry in array instructions.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetInstructionExecutionFrequency(
        uint32_t numberOfInstructions,
        CADIInstructionProfileResults_t * instructions,
        uint32_t & actualNumberOfInstructions) =0
```

where:

`numberOfInstructions`

is the desired number of array entries to read.

`instructions` contains the results.

`actualNumberOfInstructions`

is the number of array entries actually read.

3.4.16 CADIProfiling::CADIProfileSetInstructionExecutionFrequency()

Writes the execution counts for `numberOfInstructions` instructions according to values in *instructions*.

```
virtual CADIReturn_t CADIProfiling::CADIProfileSetInstructionExecutionFrequency(
        uint32_t numberOfInstructions,
        CADIInstructionProfileResults_t * instructions,
        uint32_t & actualNumberOfInstructions) =0
```

where:

`numberOfInstructions`

is the desired number of array entries to write.

`instructions` contains the values to write.

`actualNumberOfInstructions`

is the number of array entries actually written.

3.4.17 CADIProfiling::CADIProfileRegisterResourceAccessCallBack()

Registers given resource access *callback* called if resource *name* is accessed as specified by *accessType*.

```
virtual CADIReturn_t CADIProfiling::CADIProfileRegisterResourceAccessCallBack(
        CADIProfileResourceCallBack_t callBack, const char * name,
        CADIProfileResourceAccessType_t accessType) =0
```

3.4.18 CADIProfiling::CADIProfileUnregisterResourceAccessCallBack()

Unregisters the profile hazard callbacks.

```
virtual CADIReturn_t CADIProfiling::CADIProfileUnregisterResourceAccessCallBack(
        CADIProfileResourceCallBack_t callBack, const char * name) =0
```

3.4.19 CADIProfiling::CADIProfileRegisterCallBack()

Registers the profile hazard callbacks.

```
virtual CADIReturn_t CADIProfiling::CADIProfileRegisterCallBack (
        CADIProfileHazardCallBack_t callBack) =0
```


3.4.20 CADIProfiling::CADIProfileUnregisterCallBack()

Unregisters the hazard callback.

```
virtual CADIReturn_t CADIProfiling::CADIProfileUnregisterCallBack (
        CADIProfilingCallbacks *callbackObject) =0
```

3.4.21 CADIProfilingCallbacks::profileResourceAccess()

Profile resource callback.

```
virtual CADIReturn_t CADIProfilingCallback::profileResourceAccess (
        const char * name, CADIProfileResourceAccessType_t accessType) =0
```

3.4.22 CADIProfilingCallbacks::profileRegisterHazard()

Profile hazard callback.

```
virtual CADIReturn_t CADIProfilingCallback::profileRegisterHazard(
        CADIProfileHazardDescription_t desc) =0
```

desc is of type CADIProfileHazardDescription_t:

```
class CADIProfileHazardDescription_t
{
public:
      CADIProfileHazardDescription_t(CADIProfileHazardTypes_t type =
          CADI_PROF_HAZARD_RESOURCE_MAX_ACCESS,
          uint32_t numberOfAccesses = 0,
          uint32_t originInstructionFID = 0,
          uint32_t affectedInstructionFID = 0,
          const char *resource_par = "",
          const char *messages_par = "") :
          type(type), numberOfAccesses(numberOfAccesses),
          originInstructionFID(originInstructionFID),
          affectedInstructionFID(affectedInstructionFID){
              AssignString(resource, resource_par, CADI_DESCRIPTION_SIZE);
              AssignString(message, messages_par, CADI_DESCRIPTION_SIZE);
          }

          CADIProfileHazardTypes_t type;
          uint32_t numberOfAccesses; // number of accesses to affected resource
          uint32_t originInstructionFID;  // FID of the originator
                                          // resource/instruction
          uint32_t affectedInstructionFID;  // name of the affected
                                            // resource/instruction
          char        resource[CADI_DESCRIPTION_SIZE];
          char        message[CADI_DESCRIPTION_SIZE];  // hazard message
};
```

3.5 The CADICallback class

The CADICallbackObj class is the base class for the CADI callbacks in the component.

3.5.1 CADICallbackObj::appliOpen()

Opens an application and returns the ID of the stream.

```
virtual uint32_t CADICallbackObj::appliOpen (const char * sFileName,
        const char * mode) =0
```

where:

sFileName is name of the file to be opened.

mode indicates the permitted access on the file. See the ANSI C definition of fopen for possible values of this parameter.

3.5.2 CADICallbackObj::appliInput ()

Input data from the stream opened by appliOpen().

```
virtual void CADICallbackObj::appliInput (uint32_t streamId, uint32_t count,
        uint32_t * actualCount, char * buffer) =0
```

where:

streamId is the stream identifier. Set to CADI_STREAMID_STDIN for stdin.

count is the number of characters requested.

actualCount is the number of characters supplied. This number must never be greater than the number of characters requested. If this number is equal to the number of characters requested, the caller can repeat the call to request more input. A return value of 0 indicates end of file. A return of –1 indicates an error such as, for example, an invalid stream ID.

buffer is the characters stream. The buffer is not null terminated.

3.5.3 CADICallbackObj::appliOutput ()

Write data to the stream opened by `appliOpen()`.

```
virtual void CADICallbackObj::appliOutput (uint32_t streamId, uint32_t count,
        uint32_t * actualCount, const char * buffer) =0
```

where:

`streamId` is the stream identifier and must be either `CADI_STREAMID_STDOUT` or `CADI_STREAMID_STDERR`.

`count` is the number of characters to output.

`actualCount` is the number of characters output to the file. A return value of 0 indicates end of file. A return of –1 indicates an error.

`buffer` contains the characters to output. This buffer can contain null characters and is not null terminated.

3.5.4 CADICallbackObj::appliClose()

Note

This function is deprecated. Do not use it in new models.

Close the stream opened by `appliOpen()`. If the return value is 1, the file was successfully closed. A return value of –1 indicates an error.

```
virtual uint32_t CADICallbackObj::appliClose (uint32_t streamID) =0
```

3.5.5 CADICallbackObj::doString()

Output a string from the debugger.

```
virtual void CADICallbackObj::doString (char * stringArg) =0
```

3.5.6 CADI::CADICaptureSemihosting()

Specify that this debugger and this callback object wants to capture the semihosting input exclusively. The tool will forward all inputs through `appliInput` callback to this debugger exclusively (the outputs through `appliOutput` are broadcast).

```
virtual CADIReturn_t CADI::CADICaptureSemihosting(CADICallbackObj * callback) =0
```

where:

`callback` is the callback object that will capture the semihosting.

3.5.7 CADI::CADIConsoleGetChannels()

This is a reverse console function.

```
virtual CADIReturn_t CADIConsoleGetChannels(uint32_t startIndex,
            uint32_t desiredCount, uint32_t *actualCount,
            CADIConsoleChannel_t *channels) =0
```

where:

`startIndex` is the start index in the list of channels.

`desiredCount` is the number of channels to return.

`actualCount` is the actual number of channels returned.

`channels` is an array of channels.

3.5.8 CADI::CADIConsoleNotifyInput()

This is a reverse console function.

```
virtual CADIReturn_t CADIConsoleNotifyInput(uint32_t streamID) = 0
```

where:

`streamID` is the identifier for the requested stream.

3.5.9 CADI::CADISemiHostingGetInputChannels()

This is a reverse semihosting function.

```
virtual CADISemiHostingGetInputChannels(uint32_t startIndex,
        uint32_t desiredCount, uint32_t *actualCount,
        CADISemiHostingInputChannel_t *channels) =0
```

where:

`startIndex` is the start index in the list of channels.

`desiredCount` is the number of channels to return.

`actualCount` is the actual number of channels returned.

`channels` is an array of channels.

3.5.10 CADI::CADISemiHostingSendInput()

This is a reverse semihosting function.

```
virtual CADIReturn_t CADISemiHostingSendInput(uint32_t channelID,
         uint32_t inputCount, uint32_t *input) =0
```

where:

`channelID` is identifier of the channel to use.

`inputCount` is number of values to send.

`input` is an array containing the values to send.

3.5.11 CADICallbackObj::modeChange()

Set a new execution mode from the callback object.

```
virtual void CADICallbackObj::modeChange (uint32_t newMode,
        CADIBptNumber_t bptNumber) =0
```

where:

`newMode` is one of the `CADI_EXECMODE_*` constants (see *CADI_EXECMODE_t enumeration* on page 3-90).

`bptNumber` is the breakpoint number. This value is used if the debugger has an action associated with that particular breakpoint. Temporary breakpoints, for example, might run a script after the breakpoint was hit.

3.5.12 CADICallbackObj::reset()

Reset the execution from the callback object. See `CADIExecGetResetLevels()` for a description of the levels.

```
virtual void CADICallbackObj::reset (uint32_t resetLevel) =0
```

3.5.13 CADICallbackObj::cycleTick()

Deprecated. Do not use.

```
virtual void CADICallbackObj::cycleTick (void) =0
```

3.5.14 CADICallbackObj::killInterface()

Deprecated. Do not use.

```
virtual void CADICallbackObj::killInterface (void) =0
```

3.5.15 CADICallbackObj::bypass()

Reserved for future use by the callback object.

```
virtual uint32_t CADICallbackObj::bypass (uint32_t commandLength,
        const char * command, uint32_t maxResponseLength, char * response) =0
```

3.5.16 CADICallbackObj::lookupSymbol()

Reserved for future use by the callback object.

```
virtual uint32_t CADICallbackObj::lookupSymbol (uint32_t symbolLength,
        const char * symbol, uint32_t maxResponseLength, char * response) =0
```

3.5.17 CADICallbackObj::refresh()

Use this callback whenever the state of a target changes spontaneously while the model is in the `stopped` state. Do not use it with a `modeChange(Stop)`, `modeChange(Error)` or `modeChange(ResetDone)` callback.

A target can notify a debugger to update its display if, for example, a register value changes in the target because it was edited by a debugger. The target uses `refresh(REGISTERS)` to notify the other debuggers of the register change. If, however, a target hits a breakpoint and stops, it must call the necessary `modeChange()` callbacks instead of the `refresh()` callbacks.

```
virtual void CADICallbackObj::refresh (uint32_t refreshReason) =0
```

refreshReason is a combination of the `CADI_REFRESH_REASON_t` constants.

```
enum CADIRefreshReason_t
{
   CADI_REFRESH_REASON_MEMORY      = 1,

   // also for CADIGetInstructionCount/CADIGetCycleCount
   CADI_REFRESH_REASON_REGISTERS   = 2,

   CADI_REFRESH_REASON_BREAKPOINTS = 4,
   CADI_REFRESH_REASON_PARAMETERS  = 8,

   // something changed which is not one of the above
```



```
    CADI_REFRESH_REASON_OTHER       = (1 << 31),

    // all of the above at the same time
    CADI_REFRESH_REASON_ALL         = 0xFFFFFFFF
};
```

See *CADI_EXECMODE_t enumeration* on page 3-90 for details on the relationship between `modeChange()` callbacks and `refresh()` callbacks. A target must not call this function while the simulation is running.

3.6 CADIBroker

This interface allows connecting to existing simulations and creating new simulations.

Note

The CADI broker owns all CADI simulations and no other class is permitted to delete them.

If a CADI factory creates a simulation, it must transfer the pointer to the new simulation to the broker.

If the simulation is shut down or killed, the broker is responsible for deleting the simulation. This must be done by processing `GetSimulationInfos()` and checking for running simulations (check that the reference count is 0 and any other implementation-specific conditions are in the appropriate state).

Example 3-5 The CADIBroker class

```
class WEXP CADIBroker: public CAInterface
{
public:
static if_name_t IFNAME() { return "eslapi.CADIBroker2"; }
static if_rev_t IFREVISION() { return 0; }
virtual ~CADIBroker() {}
virtual void Release () = 0;
virtual CADIReturn_t GetSimulationFactories(uint32_t startFactoryIndex,
uint32_t desiredNumberOfFactories, CADISimulationFactory **factoryList,
uint32_t *actualNumberOfFactories) = 0;
virtual CADIReturn_t GetSimulationInfos(uint32_t startSimulationInfoIndex,
uint32_t desiredNumberOfSimulations, CADISimulationInfo_t *simulationList,
uint32_t *actualNumberOfSimulations) = 0;
virtual CADISimulation* SelectSimulation( uint32_t simulationId,
CADIErrorCallback* errorCallbackObject, CADISimulationCallback*
simulationCallbackObject, char simulationCallbacksEnable[CADI_SIM_CB_Count])=0;
};
```

3.6.1 CADIBroker::Release()

Release this simulation. A debugger is expected to release the simulation as soon as the CADI target is obtained.

```
virtual void Release() =0;
```

3.6.2 CADIBroker::GetSimulationFactories()

Returns a list of possible simulation factories provided by this simulation broker. This list is static for a given CADIBroker.

Because this function copies an array of CADISimulationFactory pointers, the caller must be responsible for calling ReleaseRef() on each of the referenced objects.

```
virtual CADIReturn_t CADIBroker::GetSimulationFactories(
                     uint32_t startFactoryIndex,
                     uint32_t desiredNumberOfFactories,
                     CADISimulationFactory **factoryList,
                     uint32_t *actualNumberOfFactories) = 0;
```

where:

startFactoryIndex

is the index of first factory to return. If startFactoryIndex exceeds the maximum factory index, CADI_STATUS_IllegalArgument is returned.

desiredNumberOfFactories

is the desired number of factories to return.

Caution

The factoryList array must be at least this size.

factoryList is the array of factory pointers returned. This array must be allocated by caller. The minimum size of this array is desiredNumberOfFactories.

Note

The returned factory pointers must not be used to delete the factories. The factories are owned by the broker.

actualNumberOfFactories

is the actual number of factories returned.

3.6.3 CADIBroker::GetSimulationInfos()

Returns a list of simulation infos informing about the running simulations managed by this CADI simulation broker.

Note

This list may change dynamically during lifetime of this CADIBroker.

```
virtual CADIReturn_t CADIBroker::GetSimulationInfos(
                     uint32_t startSimulationInfoIndex,
                     uint32_t desiredNumberOfSimulations,
                     CADISimulationInfo_t *simulationList,
                     uint32_t *actualNumberOfSimulations) = 0;
```

where:

`startSimulationInfoIndex`

is the index of the first simulation info to return.

If `startSimulationInfoIndex` exceeds the maximum simulation info index, `CADI_STATUS_IllegalArgument` is returned.

`desiredNumberOfSimulations`

is the desired number of simulation infos to return.

Caution

Array `simulationInfoList` must have at least this size.

`simulationList`

is the array of simulation infos returned. This array must be allocated by the caller.

Note

The minimum size of this array is `desiredNumberOfSimulationInfos`.

`actualNumberOfSimulations`

is the actual number of simulation infos returned.

3.6.4 CADIBroker::SelectSimulation()

This method enables connecting to the simulation selected by the simulation identifier. A pointer to the simulation is returned on success. If no simulation with the given id is managed by this broker, 0 is returned.

```
virtual CADISimulation* CADIBroker::SelectSimulation( uint32_t simulationId,
                        CADIErrorCallback* errorCallbackObject,
                        CADISimulationCallback* simulationCallbackObject,
                        char simulationCallbacksEnable[CADI_SIM_CB_Count]) = 0;
```


where:

`simulationId`

is the id of the simulation to be returned. This is part of the respective entry in the list of the simulation infos `simulationList` returned by `GetSimulationInfos()`.

`errorCallbackObject`

is the error callback to be used for signaling error conditions.

`simulationCallbackObject`

is the callback object to be used for signaling model-wide conditions.

This callback might be called during execution of `SelectSimulation()` to, for example, signal that the simulation wants to shut down.

Note

This callback might be called during execution of `SelectSimulation()` to, for example, signal that the simulation wants to shut down.

`simulationCallbacksEnable`

The elements of this array enable or disable specific simulation callbacks. The simulation must always check if the callbacks are enabled or not and these should not be called if they are disabled. The listener might not want to be called in certain cases.

`return value`

is the pointer to the simulation or NULL.

3.6.5 CADIErrorCallback::Error()

The `CADIErrorCallback` class is the base class for CADI error callbacks. The function `Error()` signals an error to the CADI listeners.

```
virtual void CADIErrorCallback::Error(CADIFactorySeverityCode_t severity,
                            CADIFactoryErrorCode_t errorCode,
                            uint32_t erroneousParameterId,
                            const char *message) = 0;
```

where:

`severity` is the severity of the error.

`errorCode` is the error code.

`erroneousParameterId`

If this error refers to a parameter, this is the id of the parameter that caused the error.

`message` is the error message text.

3.6.6 Creating the CADIBroker

Example 3-6 shows the prototypes for the functions that create the CADIBroker.

Example 3-6 Creating the CADIBroker

```
extern "C"
{
// Global function exported by a dynamically loaded object.
// This function must exist in a dynamically loaded object (DLL/.so).
// It allows the client to instantiate the CADIBroker.
CADI_WEXP eslapi::CADIBroker * CreateCADIBroker();
}
```

The prototype type definition `CADIBroker * (CreateCADIBroker_t)()` enables a global function to instantiate a broker. This is the type of the `CreateCADIBroker` global C function that a client locates from a dynamically loaded object. Clients must locate this symbol and cast it as a pointer to `CreateCADIBroker_t`:

```
CreateCADIBroker_t:
void * entry = lookup_symbol(dll, "CreateCADIBroker");
CADIBroker *broker = ((*CADIBroker::CreateCADIBroker_t)entry)();
```


3.7 The CADISimulationFactory class

The CADISimulationFactory class provides the mechanism used to start new simulations.

Example 3-7 CADISimulationFactory class

```
class CADI_WEXP CADISimulationFactory : public CAInterface
{
public:
    // Return the CAInterface name for this interface.
    static if_name_t IFNAME() { return "eslapi.CADISimulationFactory2"; }
    // Specify the current minor revision for this interface.
    static if_rev_t IFREVISION() { return 0; }

    virtual void Release () = 0;
    virtual const char* GetName() = 0;
    virtual CADIReturn_t GetParameterInfos(uint32_t startParameterInfoIndex,
                    uint32_t desiredNumberOfParameterInfos,
                    CADIParameterInfo_t *parameterInfoList,
                    uint32_t *actualNumberOfParameterInfos) = 0;

    virtual CADISimulation *Instantiate(CADIParameterValue_t *parameterValues,
                     CADIErrorCallback *errorCallbackObject,
                     CADISimulationCallback *simulationCallbackObject,
                     char simulationCallbacksEnable[CADI_SIM_CB_Count]) = 0;
};
```

3.7.1 CADISimulationFactory::Release()

Release this simulation. A debugger is expected to release the simulation factory as soon as the CADI target is obtained.

```
virtual void CADISimulationFactory::Release() =0;
```

3.7.2 CADISimulationFactory::GetParameterInfos()

Returns a list of simulation infos informing about the running simulations managed by this CADI simulation broker.

Note

This list may change dynamically during lifetime of this CADIBroker.

```
virtual CADIReturn_t CADISimulationFactory::GetSimulationInfos(
                     uint32_t startSimulationInfoIndex,
                     uint32_t desiredNumberOfSimulations,
                     CADISimulationInfo_t *simulationList,
                     uint32_t *actualNumberOfSimulations) = 0;
```

where:

`startSimulationInfoIndex`

is the index of the first simulation info to return. If `startSimulationInfoIndex` exceeds the maximum simulation info index, `CADI_STATUS_IllegalArgument` is returned

`desiredNumberOfSimulations`

is the desired number of simulation infos to return.

Caution

Array `simulationInfoList` must have at least this size.

`simulationList`

is the array of simulation infos returned. This array must be allocated by the caller.

Note

The minimum size of this array is `desiredNumberOfSimulationInfos`.

`actualNumberOfSimulations`

is the actual number of simulation infos returned.

3.7.3 CADISimulationFactory::Instantiate()

This method instantiate a simulation based on the given parameter values. Errors occurring during system initialization are signaled through the given error callback `CADIErrorCallback`.

Note

This call might require a significant amount of time to complete. The call does not return until the instantiation is completed.


```
virtual CADISimulation * CADISimulationFactory::Instantiate(
                    CADIParameterValue_t *parameterValues,
                    CADIErrorCallback *errorCallbackObject,
                    CADISimulationCallback *simulationCallbackObject,
                    char simulationCallbacksEnable[CADI_SIM_CB_Count]) = 0;
```

where:

`parameterValues`

are the parameter values for the simulation.

`errorCallbackObject`

is the error callback object to be used for signaling error conditions.

`simulationCallbackObject`

is the callback object to be used for signaling model-wide conditions.

`simulationCallbacksEnable`

The elements of this array enable or disable specific simulation callbacks.

Note

The simulation must always check if the callbacks are enabled or not and these must not be called if they are disabled. The listener might not want to be called in certain cases.

`return value`

is the pointer to the created simulation or NULL.

3.7.4 The CADISimulationCallback class

CADISimulationCallback is the base class for simulation callbacks. It enables registering as a listener for system-wide callbacks.

```
class CADI_WEXP CADISimulationCallback :
        public CAInterface
{
public:
        // Return the CAInterface name for this interface.
        static if_name_t IFNAME() { return "eslapi.CADISimulationCallback2"; }
        // Specify the current minor revision for this interface.
        static if_rev_t IFREVISION() { return 0; }
        virtual void simMessage(const char *message) = 0;
        virtual void simShutdown() = 0;
        virtual void simKilled() = 0;
};
```

3.7.5 CADISimulationCallback::simMessage()

This method enables sending system-wide messages to all listeners.

```
virtual void CADISimulationCallback::simMessage(const char *message) = 0;
```

where:

`message` is the message text to send to the listeners.

3.7.6 CADISimulationCallback::simShutdown()

This method enables the simulation to signal that it wants to shut down. All clients are requested to unregister their callback handlers, and release any references to the simulation.

```
virtual void CADISimulationCallback::simShutdown() = 0;
```

3.7.7 CADISimulationCallback::simKilled()

The simulation is being forcedly terminated. After this call returns, the client must cease all communication with the simulation. This callback is intended to provide last-ditch recovery in situations where it is not possible to go through the clean `simShutdown()` route.

```
virtual void CADISimulationCallback::simKilled() = 0;
```

3.7.8 The CADIErrorCallback class

CADIErrorCallback is the base class for error callbacks.

```
class CADI_WEXP CADIErrorCallback : public CAInterface
{
public:
    // Return the CAInterface name for this interface.
    static if_name_t IFNAME() { return "eslapi.CADIErrorCallback2"; }

    // Specify the current minor revision for this interface.
    static if_rev_t IFREVISION() { return 0; }

    // This message is called to signal an error to the listeners
    virtual void Error(CADIFactorySeverityCode_t severity,
           CADIFactoryErrorCode_t errorCode, uint32_t erroneousParameterId,
           const char *message) = 0;
};
```


3.7.9 CADIErrorCallback::Error()

This method is called to signal an error to the listeners.

```
virtual void Error(CADIFactorySeverityCode_t severity,
            CADIFactoryErrorCode_t errorCode,  uint32_t erroneousParameterId,
            const char *message) = 0;
```

where:

`severity` is the severity of the error (severities are defined in the `CADIFactorySeverityCode_t` type).

`errorCode` is the error code (codes are defined in the `CADIFactoryErrorCode_t` type).

`erroneousParameterId`

if this error refers to a parameter, this is the id of the parameter causing the error.

`message` is the error message.

3.8 CADI data structures

This section describes some of the data structures and enumerations that are used by the CADI interface. See the `CADITypes.h` file for details on data structures that are not covered in this section. This section describes the following structures:

- *CADITargetFeatures_t structure*
- *CADICallbackType_t* on page 3-81
- *CADICallbackObj* on page 3-82
- *CADIReg_t* on page 3-82
- *CADIRegInfo_t* on page 3-83
- *CADIRegGroup_t* on page 3-85
- *CADIMemSpaceInfo_t* on page 3-86
- *CADIMemBlockInfo_t* on page 3-87
- *CADIPipeStage_t* on page 3-89
- *CADIPipeStageContentInfo_t* on page 3-89
- *CADIBptConfigure_t* on page 3-89
- *CADIDisassemblerStatus* on page 3-90
- *CADI_EXECMODE_t enumeration* on page 3-90
- *CADIExecMode_t structure* on page 3-91
- *CADIFactoryErrorCode_t* on page 3-92
- *CADIFactorySeverityCode_t* on page 3-93
- *CADISimulationInfo_t* on page 3-93.
- *CADIParameterInfo_t and CADIParameterValue_t* on page 3-95
- *CADIReturn_t* on page 3-96
- *CADIBptCondition_t and CADIBptConditionOperator_t* on page 3-97
- *CADIBptRequest_t* on page 3-94.

3.8.1 CADITargetFeatures_t structure

The `CADITargetFeatures_t` structure entries are:

`char targetName[CADI_NAME_SIZE]`

is the target name.

`char targetVersion[CADI_NAME_SIZE]`

is the target version.

`uint32_t nrBreakpointsAvailable`

is the number of breakpoints available for the interface.

`uint8_t fOverlaySupportAvailable`

indicates whether overlay is supported.

`uint8_t fProfilingAvailable`

indicates whether profiling is supported for this interface.

`uint32_t nrResetLevels`

is the number of reset levels (for example, hard or soft reset).

`uint32_t nrExecModes`

is the number of execution modes.

`uint32_t nrExceptions`

is the number of exceptions.

`uint32_t nrMemSpaces`

is the number of memory spaces.

`uint32_t nrRegisterGroups`

is the number of register groups.

`uint32_t nrPipeStages`

is the number of pipeline stages.

`uint32_t nPCRegNum`

is the number of the register that is used for the program counter.

`uint16_t handledBreakpoints`

is the number of handled breakpoints.

`uint32_t nrOfHWThreads`

is the number of hardware threads.

`bool nExtendedTargetFeaturesRegNumValid`

indicates whether extended target features are supported for registers.

`uint32_t nExtendedTargetFeaturesRegNum`

is the register id of a string register which contains a static string consisting of colon separated tokens or arbitrary non-colon-ASCII char such as `FOO:BAR:ANSWER=42:STARTUP=0xe000`.

The set and semantics of supported tokens are out of scope of the CADI interface itself. There is no length restriction on this feature string. Having such a string register is optional. Models which do not provide it

must set nExtendedTargetFeaturesRegNumValid to false. In this case, the value of this field must be ignored. Having no such register and having a string register that provides an empty string is equivalent. The following tokens (where n denotes a decimal unsigned 32 bit integer) are defined for CADI 2.0:

`PC_MEMSPACE_REGNUM=n`

Register ID of the a register which contains the memory space ID of the current PC described in nPCRegNum.

`SP_REGNUM=n:`

Register ID of the stack pointer (or a register with similar semantics).

`LR_REGNUM=n:`

Register ID of the link register (or a register with similar semantics).

`STACK_MEMSPACE_ID=n:`

CADI memory space id used for stack unwinding

`LOCALVAR_MEMSPACE_ID=n:`

CADI memory space id used for local variables

`GLOBALVAR_MEMSPACE_ID=n:`

CADI memory space id used for global vars

Targets that do not have one of the features described above simply will not expose such a token.

`char canonicalRegisterDescription`

is a string that describes the contents of the `canonicalRegisterNumber` field of `CADIRegInfo_t`. Canonical register numbers are intended to be target-specific numbers to identify registers in the device by some scheme other than the DWARF index. The format of this field is `domain_name/string`. The `domain_name` is that of the organization specifying the scheme. The string part is left to the organization to specify. An example would be `arm.com/my/reg/numbers`.

`char canonicalMemoryDescription[CADI_DESCRIPTION_SIZE]`

is a string that describes the contents of the `canonicalMemoryNumber` field of `CADIMemSpaceInfo_t`. Canonical memory numbers are intended to be target-specific numbers to identify memory spaces in the device by some scheme other than the DWARF index. The format of this field is `'domain_name/string'`. The `domain_name` is that of the organization specifying the scheme. The string part is left to the organization to specify such as, for example, `arm.com/my/mem/numbers`.

`uint8_t canCompleteMultipleInstructionsPerCycle`

is true if the target can complete multiple instructions in a single simulation cycle.

3.8.2 CADICallbackType_t

The values in this type indicate the callback type used by `CADIXfaceAddCallback()`.

```
enum CADICallbackType_t
{
    CADI_CB_AppliOpen       =  0, /* Open the specified filename and return a
           streamID that the AppliInput and Applioutput functions can use. */

    CADI_CB_AppliInput      =  1, /* This is used for input.
            Data travels from the host to the target. */

    CADI_CB_AppliOutput     =  2, /* This is used for output.
            Data travels from the target to the host. */

    CADI_CB_AppliClose      =  3, // Close the stream specified by streamID.

    CADI_CB_String          =  4, /* The target system calls this to have the
            debugger display a string. Among other things, it can be used for
            things like hazard and stall indication. */

    CADI_CB_ModeChange      =  5, /* Call this when the target changes execution
            modes as defined by CADIExecGetModes. The bptNumber parameter is
            ignored if the mode is not CADI_EXECMODE_Bpt. */

    CADI_CB_Reset           =  6, // Called when the target is reset.

    CADI_CB_CycleTick       =  7, /* This callback, when installed, is
              called after every cycle that is executed by the target.*/

    CADI_CB_KillInterface   =  8, /* This call must ALWAYS be enabled. This is
              called when the target is dying. No further communication with the
              target is allowed after this callback is made. */

    CADI_CB_Bypass          =  9, /* Callback to bypass the interface, to allow
              any string-based communication with the debugger.*/

    CADI_CB_LookupSymbol    = 10, //  Lookup a symbol from the debugger

    CADI_CB_DisasmNotifyModeChange = 11, // Target mode was changed.

    CADI_CB_DisasmNotifyFileChange = 12, // Target file was changed.

    CADI_CB_Refresh         = 13, /* Used to notify debugger that it needs to
```

```
                refresh its state (e.g., register values changed) */

    CADI_CB_ProfileResourceAccess = 14, // Profile resource callback

    CADI_CB_ProfileRegisterHazard = 15, // Register hazard callback

    CADI_CB_Count           = 16,

    CADI_CB_ENUM_MAX = 0xFFFFFFFF
    };
```

3.8.3 CADICallbackObj

The CADICallbackObj definition is:

```
class CADICallbackObj
{
public:
      virtual ~CADICallbackObj(){}
      virtual uint32_t  appliOpen(const char *sFileName, const char *mode) = 0;
      virtual void appliInput(uint32_t streamId, uint32_t count,
                      uint32_t *actualCount, char *buffer) = 0;
      virtual void appliOutput(uint32_t streamId, uint32_t count,
                     uint32_t *actualCount, const char *buffer) = 0;
      virtual uint32_t  appliClose(uint32_t streamID) = 0;
      virtual void doString(char *stringArg) = 0;
      virtual void modeChange(uint32_t newMode, CADIBptNumber_t bptNumber) = 0;
      virtual void reset(uint32_t resetLevel) = 0;
      virtual void cycleTick(void) = 0;
      virtual void killInterface(void) = 0;
      virtual uint32_t bypass(uint32_t commandLength, const char * command,
                  uint32_t maxResponseLength, char * response) { return 0; }
      virtual uint32_t lookupSymbol(uint32_t symbolLength, const char * symbol,
                  uint32_t maxResponseLength, char * response) { return 0; }
    };
```

3.8.4 CADIReg_t

This data buffer is used to read and write register values. The register data is packed, byte-by-byte, into the bytes array. Data is always encoded in little endian mode. For example, the lowest address in the bytes array contains the least significant value of the register.

Example 3-8 CADIReg_t

```
struct CADIReg_t
{
public: // methods
        CADIReg_t(uint32_t regNumber = 0, uint64_t bytes_par = 0, uint16_t offset128 = 0,
                  bool isUndefined = false, CADIRegAccessAttribute_t attribute = CADI_REG_READ_WRITE) :
                  regNumber(regNumber), offset128(offset128), isUndefined(isUndefined),
                  attribute(attribute)
        {
            for(int i=0; i < 8; ++i)
                bytes[i] = uint8_t(bytes_par >> (i * 8));
        }

public: // data
        uint32_t   regNumber; /* From debugger to target: Register ID to be read/written. */
        uint8_t    bytes[16]; /* From target to debugger for reads, from debugger to target for writes:
                                 Value to be read/written in little endian
                                 (regardless of the endianness of the host or the target). */
        uint16_t   offset128; /* From debugger to target:
                                 Specify which part of the register value to read/write for
                                 long registers > 128 bits. Measured in multiples of 128 bits,
                                 e.g. 1 means bytes[0..15] contain bits 128..255). The actual bitwidth of
                                 non-string registers is determined by the 'bitsWide' field in
                                 CADIRegInfo_t. Similarly for string registers, specify the offset in
                                 units of 16 chars into the string which is to be read or written,
                                 e.g. offset128=1 means read/write str[16..31]. Reads to offsets beyond
                                 the length of the string are explicitly allowed and need to
                                 result in bytes[0..15] being all zero. Writes may make the string longer
                                 by writing nonzero data to offsets greater than the current length of a
                                 string. Writes may make a string shorter by writing data containing at
                                 least one zero byte to a specific offset. Write sequences always write
                                 lower offsets before higher offsets and must always be terminated by at
                                 least one write containing at least one zero byte.
                                 Unused chars in bytes[0..15] (after the terminating zero byte) should be
                                 set to zero.
                                 The 'bitsWide' field in CADIRegInfo_t is ignored for string registers.*/
        bool  isUndefined;    /* From target to debugger: If true the value of the register is
                                 completely undefined. Bytes[0..15] should be ignored. */
        CADIRegAccessAttribute_t attribute; /* Undefined for CADI2.0. Targets and Debuggers should set
                                 this to 0. */
};
```

3.8.5 CADIRegInfo_t

This structure defines information about a register.

Example 3-9 CADIRegInfo_t

```
struct CADIRegInfo_t
{
public: // methods
    CADIRegInfo_t(const char *name_par = "",  const char *description_par = "",
                uint32_t regNumber = 0, uint32_t bitsWide = 0,
                int32_t hasSideEffects = 0,
                CADIRegDetails_t details = CADIRegDetails_t(),
                CADIRegDisplay_t display = CADI_REGTYPE_HEX,
                CADIRegSymbols_t symbols = CADIRegSymbols_t(),
                CADIRegFloatFormat_t fpFormat = CADIRegFloatFormat_t(),
                uint32_t lsbOffset = 0, uint32_t dwarfIndex = ~0U,
                bool isProfiled = false, bool isPipeStageField = false,
                uint32_t threadID = 0,
                CADIRegAccessAttribute_t attribute = CADI_REG_READ_WRITE,
                uint32_t canonicalRegisterNumber_ = 0):
            regNumber(regNumber),  bitsWide(bitsWide),
            hasSideEffects(hasSideEffects), details(details), display(display),
            symbols(symbols), fpFormat(fpFormat), lsbOffset(lsbOffset),
            dwarfIndex(dwarfIndex), isProfiled(isProfiled),
            isPipeStageField(isPipeStageField), threadID(threadID),
            attribute(attribute),
            canonicalRegisterNumber(canonicalRegisterNumber_)
        {
            AssignString(name, name_par, CADI_NAME_SIZE);
            AssignString(description, description_par, CADI_DESCRIPTION_SIZE);
        }

public: // data
        char              name[CADI_NAME_SIZE];
        char              description[CADI_DESCRIPTION_SIZE];
        uint32_t          regNumber; // Register ID. Used by read/write
                                     // functions to identify the register.
        uint32_t          bitsWide; /* Bitwidth of non-string register.
                                       Ignored for string registers (targets
                                       should specify 0 for string registers). */
        int32_t              hasSideEffects;
        CADIRegDetails_t  details;
        CADIRegDisplay_t  display;        // Default is "HEX".
        CADIRegSymbols_t  symbols;        // For type "symbolic" only.
        CADIRegFloatFormat_t  fpFormat;   // For type "float" only.
        uint32_t           lsbOffset;     /* Offset of the least significant bit
                                        relative to bit 0  in the parent register
                                        (or 0 if there is no parent).*/
        enum { CADI_REGINFO_NO_DWARF_INDEX = 0xffffffff };
        uint32_t           dwarfIndex; /* DWARF register index
                                    (CADI_REGINFO_NO_DWARF_INDEX if register has no
                                     DWARF register  index).*/
```



```
        bool             isProfiled;     // Profiling info is available
        bool             isPipeStageField; /* Is pipe stage field, also true
                                              for pc and contentInfoRegisterId
                                              in CADIPipeStage_t. */
        uint32_t         threadID;   // Hardware thread ID, always set to 0.
        CADIRegAccessAttribute_t attribute; ///< Register access attributes.

        uint32_t         canonicalRegisterNumber; /* The canonical register
                           number as defined by the scheme specified in
                           CADITargetFeatures_t::canonicalRegisterDescription.
                           If the scheme is the empty string then no
                           meaning can be ascribed to this field.  */

};
```

3.8.6 CADIRegGroup_t

Register group description. All fields are target to debugger fields.

Example 3-10 CADIRegGroup_t

```
struct CADIRegGroup_t
{
public: // methods
    CADIRegGroup_t(uint32_t groupID = 0,
                const char *description_par = "",  uint32_t numRegsInGroup = 0,
                 const char *name_par = "", bool isPseudoRegister = false) :
            groupID(groupID), numRegsInGroup(numRegsInGroup),
            isPseudoRegister(isPseudoRegister)
    {
        AssignString(description, description_par, CADI_DESCRIPTION_SIZE);
        AssignString(name, name_par, CADI_NAME_SIZE);
    }

public: // data
    uint32_t   groupID;
    char  description[CADI_DESCRIPTION_SIZE];
    // This is the total number of registers in the group, including any
    // registers that are not direct children of this group.
    uint32_t   numRegsInGroup;
    char  name[CADI_NAME_SIZE];
    bool  isPseudoRegister; /* True means that this register group is not
                       displayed in the register window in the debugger.
                       The registers in this group are probably serving other
```



```
                                purposes such as pipeline stage fields or other special
                                purpose registers (like the PC memory space). */
    };
```

3.8.7 CADIMemSpaceInfo_t

Memory space info data. Each memory space (program and data, for example) in the system has a separate set of addresses. Any location in the memory of a device can be fully specified with no less than an indication of the memory space and the address within that space. Only one space can have the isProgramMemory flag set.

Example 3-11 CADIMemSpaceInfo_t

```
struct CADIMemSpaceInfo_t
{
public: // methods
        CADIMemSpaceInfo_t(const char *memSpaceName_par = "", const char *description_par = "",
                           uint32_t memSpaceId = 0, uint32_t bitsPerMau = 0,
                           CADIAddrSimple_t maxAddress = 0, uint32_t nrMemBlocks = 0,
                           int32_t isProgramMemory = false, CADIAddrSimple_t minAddress = 0,
                           int32_t isVirtualMemory = false, uint32_t isCache = false,
                           uint8_t endianness = 0, uint8_t invariance = 0,
                           uint32_t dwarfMemSpaceId = NO_DWARF_ID) :
            memSpaceId(memSpaceId), bitsPerMau(bitsPerMau), maxAddress(maxAddress),
            nrMemBlocks(nrMemBlocks), isProgramMemory(isProgramMemory), minAddress(minAddress),
            isVirtualMemory(isVirtualMemory), isCache(isCache), endianness(endianness),
            invariance(invariance), dwarfMemSpaceId(dwarfMemSpaceId)
        {
            AssignString(memSpaceName, memSpaceName_par, CADI_NAME_SIZE);
            AssignString(description, description_par, CADI_DESCRIPTION_SIZE);
        }

public: // data
        char      memSpaceName[CADI_NAME_SIZE];   // Memory space name.
        char      description[CADI_DESCRIPTION_SIZE];  // Memory space description.
        uint32_t   memSpaceId;                 // Memory space ID.
        uint32_t   bitsPerMau;                 // Bits per Minimum Addressable Unit (e.g., 8 for byte).
        CADIAddrSimple_t maxAddress;           // Maximum address of this memory space.
        uint32_t   nrMemBlocks;                // Number of memory blocks.
        int32_t   isProgramMemory;             // Only one space can have the isProgramMemory flag set.
        CADIAddrSimple_t minAddress;           // Minimum address of this memory space.
        int32_t   isVirtualMemory;             // This memory space is a Virtual/Physical space.
        uint32_t   isCache;                    // This memory space is a cache.

        uint8_t    endianness; // endianness, 0=mono-endian (arch defined), 1=LE, 2=BE
        uint8_t    invariance; // unit of invariance in bytes, 0=fixed invariance (arch defined)
```



```
        enum { NO_DWARF_ID = 0xffffffff };
        uint32_t   dwarfMemSpaceId; /* DWARF memory space ID (NO_DWARF_ID
                                       if memory space has no DWARF memory space ID). */

        uint32_t   canonicalMemoryNumber; /* The canonical memory number as defined by the scheme
                                             specified in
                                             CADITargetFeatures_t::canonicalMemoryDescription.
                                             If the scheme is the empty string then no meaning can be
                                             ascribed to this field. */
};
```

3.8.8 CADIMemBlockInfo_t

This is a single block of memory addresses (inside a single memory space) that all have the same properties. For example, different memory blocks in the same memory space may be read-only. Blocks can be nested within one another. Blocks at the root level have CADI_MEMBLOCK_ROOT as the parent ID. Name is used to give the user an idea of the type of memory ("off chip", for example). If cyclesToAccess is 0, the number is unknown or irrelevant.

Example 3-12 CADIMemBlockInfo_t

```
struct CADIMemBlockInfo_t
{
public: // methods
    CADIMemBlockInfo_t(const char *name_par = "",
                const char *description_par = "", uint16_t id = 0, uint16_t parentID = 0,
                CADIAddrSimple_t startAddr = 0, CADIAddrSimple_t endAddr = 0,
                uint32_t cyclesToAccess = 0, CADIMemReadWrite_t readWrite = CADI_MEM_ReadWrite,
                uint32_t *supportedMultiplesOfMAU_ = 0, uint32_t endianness = 0,
                uint32_t invariance = 0) :
            id(id), parentID(parentID), startAddr(startAddr), endAddr(endAddr),
            cyclesToAccess(cyclesToAccess), readWrite(readWrite), endianness(endianness),
            invariance(invariance)
        {
            AssignString(name, name_par, CADI_NAME_SIZE);
            AssignString(description, description_par, CADI_DESCRIPTION_SIZE);
            if (supportedMultiplesOfMAU_)
                std::memcpy(supportedMultiplesOfMAU, supportedMultiplesOfMAU_,
                            sizeof(supportedMultiplesOfMAU));
            else
                std::memset(supportedMultiplesOfMAU, 0, sizeof(supportedMultiplesOfMAU));
        }

public: // data
```

```
    char          name[CADI_NAME_SIZE];      // Memory block name.
    char          description[CADI_DESCRIPTION_SIZE];  // Memory block description.
    uint16_t      id;                        // Memory block ID.
    uint16_t      parentID;                  // The ID of the parent. CADI_MEMBLOCK_ROOT if no parent.
    CADIAddrSimple_t  startAddr;             // The start address of this memory block.
    CADIAddrSimple_t  endAddr;               // The end address of this memory block.
    uint32_t      cyclesToAccess;           // Number of cycles needed for an access to this block.
    CADIMemReadWrite_t readWrite;           // The read/write type of this block
    uint32_t      supportedMultiplesOfMAU[CADI_MAU_MULTIPLES_LIST_SIZE];
                              // Allowed multiples of the bits per mau, measured in bits

    uint8_t    endianness;  // endianness, 0=same as owning memory space, 1=LE, 2=BE
    uint8_t    invariance;  // unit of invariance in bytes, 0=same as owning memory space
};
```


3.8.9 CADIPipeStage_t

The definition of CADIPipeStage_t is:

```
typedef struct CADIPipeStage_t
{
  uint32_t  id;          // Pipestage id
  char   name[CADI_NAME_SIZE];
  uint32_t  pc;          // Register id that holds the addr of the instruction
  uint32_t  contentInfoRegisterId;  // Register id that holds the current
                                    // content info for this pipe stage
  // (The register values correspond to the CADIPipeStageContentInfo_t enum)
} CADIPipeStage_t;
```

3.8.10 CADIPipeStageContentInfo_t

The definition of CADIPipeStageContentInfo_t is:

```
enum CADIPipeStageContentInfo_t
{
  CADI_PIPESTAGE_Invalid,     // This pipe stage is empty or invalid,
                              // nothing is displayed.
  CADI_PIPESTAGE_OpcodeOnly,  // An instruction is in this stage,
                              // only the opcode is valid.
  CADI_PIPESTAGE_DisassemblyOnly, // An instruction is in this stage,
                                  // only the disassembly is valid.
  CADI_PIPESTAGE_Instruction,     // An instruction is in this stage, both the
                                  // opcode and the disassembly are valid.
  CADI_PIPESTAGE_ENUM_COUNT,
  CADI_PIPESTAGE_MAX = 0xFFFFFFFF
};
```

3.8.11 CADIBptConfigure_t

The definition of CADIBptConfigure_t is:

```
typedef enum CADIBptConfigure_t
  {
    CADI_BPT_Disable,
    CADI_BPT_Enable
  } CADIBptConfigure_t;
```

3.8.12 CADIDisassemblerStatus

The CADIDisassemblerStatus enumeration is:

```
enum CADIDisasmStatus
{
  CADIDISASM_OK,              // disassembling completed successfully
  CADIDISASM_NO_INSTRUCTION,  // current address points to illegal
                              // instructions/data
  CADIDISASM_ILLEGAL_ADDRESS, // address out of range (memory read failed)
  CADIDISASM_ERROR,           // other error
  CADIDISASM_OK_ISCALL = -1   // disassembly completed successfully,
                              // instruction is a call
};
```

3.8.13 CADI_EXECMODE_t enumeration

The values in CADI_EXECMODE_t enumeration are shown in Example 3-13:

Example 3-13 CADI_EXECMODE_t

```
enum CADI_EXECMODE_t {
     // modeChange(CADI_EXECMODE_Stop): The simulation was in state 'running' and has now stopped.
     // This is always the last callback in a sequence of callbacks when the simulation stopped.
     // If the stop was because one or more breakpoints have been hit then this callback is preceded
     // by one or more modeChange(CADI_EXECMODE_Bpt, num) callbacks where 'num' specified the
     // breakpoint(s) being hit.
     // CADIExecStop() eventually results in a modeChange(CADI_EXECMODE_Stop) callback.
     // This callback implies a refresh(REGISTERS|MEMORY) callback which means that a debugger should
     // assume registers and memory to have changed.
     CADI_EXECMODE_Stop = 0,

     // modeChange(CADI_EXECMODE_Run): The simulation was in state 'stopped' and is now running.
     // CADIExecContinue() and CADIExecSingleStep() eventually result in a
     //  modeChange(CADI_EXECMODE_Run) callback.
     CADI_EXECMODE_Run = 1,

     // modeChange(CADI_EXECMODE_Bpt, num): A specific breakpoint was hit. The breakpoint number
     // 'num' of the breakpoint being hit is passed as the second parameter in the modeChange callback.
     // This callback may be called several times in a straight sequence if multiple breakpoints have
     // been hit at the same time. A modeChange(CADI_EXECMODE_Stop) callback is always following and
     // terminating this sequence, except when 'continueExecution' was true for all breakpoints being
     // hit. This callback does *not* mean that the simulation stopped. It may be followed by more
     // modeChange(CADI_EXECMODE_Bpt, num) callbacks. The final modeChange(CADI_EXECMODE_Stop) is
     // responsible for signaling that the simulation stopped.

     CADI_EXECMODE_Bpt = 2,
```



```
// modeChange(CADI_EXECMODE_Error) means: Same as modeChange(CADI_EXECMODE_Stop), but the model
// is in a state 'stopped and error' after this callback. This means that all execution control
// functions are disabled. CADIExecReset() needs to be called first to enable them again. This
// callback is not followed by another modeChange(CADI_EXECMODE_Stop) callback, it implies
// modeChange(CADI_EXECMODE_Stop).

// This callback implies a refresh(REGISTERS|MEMORY) callback which means that a debugger should
// assume registers and memory to have changed.
CADI_EXECMODE_Error = 3,

// Reserved for future use.
CADI_EXECMODE_HighLevelStep = 4,
// Reserved for future use.
CADI_EXECMODE_RunUnconditionally = 5,

// modeChange(CADI_EXECMODE_ResetDone): The CADIExecReset() request recently requested by a
// debugger is now complete. This is always the last callback in a sequence of callbacks cased
// by a CADIExecReset(). A modeChange(CADI_EXECMODE_Stop) might happen before this callback if the
// model was running when CADIExecReset() was issued. If a debugger which did not call
// CADIExecReset() receives this means that some other debugger or the simulation environment
// itself completed a reset. It is safe to ignore this callback, since the display update in the
// debugger is triggered by the refresh() callback.

// This callback implies a refresh(REGISTERS|MEMORY) callback which means that a debugger should
// assume registers and memory to have changed.
CADI_EXECMODE_ResetDone = 5,

CADI_EXECMODE_ENUM_MAX = 0xFFFFFFFF
};
```

3.8.14 CADIExecMode_t structure

This structure is used to return the execution mode.

```
struct CADIExecMode_t
{
public:
   CADIExecMode_t(uint32_t number = 0, const char *name_par = "") :
          number(number)
   {
      AssignString(name, name_par, CADI_NAME_SIZE);
   }
   uint32_t   number;
   char  name[CADI_NAME_SIZE];
}
```

number indicates the execution mode and must be one of:

- CADI_EXECMODE_Run for run mode
- CADI_EXECMODE_Bpt for breakpoint mode
- CADI_EXECMODE_Error if the target encounters errors and is not in a CADI call.

3.8.15 CADIFactoryErrorCode_t

The CADIFactoryErrorCode_t and CADIFactorySeverityCode_t types specify the values for the different error conditions.

Example 3-14 CADIFactoryErrorCode_t

```
enum CADIFactoryErrorCode_t
{
        CADIFACT_ERROR_OK,              // no error at all, message is empty
        // license checking
        CADIFACT_ERROR_LICENSE_FOUND_BUT_EXPIRED,
        CADIFACT_ERROR_LICENSE_NOT_FOUND,
        CADIFACT_ERROR_LICENSE_COUNT_EXCEEDED,
        CADIFACT_ERROR_CANNOT_CONTACT_LICENSE_SERVER,
        // always warning = true
        CADIFACT_ERROR_WARNING_LICENSE_WILL_EXPIRE_SOON,

         // for all other license errors
        CADIFACT_ERROR_GENERAL_LICENSE_ERROR,

        // info: the parameter which caused this error is indicated
        // in erroneousParameterId
        // dataType != dataType
        CADIFACT_ERROR_PARAMETER_TYPE_MISMATCH,
        CADIFACT_ERROR_PARAMETER_VALUE_OUT_OF_RANGE,
         // not out of range but still invalid
        CADIFACT_ERROR_PARAMETER_VALUE_INVALID,

        CADIFACT_ERROR_UNKNOWN_PARAMETER_ID,

         // for all other errors concerning a specific parameter
        CADIFACT_ERROR_GENERAL_PARAMETER_ERROR,

        // other, for everything else which prevented the CADI interface from
        // being created
        CADIFACT_ERROR_GENERAL_ERROR,

        // always warning = true, for everything else which still allowed the
        // CADI interface to be created
        CADIFACT_ERROR_GENERAL_WARNING,
```



```
        CADIFACT_ERROR_MAX = 0xFFFFFFFF
};
```

3.8.16 CADIFactorySeverityCode_t

The severity code is based on the to the error codes in CADIFactoryErrorCode_t and enables easy detection of errors and warnings

Example 3-15 CADIFactorySeverityCode_t

```
enum CADIFactorySeverityCode_t
{
        CADIFACT_SEVERITY_OK,           // no error at all, model created
        CADIFACT_SEVERITY_WARNING,      // only a warning, model still created
        CADIFACT_SEVERITY_ERROR,        // error, model not created
        CADIFACT_SEVERITY_MAX = 0xFFFFFFFF
};
```

3.8.17 CADISimulationInfo_t

This structure contains details about the simulation.

Example 3-16 CADISimulationInfo_t

```
struct CADISimulationInfo_t
{
    public: // methods
        CADISimulationInfo_t(uint32_t id = 0, const char *name_par = "",
                    const char *description_par = "") :  id(id)
        {
            AssignString(name, name_par, CADI_NAME_SIZE);
            AssignString(description, description_par, CADI_DESCRIPTION_SIZE);
        }

    public: // data
        uint32_t   id;                          // Used for identification
        char name[CADI_NAME_SIZE];              // name of simulation
        char description[CADI_DESCRIPTION_SIZE];    // simulation description

};
```

3.8.18 CADIBptRequest_t

The breakpoint request provides the PC address at which a breakpoint should occur and a string that describes the condition of the breakpoint. The target decides whether it can implement the breakpoint conditions.

Example 3-17 CADIBptRequest_t

```
struct CADIBptRequest_t
{
public: // methods
        CADIBptRequest_t(const CADIAddrComplete_t address = CADIAddrComplete_t(),
               uint64_t sizeOfAddressRange=0, int32_t enabled=0, const char *conditions_par = "",
               bool useFormalCondition = 1, CADIBptCondition_t formalCondition = CADIBptCondition_t(),
               CADIBptType_t type = CADI_BPT_PROGRAM, uint32_t regNumber = 0,
               int32_t temporary = false, uint8_t triggerType = 0,
                 uint32_t continueExecution = false) :
                 address(address), sizeOfAddressRange(sizeOfAddressRange), enabled(enabled),
                 useFormalCondition(useFormalCondition), formalCondition(formalCondition), type(type),
                 regNumber(regNumber), temporary(temporary), triggerType(triggerType),
                 continueExecution(continueExecution)
        {
            AssignString(conditions, conditions_par, CADI_DESCRIPTION_SIZE);
        }

public: // data
        CADIAddrComplete_t  address;         // The PC address at which the breakpoint should occur.
        uint64_t            sizeOfAddressRange; // Used only if type = CADI_BPT_PROGRAM_RANGE
        int32_t             enabled;            // Enable/Disable breakpoint
        char                conditions[CADI_DESCRIPTION_SIZE]; /* The breakpoint condition. Ultimately
                                                the target decides if it can implement breakpoint
                                                conditions.*/
        bool                useFormalCondition;     /* 0 = use free-form "conditions",
                                                       1 = use "formalCondition"*/
        CADIBptCondition_t  formalCondition;    // Formal conditions
        CADIBptType_t       type;               // Type
        uint32_t            regNumber;          // For register type only
        int32_t             temporary;          // Temporary breakpoint

        uint8_t             triggerType;        /* Allow breakpoints that trigger only on
                                                read/write/modify. This only has meaning for
                                                CADI_BPT_REGISTER and CADI_BPT_MEMORY breakpoints.
                                                The debugger should set this to zero for other
                                                breakpoint types.  Setting this to zero for
                                                CADI_BPT_REGISTER and CADI_BPT_MEMORY results in
                                                undefined behaviour and must not be done.   */
        uint32_t            continueExecution;  /* 1 = Continue execution after breakpoint has been hit
                                                This field should be obeyed by \e types of breakpoints,
```



```
                                        including CADI_BPT_INST_STEP, etc.    */
};
```

3.8.19 CADIParameterInfo_t and CADIParameterValue_t

CADIParameterInfo_t and CADIParameterValue_t structures are used to configure component parameters.

Example 3-18 CADIParameterInfo_t

```
struct CADIParameterInfo_t
{
public: // methods
    CADIParameterInfo_t(uint32_t id=0, const char *name_par="",
                    CADIValueDataType_t dataType=CADI_PARAM_INVALID,
                    const char *description_par = "", uint32_t isRunTime = 0,
            int64_t minValue = 0, int64_t maxValue = 0,
            int64_t defaultValue = 0,   const char *defaultString_par = "") :
            id(id), dataType(dataType), isRunTime(isRunTime),
            minValue(minValue), maxValue(maxValue), defaultValue(defaultValue)
        {
            AssignString(name, name_par, CADI_NAME_SIZE);
            AssignString(description, description_par, CADI_DESCRIPTION_SIZE);
            AssignString(defaultString, defaultString_par,
            CADI_DESCRIPTION_SIZE);
        }

public: // data
        uint32_t   id;                 // Used for identification
        char  name[CADI_NAME_SIZE];   // Name of the parameter
        CADIValueDataType_t dataType; // Data type for interpretation purposes
                                      //  of the debugger
        char  description[CADI_DESCRIPTION_SIZE];  // Parameter description
        uint32_t   isRunTime;          /* If equals to 0, then the parameter is
                                       instantiation-time only.
                                       If equals to 1, then the parameter can be
                                       changed at run-time */
        int64_t   minValue;            // minimum admissible value
        int64_t   maxValue;            // maximum admissible value
        int64_t   defaultValue;        // default value if parameter is type
                                       //  bool/int
        char  defaultString[CADI_DESCRIPTION_SIZE]; // default string if
                                       // parameter is type CADI_PARAM_STRING
};
```

Example 3-19 CADIParameterValue_t

```
struct CADIParameterValue_t
{
public: // methods
    CADIParameterValue_t(uint32_t parameterID = static_cast<uint32_t>(-1),
                    CADIValueDataType_t dataType=CADI_PARAM_INVALID,
                    int64_t intValue = 0,  const char *stringValue_par="") :
            parameterID(parameterID), dataType(dataType), intValue(intValue)
        {
            AssignString(stringValue, stringValue_par, CADI_DESCRIPTION_SIZE);
        }


public: // data
        uint32_t   parameterID;        // Refers to the id of respective
                                       //  CADIParameterInfo_t.
        CADIValueDataType_t dataType;  // Data type.
        int64_t   intValue;                   // This also contains the BOOL
                                              // (0 = false, 1 = true).
        char  stringValue[CADI_DESCRIPTION_SIZE];  // String value if type is
                                                   // string.
};
```

3.8.20 CADIReturn_t

This is the result returned by most calls and it is a general indication of the status of the call. When an error is detected, the debugger can call the CADIXfaceGetError API to retrieve an error message in text form.

Example 3-20 CADIReturn_t

```
enum CADIReturn_t
{

   CADI_STATUS_OK,                    // The call was successful.
   CADI_STATUS_GeneralError,          /* This indicates an error that isn't
                                      sufficiently explained by one of the
                                      other error status values.*/
   CADI_STATUS_UnknownCommand,        // The command is not recognized.
   CADI_STATUS_IllegalArgument,       // An argument value is illegal.
   CADI_STATUS_CmdNotSupported,       // The command is recognized but not
supported.
   CADI_STATUS_ArgNotSupported,       /* An argument to the command is
                                       recognized but not supported. For
```



```
                                    example, the target does not support
                                    a particular type of complex
                                    breakpoint.*/
   CADI_STATUS_InsufficientResources,  /* Not enough memory or other resources
                                       exist to fulfill the command.*/
   CADI_STATUS_TargetNotResponding,    /* A timeout has occurred across the
                                       CADI interface - the target did not
                                       respond to the command. */
   CADI_STATUS_TargetBusy,             /* The target received a request, but is
                                       unable to process the command. The caller
                                       can try this call again after some
                                       time.*/
   CADI_STATUS_BufferSize,             // Buffer too small (for char* types)
   CADI_STATUS_SecurityViolation,      // Request has not been fulfilled due to
                                       // a security violation
   CADI_STATUS_PermissionDenied,       // Request has not been fulfilled since
                                       // the permission was denied
   CADI_STATUS_ENUM_MAX = 0xFFFFFFFF   // Max enum value.
};
```

3.8.21 CADIBptCondition_t and CADIBptConditionOperator_t

Breakpoint comparison operations only apply to `CADI_BPT_MEMORY` and `CADI_BPT_REGISTER` breakpoints. Other breakpoints must always specify `CADI_BPT_COND_UNCONDITIONAL` as `conditionOperator`. Breakpoint conditions are always applied as a secondary condition after the primary condition of the breakpoint which depends on the breakpoint type and the trigger type.

`CADI_BPT_PROGRAM`, `CADI_BPT_PROGRAM_RANGE`, `CADI_BPT_INST_STEP`, `CADI_BPT_EXCEPTION` should obey the `ignoreCount` if the `useFormalCondition` is set. However, the debugger must ensure that `conditionOperator` is `CADI_BPT_COND_UNCONDITIONAL`, otherwise the behaviour is undefined.

Example 3-21 CADIBptCondition_t

```
struct CADIBptCondition_t
{
public: // methods
   CADIBptCondition_t(CADIBptConditionOperator_t conditionOperator =
       CADI_BPT_COND_UNCONDITIONAL, int64_t comparisonValue = 0,
       uint32_t threadID = 0, uint32_t ignoreCount = 0, uint32_t bitwidth = 0) :
       conditionOperator(conditionOperator), comparisonValue(comparisonValue),
       threadID(threadID), ignoreCount(ignoreCount), bitwidth(bitwidth)
        {
        }
```

```
public: // data
   CADIBptConditionOperator_t   conditionOperator;  // Operator for condition
   int64_t                      comparisonValue;    // Value to compare against
   uint32_t                      threadID;          // Reserved.
   uint32_t                      ignoreCount;
   uint32_t                      bitwidth;       // width of comparision value
};
```

The conditional breakpoint operations are enumerated in CADIBptConditionOperator_t.

Example 3-22 CADIBptConditionOperator_t

```
enum CADIBptConditionOperator_t
{
    CADI_BPT_COND_UNCONDITIONAL,   // Normal breakpoint, always break, no additional condition.
    CADI_BPT_COND_EQUALS,          // Only break if value == comparisionValue (unsigned comparison)
    CADI_BPT_COND_NOT_EQUALS,      // Only break if value != comparisionValue (unsigned comparison)

        // signed comparison
    CADI_BPT_COND_GREATER_THAN_SIGNED,            // Only break if value >  comparisionValue
    CADI_BPT_COND_GREATER_THAN_OR_EQUALS_SIGNED,  // Only break if value >= comparisionValue
    CADI_BPT_COND_LESS_THAN_SIGNED,               // Only break if value <  comparisionValue
    CADI_BPT_COND_LESS_THAN_OR_EQUALS_SIGNED,     // Only break if value <= comparisionValue

        // unsigned comparison
    CADI_BPT_COND_GREATER_THAN_UNSIGNED,             // Only break if value >  comparisionValue
    CADI_BPT_COND_GREATER_THAN_OR_EQUALS_UNSIGNED,   // Only break if value >= comparisionValue
    CADI_BPT_COND_LESS_THAN_UNSIGNED,                // Only break if value <  comparisionValue
    CADI_BPT_COND_LESS_THAN_OR_EQUALS_UNSIGNED,      // Only break if value <= comparisionValue
    CADI_BPT_COND_ENUM_COUNT,                        // Not a valid condition operator

        // legacy support, same as signed comparison
    CADI_BPT_COND_GREATER_THAN = CADI_BPT_COND_GREATER_THAN_SIGNED,
    CADI_BPT_COND_GREATER_THAN_OR_EQUALS = CADI_BPT_COND_GREATER_THAN_OR_EQUALS_SIGNED,
    CADI_BPT_COND_LESS_THAN = CADI_BPT_COND_LESS_THAN_SIGNED,
    CADI_BPT_COND_LESS_THAN_OR_EQUALS = CADI_BPT_COND_LESS_THAN_OR_EQUALS_SIGNED,

        // these are no breakpoint conditions:
    CADI_BPT_COND_ENUM_MAX = 0xFFFFFFFF
};
```


3.9 Accessing the debug interface from sc_main()

The CADI interface is typically used by attached debuggers. The interface can, however, be directly accessed from the top-level application that creates and connects the modules. This might be done, for example, to test the CADI interface or to write a customized debug function.

Example 3-23 shows the CADI methods that must be created for the component:

Example 3-23 CADI callback used by main()

```
class TopCADICallbackObj : public CADICallbackObj
{
public:
    TopCADICallbackObj ()     { }
     ~TopCADICallbackObj ()    { }

    virtual uint32_t appliOpen(const char *sFileName, const char *mode)
    {
        return 0;
    }

    virtual void appliInput (uint32_t streamId, uint32_t count,
                  uint32_t *actualCount, char *buffer)
    {
        // return value to debugger
        static int id = 100;
        sprintf (buffer, "Input:%d\n", id);
        (* actualCount) = (uint32_t) (strlen (buffer) + 1);
        ++ id;
    }

    virtual void appliOutput(uint32_t streamId, uint32_t count,
                 uint32_t * actualCount, const char *buffer)
    {
        //output text from debugger
        char * tmp = new char [count + 1];
        memcpy (tmp, buffer, count);
        tmp [count] = 0;
        printf (tmp);
        (* actualCount) = count;
    }

    virtual uint32_t appliClose(uint32_t streamID)
    {
        return 0;
    }
```

```
    virtual void doString(char *stringArg)    { }

    virtual void modeChange(uint32_t newMode, CADIBptNumber_t bptNumber)
    {
        if (newMode == CADI_EXECMODE_Stop)
        {
            // we are done, finish execution
            SEMINC(semaphore);
        }
    }

    virtual void reset(uint32_t resetLevel)    { }

    virtual void cycleTick(void)     { }

    virtual void killInterface(void)  { }
};
```

Example 3-24 shows the code that must be added to the top-level application to access CADI data:

Example 3-24 CADI code in main function

```
int sc_main (int argc, char *argv[])
{
    //Instantiate the modules
    .
    .
    .
    Slave_casi * s1 = new Slave_casi("Slave1");
    Slave_casi * s2 = new Slave_casi("Slave2");

    // setup CADI
    char enable [CADI_CB_Count];
    memset (& enable[0], 1, sizeof (enable));
    s1->getCADI ()->CADIXfaceAddCallback (new TopCADICallbackObj (), enable);
    s2->getCADI ()->CADIXfaceAddCallback (new TopCADICallbackObj (), enable);

    //Call init functions and setup memory map interface
    .
    .
    .
    //Connect the ports, call all interconnect functions, and reset
    .
    .
    .
    // start simulation
```



```
sc_start (0x8000); // stops at 0x4000
.
.
.
// get CADI object from port and read register data
CADI * cadi1 = s1->getCADI ();

uint32_t actual;
static CADIRegGroup_t regGroups [2];
result = cadi->CADIRegGetGroups(0, features.nrRegisterGroups,
             & actualCount, regGroups);
if (result != CADI_STATUS_OK || actualCount < 1)
{
    printf("CADIRegGetGroups call ..... failed!\n");
}

CADIRegGroup_t & regGroup = regGroups [0];
static CADIRegInfo_t regs [2];
result = cadi1->CADIRegGetMap (regGroups [0].groupID, 0, 2, & actual, regs);
if (result != CADI_STATUS_OK || actualCount < 1)
{
    printf("CADIRegGetMap call ..... failed!\n");
}

CADIRegInfo_t & regInfo = regInfos [0];
static CADIReg_t regs [1];
regs [0].regNumber = regInfo.regNumber;
CADIReg_t & reg = regs [0];
SEMOPEN (semaphore);
cadi->CADIExecContinue ();

CADIReg_t reg;
memset (& reg, 0, sizeof (CADIReg_t));
reg.regNumber = regs [1].regNumber;
cadi1->CADIRegRead (1, & reg, & actual, 0);
uint32_t tmp32 = 0;
tmp32 = (tmp32<<8) | reg.bytes[3];
tmp32 = (tmp32<<8) | reg.bytes[2];
tmp32 = (tmp32<<8) | reg.bytes[1];
tmp32 = (tmp32<<8) | reg.bytes[0];

printf ("CADI reg 0x%x\n", tmp32);

// terminate components
.
.
.

cadi->CADIExecStop ();
SEMDEC (semaphore);
```

```
    SEMCLOSE (semaphore);
    printf ("Closing CADI\n");
    cadi->CADIXfaceRelease (& refCount);
    DestroyCADIFactory (factory);

    fflush (stdout);
    fflush (stderr);
    return 0;
}
```

The module s1 is an instance of a Slave_casi component and it must provide the getCADI() function and CADI object (based on CADIBase) that enables accessing the registers and memory of the module as shown in Example 3-25.

Example 3-25 Slave_casi CADI functions

```
Slave_casi::Slave_casi(sc_module_name name, CASIModuleIF * parent)
                      : CASIModule(name, parent)
{
    // instantiate ports
    .
    .
    .
    // Create CADI object
    cadi = new CADISlave (this);
    .
    .
    .

}

CADI* Slave_casi::getCADI()
{
    return cadi;
}
```


The CADISlave class contains the access functions as shown in Example 3-26.

Example 3-26 CADISlave CADI functions

```
CADIReturn_t CADISlave::CADIRegGetGroups( uint32_t groupIndex,
        uint32_t desiredNumOfRegGroups,uint32_t* actualNumOfRegGroups,
        CADIRegGroup_t* grp )
{
    if ( groupIndex >= GROUP_COUNT )
    {
        return CADI_STATUS_IllegalArgument;
    }

    uint32_ti;
    for( i = groupIndex; ( i < groupIndex + desiredNumOfRegGroups ) && ( i <
        GROUP_COUNT ); i++ )
    {
        grp[i] = regGroup[i];
    }
    *actualNumOfRegGroups = i - groupIndex;
    return CADI_STATUS_OK;
}

CADIReturn_t CADISlave::CADIRegGetMap( uint32_t groupID, uint32_t regIndex,
           uint32_t registerSlots, uint32_t* registerCount, CADIRegInfo_t* reg )
{
    if ( groupID >= GROUP_COUNT )
    {
        return CADI_STATUS_IllegalArgument;
    }
    uint32_t i;
    uint32_tstart = regIndex + CADISlave_group_info[groupID].start;
    uint32_tend = regIndex + CADISlave_group_info[groupID].end;
    for ( i = 0; ( i < registerSlots ) && ( start + i <= end ); i++ )
    {
        reg[i] = regInfo[start + i];
    }
    *registerCount = i;
    return CADI_STATUS_OK;
}

CADIReturn_t CADISlave::CADIRegRead( uint32_t regCount, CADIReg_t* reg,
                     uint32_t* numRegsRead, uint8_t doSideEffects )
{
    UNUSEDARG(doSideEffects);
    uint32_ti;
    for ( i = 0; i < regCount; i++ )
    {
```

```
            uint16_t tmp16 = 0;
            uint32_t tmp32 = 0;
            uint64_t tmp64 = 0;
            switch (reg[i].regNumber)
            {
            case 0:
                //64-bit Register
                tmp64 = target->r_reg2;
                reg[i].bytes[0] = (uint8_t)((tmp64 >> 0) & 0xff);
                reg[i].bytes[1] = (uint8_t)((tmp64 >> 8) & 0xff);
                reg[i].bytes[2] = (uint8_t)((tmp64 >> 16) & 0xff);
                reg[i].bytes[3] = (uint8_t)((tmp64 >> 24) & 0xff);
                reg[i].bytes[4] = (uint8_t)((tmp64 >> 32) & 0xff);
                reg[i].bytes[5] = (uint8_t)((tmp64 >> 40) & 0xff);
                reg[i].bytes[6] = (uint8_t)((tmp64 >> 48) & 0xff);
                reg[i].bytes[7] = (uint8_t)((tmp64 >> 56) & 0xff);
                break;
            case 1:
                //32-bit Register
                tmp32 = target->r_reg3;
                reg[i].bytes[0] = (uint8_t)((tmp32 >> 0) & 0xff);
                reg[i].bytes[1] = (uint8_t)((tmp32 >> 8) & 0xff);
                reg[i].bytes[2] = (uint8_t)((tmp32 >> 16) & 0xff);
                reg[i].bytes[3] = (uint8_t)((tmp32 >> 24) & 0xff);
                break;
            case 2:
                //16-bit Register
                tmp16 = target->r_reg0;
                reg[i].bytes[0] = (uint8_t)((tmp16 >> 0) & 0xff);
                reg[i].bytes[1] = (uint8_t)((tmp16 >> 8) & 0xff);
                break;
            case 3:
                //8-bit Register
                tmp16 = target->r_reg1;
                reg[i].bytes[0] = (uint8_t)((tmp16 >> 0) & 0xff);
                break;
            default:
                break;
            }
        }
        *numRegsRead = regCount;
        return CADI_STATUS_OK;
}
```


Chapter 4
The Cycle Accurate Profiling Interface

This chapter describes how to use the *Cycle Accurate Profiling Interface* (CAPI) to gather customized profiling data and to enable the user to visualize that data. It contains the following sections:

4.1 Introduction to CAPI

The CAPI interface supports a generic implementation of profiling and enables the collection of different types of data that are organized around streams and channels of information.

To support profiling, a component must:

- implement and register the CAPI interface
- supply the type of information to profile
- gather the information by calling, for example, `CAPIRecordEvents()`.

Depending on the simulation environment, the collected information from components can be displayed in profile windows or transferred to an external file.

Note

OSCI does not inherently support for collecting or displaying profile information.

4.1.1 Profiling streams and channels

The CAPI interface implemented by a component has one or more profiling streams. Each stream contains one thread of related information. For instance, a bus component might allocate a profiling stream to each master to collect:

- grants
- conflicts
- reads
- writes.

Each profiling stream is composed of one or more profiling channels. Each channel represents one item of information to be collected and placed in the stream.

The Cache component in *Example CAPI implementation* on page 4-28, for example, has one stream with two channels:

channel 1 records the address.

channel 2 records the operation type as one of:

- Read misses
- Read hits
- Write misses
- Write hits
- Refills
- Writebacks.

The operation channels can be displayed graphically.

4.2 The CAPI classes

The relationship between the various CAPI data structures is shown in Figure 4-1. (The `eventsTrace` data blocks have three data entries in this example.)

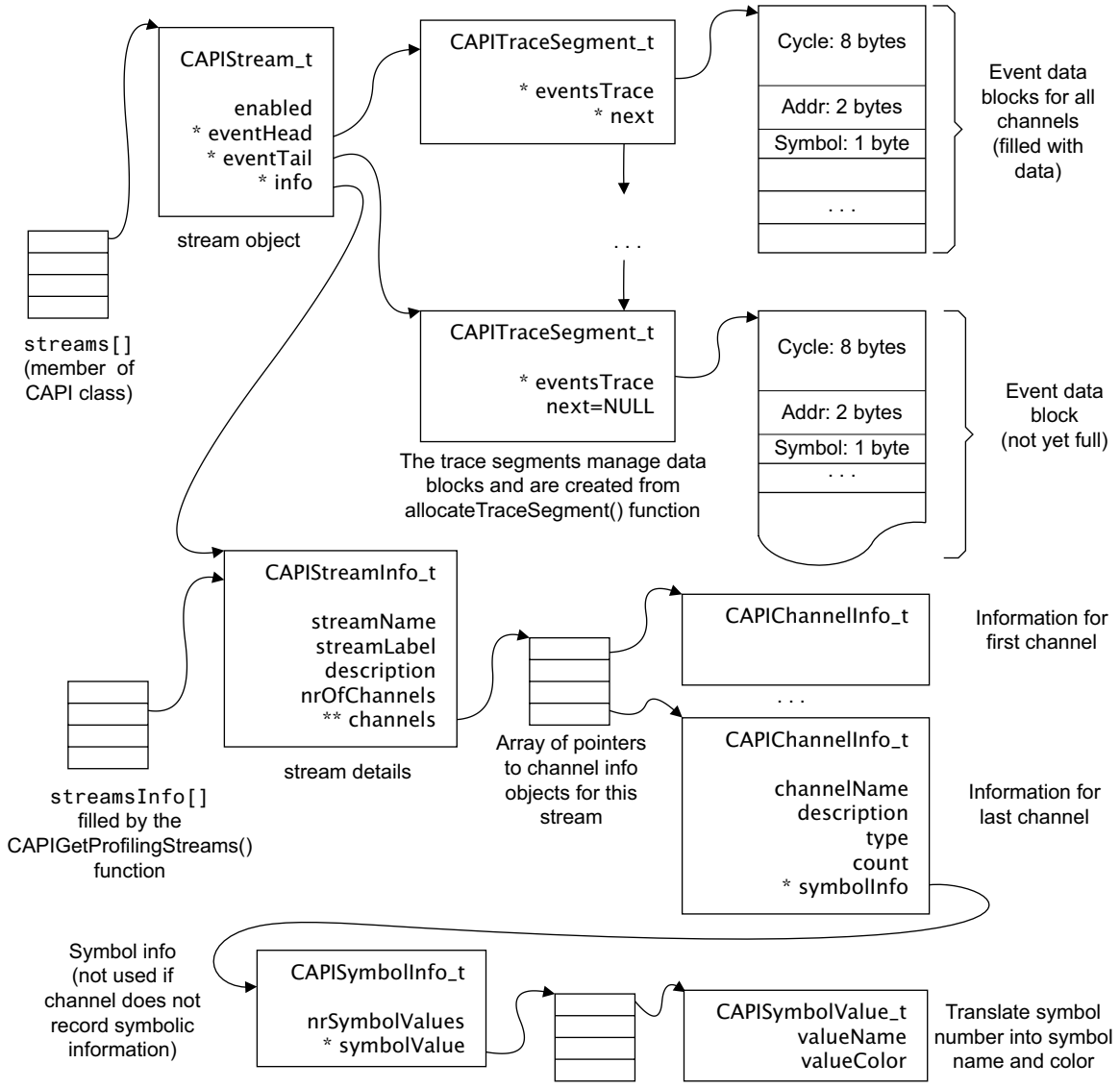

Figure 4-1 CAPI data structures and the profiling stream

The CAPI class hierarchy is shown in Figure 4-2.

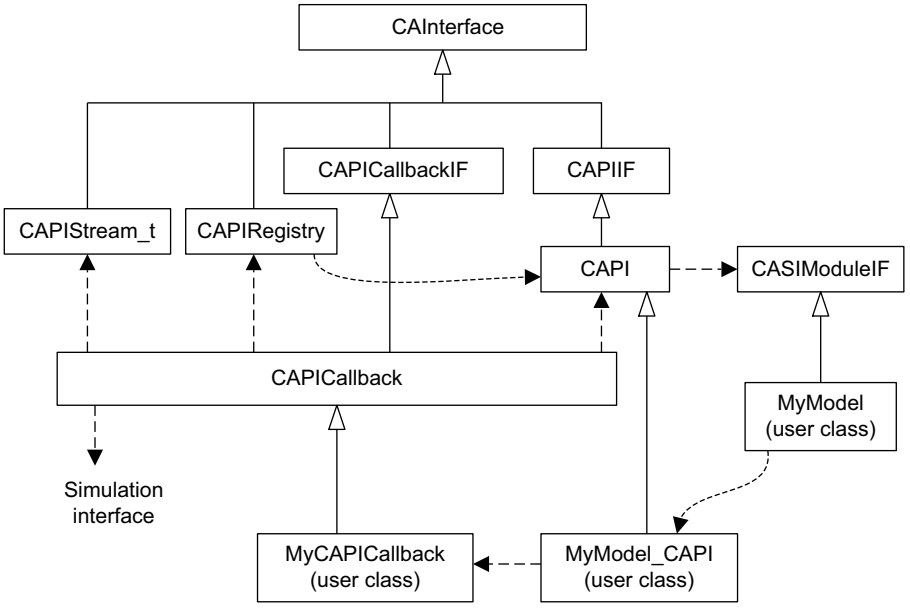

Figure 4-2 CAPI class hierarchy

Note

See the `CAPITypes.h` file for definitions of enumerations and data structures that are used with the CAPI interface.

Supporting profiling requires that a component:

- implement a class inheriting from CAPI,
- specify the information regarding the information to be profiled (such as the profiling streams and channels to be collected).

The derived CAPI class *CompName*`_CAPI` for your component (named *CompName*) must implement an empty constructor. Calling the base class generates calls to the CAPI constructor and the function `CAPIGetProfilingStreams()` that returns the profiling stream to be collected for this component.

```
CompName_CAPI::CompName_CAPI(eslapi::CASIModuleIF *comp):CAPI(comp)
{
}
```

```
CAPIReturn_t CompName_CAPI::CAPIGetProfilingStreams(uint32_t desiredNrStreams,
                        uint32_t *actualNrStreams,
                        eslapi::CAPIStreamInfo_t *streams);
```

The following steps show a simplified sequence that describes how to use the CAPI interface in a model during the simulation stage:

1. Get a pointer to the CAPI interface by calling `getCAPI()`.

   ```
   CAPI * capi = CompName->getCAPI()
   ```

2. Get the streams metadata by:

   ```
   uint32_t actualNrStreams;
   CAPIStreamInfo_t streamsInfo[100];
   capi->CAPIGetProfilingStreams(100, &actualNrStreams, streamsInfo);
   ```

3. A specific stream can be obtained from the metadata by selecting a specific stream by its name. Use either the literal name for the stream or the name from the `streamInfo` structure:

   ```
   CAPIStream_t * stream0 = capi->CAPIFindStream("Accesses");

   CAPIStream_t * stream0 = capi->CAPIFindStream(streamsInfo[0].streamName);
   ```

4. The returned pointer can be used to enable profiling stream:

   ```
   stream0->enabled = true;
   ```

5. The pointer is also used to collect information when an event occurs:

   ```
   CAPIRecordEvent2(stream0, (uint16_t) addr, (uint8_t) COT_READ_HIT);
   ```

For more details on using CAPI, see *Example CAPI implementation* on page 4-28.

4.2.1 The CAPI class

Example 4-1 shows the CAPI class definition.

Example 4-1 CAPI class header file

```
class CAPI : public CAPIIF
{
public:
    CAPI (CASIModuleIF * _target =NULL);
    virtual ~CAPI () {}
    virtual CAPIReturn_t CAPIGetProfilingStreams (uint32_t desiredNrStreams,
            uint32_t * actualNrStreams, CAPIStreamInfo_t * streamsInfo
            int streamIndex) = 0;
    virtual CAPIStream_t * CAPIFindStream (const char * name);
    virtual uint32_t CAPIGetNumStreams(void);
```



```
    virtual const CAPIStream_t * CAPIGetStream(uint32_t index);
    virtual uint32_t CAPIGetNumStreams(void);
    virtual CAPIStream_t * CAPIGetStream(uint32_t index);
    void CAPIAddStream (CAPIStream_t * );
    void CAPIClearStreams (void );

    virtual CASIModuleIF* CAPIGetTarget();

    // Return interface if requested
    virtual CAInterface * ObtainInterface(if_name_t ifName,
                  if_rev_t minRev, if_rev_t * actualRev)
            {
                if((strcmp(ifName,"eslapi.CAPI2") == 0) && (minRev <= 0)){
                    *actualRev = 0;
                    return this;
                }
                return NULL;
            }

protected:
    CASIModuleIF * target;
    std::vector <CAPIStream_t *> streams;

};
```

4.2.2 CAPI::CAPIGetProfilingStreams()

Returns the metadata associated with the profiling streams. Each CAPI objects defines the structure of its streams by providing an appropriate implementation.

Example usage:

```
virtual CAPIReturn_t CAPI::CAPIGetProfilingStreams (uint32_t desiredNrStreams,
         uint32_t * actualNrStreams, CAPIStreamInfo_t * streamsInfo
         int streamIndex) =0
```

where:

`desiredNrStreams`

is the size of the streams array. Call with a relatively large number to retrieve all the streams metadata.

`actualNrStreams`

returns the number of streams defined in the component.

`streamsInfo` returns the metadata for the first `desiredNrStreams` defined by the CAPI object.

`streamIndex` is the index to the point in the stream array from which the streams are returned.

Example 4-2 Getting the profiling streams

```
uint32_t nrStreams;
CAPIStreamInfo_t streamsInfo [100];
capi->CAPIGetProfilingStreams (100, & nrStreams, streamsInfo, 0);
```

4.2.3 CAPI::CAPIFindStream()

Returns the profiling stream associated with *name*.

```
virtual CAPIStream_t* CAPI::CAPIFindStream (const char * name)
```

where:

`name` matches the value in `stream->info->streamName`.

4.2.4 CAPI::CAPIGetNumStreams()

Returns the total number of streams in this CAPI interface.

```
virtual CAPIStream_t * CAPIGetStream(uint32_t index)
```

4.2.5 CAPI::CAPIGetStream()

Returns the stream for a given index.

```
virtual const CAPIStream_t * CAPIGetStream(uint32_t index)=0
```

where:

`index` is the index of the stream to return. (`index` must be between 0 and `CAPIGetNumStreams() - 1`).

4.2.6 CAPI::CAPIGetStream()

Add a stream to the stream collection.

```
void CAPIAddStream (CAPIStream_t * stream)
```

where:

`stream` is a pointer to the stream to add.

4.2.7 CAPI::CAPIClearStreams()

Removes all streams from this interface.

```
void CAPIClearStreams (void ))
```

4.2.8 CAPI::CAPIGetTarget()

Gets target module for the profile interface.

```
virtual CASIModuleIF* CAPIGetTarget())
```

4.2.9 CAPIRecordEvent1() and CAPIRecordEvent2()

Two macros are defined to simplify recording data to the profile stream:

- `CAPIRecordEvent1()` records a single argument to the specified stream.
- `CAPIRecordEvent2()` records two arguments to the specified stream.

Note

The `MXSI_EXPORT_LIBRARY` is provided to aid converting MXPI profiling to CAPI. This library contains an implementation of `CAPIRecordEvent()` that accepts a variable number of arguments.

Example 4-3 CAPIRecordEvent1() macro

```
#define CAPIRecordEvent1(stream,arg1) \
{\
  uint64_t currentCycle; \
  uint64_t offset=0,base=0;  \
  uint8_t temp8; \
  uint16_t temp16; \
  uint32_t temp32;  \
  uint64_t temp64; \
\
 if(stream->enabled){\
  if(((uint32_t) stream->nrOfEventsInLastSegment)>= \
             ((uint32_t) stream->eventsTraceSegmentSize)) \
  {\
    stream->eventTail->next = \
    eslapi::CAPIRegistry::getCAPIRegistry()->getCAPICallback()->allocateTraceSegment (stream); \
    stream->eventTail = stream->eventTail->next; \
    stream->eventTail->next = NULL; \
    stream->nrOfEventsInLastSegment = 0; \
  } \
\
```

```
    currentCycle = eslapi::CAPIRegistry::getCAPIRegistry()->getCAPICallback()->getCurrentCycle(); \
\
  base = stream->nrOfEventsInLastSegment * ((uint32_t) stream->eventWidthInBytes); \
  offset = 0; \
\
  *((uint64_t*)&(stream->eventTail->eventsTrace[base + offset])) = \
         (uint64_t) currentCycle; \
  offset += 8; \
      \
  switch(stream->info->channels[1]->type){ \
  case eslapi::CAPI_CHANNEL_TYPE_U8: \
  case eslapi::CAPI_CHANNEL_TYPE_bool: \
  case eslapi::CAPI_CHANNEL_TYPE_SYMBOL: \
    temp8 = (uint8_t) arg1; \
    *((uint8_t*)&(stream->eventTail->eventsTrace[base + offset])) =  temp8; \
    offset += 1; \
    break;  \
  case eslapi::CAPI_CHANNEL_TYPE_U16: \
    temp16 = (uint16_t) arg1; \
    *((uint16_t*)&(stream->eventTail->eventsTrace[base + offset])) = temp16; \
    offset += 2; \
    break; \
  case eslapi::CAPI_CHANNEL_TYPE_U32: \
    temp32 = (uint32_t) arg1; \
    *((uint32_t*)&(stream->eventTail->eventsTrace[base + offset])) = temp32; \
    offset += 4; \
    break; \
  case eslapi::CAPI_CHANNEL_TYPE_U64: \
    temp64 = (uint64_t) arg1; \
    *((uint64_t*)&(stream->eventTail->eventsTrace[base + offset])) = temp64; \
    offset += 8; \
    break; \
  default: \
    assert(0); \
  }\
  stream->nrOfEventsInLastSegment ++; \
  stream->nrOfEvents ++; \
 }\
}
```

Example 4-4 CAPIRecordEvent2() macro

```
#define CAPIRecordEvent2(stream,arg1,arg2) \
{\
  uint64_t currentCycle; \
  uint64_t offset=0,base=0;  \
```



```
  uint8_t temp8; \
  uint16_t temp16; \
  uint32_t temp32;  \
  uint64_t temp64; \
\
 if(stream->enabled){\
  if(((uint32_t) stream->nrOfEventsInLastSegment) >=
          ((uint32_t) stream->eventsTraceSegmentSize)) { \
    stream->eventTail->next = \
    eslapi::CAPIRegistry::getCAPIRegistry()->getCAPICallback()->allocateTraceSegment (stream); \
    stream->eventTail = stream->eventTail->next; \
    stream->eventTail->next = NULL; \
    stream->nrOfEventsInLastSegment = 0; \
  } \
\
  currentCycle = eslapi::CAPIRegistry::getCAPIRegistry()->getCAPICallback()->getCurrentCycle(); \
\
  base = stream->nrOfEventsInLastSegment * ((uint32_t) stream->eventWidthInBytes); \
  offset = 0; \
\
  *((uint64_t*)&(stream->eventTail->eventsTrace[base + offset])) = \
             (uint64_t) currentCycle; \
  offset += 8; \
      \
  switch(stream->info->channels[1]->type){ \
  case eslapi::CAPI_CHANNEL_TYPE_U8: \
  case eslapi::CAPI_CHANNEL_TYPE_bool: \
  case eslapi::CAPI_CHANNEL_TYPE_SYMBOL: \
    temp8 = (uint8_t) arg1; \
    *((uint8_t*)&(stream->eventTail->eventsTrace[base + offset])) =  temp8; \
    offset += 1; \
    break;  \
  case eslapi::CAPI_CHANNEL_TYPE_U16: \
    temp16 = (uint16_t) arg1; \
    *((uint16_t*)&(stream->eventTail->eventsTrace[base + offset])) = temp16; \
    offset += 2; \
    break; \
  case eslapi::CAPI_CHANNEL_TYPE_U32: \
    temp32 = (uint32_t) arg1; \
    *((uint32_t*)&(stream->eventTail->eventsTrace[base + offset])) = temp32; \
    offset += 4; \
    break; \
  case eslapi::CAPI_CHANNEL_TYPE_U64: \
    temp64 = (uint64_t) arg1; \
    *((uint64_t*)&(stream->eventTail->eventsTrace[base + offset])) = temp64; \
    offset += 8; \
    break; \
  default: \
    assert(0); \
  }\
```

```
      \
  switch(stream->info->channels[2]->type){ \
  case eslapi::CAPI_CHANNEL_TYPE_U8: \
  case eslapi::CAPI_CHANNEL_TYPE_bool: \
  case eslapi::CAPI_CHANNEL_TYPE_SYMBOL: \
    temp8 = (uint8_t) arg2; \
    *((uint8_t*)&(stream->eventTail->eventsTrace[base + offset])) =  temp8; \
    offset += 1; \
    break;  \
  case eslapi::CAPI_CHANNEL_TYPE_U16: \
    temp16 = (uint16_t) arg2; \
    *((uint16_t*)&(stream->eventTail->eventsTrace[base + offset])) = temp16; \
    offset += 2; \
    break; \
  case eslapi::CAPI_CHANNEL_TYPE_U32: \
    temp32 = (uint32_t) arg2; \
    *((uint32_t*)&(stream->eventTail->eventsTrace[base + offset])) = temp32; \
    offset += 4; \
    break; \
  case eslapi::CAPI_CHANNEL_TYPE_U64: \
    temp64 = (uint64_t) arg2; \
    *((uint64_t*)&(stream->eventTail->eventsTrace[base + offset])) = temp64; \
    offset += 8; \
    break; \
  default: \
    assert(0); \
  }\
  stream->nrOfEventsInLastSegment ++; \
  stream->nrOfEvents ++; \
 }\
}
```


4.3 The CAPIRegistry class

Registers a CAPI interface with the default CAPI implementation. During registration, the appropriate streams are created according to the `CAPIGetProfilingStreams` definition. A model must register its CAPI object during construction. The header file for the class is shown in Example 4-5.

Example 4-5 CAPIRegistry class

```
class WEXP CAPIRegistry: public CAInterface{
public:
       CAPIRegistry();
       // Return the CAInterface name for this interface.
       static if_name_t IFNAME() { return "eslapi.CAPIRegistry2"; }
       // Specify the current minor revision for this interface.
       static if_rev_t IFREVISION() { return 0; }

       // Registers a CAPI interface with the default CAPI implementation.
       void CAPIRegisterInterface (CAPI * capi);
       /// Unregisters a CAPI interface.
       void CAPIUnregisterInterface (CAPI *capi);
       //Call this to get the CAPI interface for a component
       CAPI * CAPIFindInterface(char * comp_name);
       uint32_t CAPIGetNumInterfaces(void);
       CAPI * CAPIGetInterface(uint32_t index);
       // Reset the CAPI information
       void reset(void);
       // Remove all CAPI information
       void clear(void);

       // Memory Management Functions
       // Sets a new global profiling stream data memory manager
       void setCAPICallback (CAPICallback * _callback) { callback = _callback; }
       // Returns the current global profiling stream data memory manager
       inline CAPICallback * getCAPICallback () { return callback; }
       //  Returns the static capiRegistry object
      inline static CAPIRegistry * getCAPIRegistry(void){
            if(capiRegistry == NULL) capiRegistry = new CAPIRegistry();
            return capiRegistry;
       }
       // Versioning base implementation
       // Return interface if requested
       virtual CAInterface * ObtainInterface(
            if_name_t    ifName,
            if_rev_t     minRev,
            if_rev_t *   actualRev)
            {
```

```
                if((strcmp(ifName,"eslapi.CAPIRegistry2") == 0) && (minRev <= 0))
                {
                    *actualRev = 0;
                    return this;
                }
                return NULL;
            }
    protected:
        static CAPIRegistry * capiRegistry;
        CAPICallback * callback;
        std::vector<CAPI*> interfaces;
};
```

4.3.1 CAPIRegistry::setCAPICallback()

Sets a new global profiling stream data memory manager.

```
static void CAPIRegistry::setCAPICallback (CAPICallback * memoryManager)
```

where:

`memoryManager`

is the memory manager callback object.

4.3.2 CAPIRegistry::getCAPICallback()

Returns the current global profiling stream data memory manager.

```
static CAPICallback* CAPIRegistry::getCAPICallback ()
           { return callback; }
```

4.3.3 CAPIRegistry::CAPIRegisterInterface()

Registers a CAPI interface with the default CAPI implementation. During registration, the appropriate streams are created according to the `CAPIGetProfilingStreams` definition. A model must register its CAPI object during construction.

```
static void CAPIRegistry::CAPIRegisterInterface (CAPI * capi)
```


4.3.4 CAPIRegistry::CAPIUnregisterInterface()

Unregisters a CAPI interface.

```
void CAPIUnregisterInterface (CAPI *capi)
```

where:

`capi` is the interface to unregister.

4.3.5 CAPIRegistry::CAPIFindInterface()

Gets the CAPI interface for a component.

```
CAPI * CAPIFindInterface(char * comp_name)
```

where:

`comp_name` is the component name.

4.3.6 CAPIRegistry::CAPIGetNumInterfaces()

Get the number of interfaces in use.

```
uint32_t CAPIGetNumInterfaces(void)
```

4.3.7 CAPIRegistry::CAPIGetInterface()

Gets an interface based on its index in the collection.

```
CAPI * CAPIGetInterface(uint32_t index)
```

where:

`index` is the index into the list of streams.

4.3.8 CAPIRegistry::reset()

Resets the CAPI information.

```
void reset(void)
```

4.3.9 CAPIRegistry::clear()

Removes all CAPI information.

```
void clear(void)
```

4.3.10 CAPIRegistry::getCAPIRegistry()

Returns the static `capiRegistry` object.

```
inline static CAPIRegistry * getCAPIRegistry(void){
            if(capiRegistry == NULL) capiRegistry = new CAPIRegistry();
            return capiRegistry;
        }
```


4.4 The CAPICallback class

The CAPICallback establishes a basic relation between CAPI and the simulation kernel. There exists a single unique CAPICallback that applies to all CAPI objects.

Example 4-6 CAPICallback definition

```
class CAPICallback : public CAPICallbackIF{
public:
      // Return the CAInterface name for this interface.
      static if_name_t IFNAME() { return "eslapi.CAPICallback2"; }
      // Specify the current minor revision for this interface.
      static if_rev_t IFREVISION() { return 0; }
      virtual CAPITraceSegment_t * allocateTraceSegment
                (const CAPIStream_t * stream)
            {return NULL;}
      virtual void deallocateTraceSegment (const CAPIStream_t * stream,
                CAPITraceSegment_t * segment) {}
      virtual uint64_t getCurrentCycle (){return (uint64_t)-1;}
      virtual uint64_t getTotalAllocatedMemory(void) {return 0;}
      // Return interface if requested
      virtual CAInterface * ObtainInterface( if_name_t ifName,
          if_rev_t minRev, if_rev_t * actualRev)
      {
         if((strcmp(ifName,"eslapi.CAPICallback2") == 0) && (minRev <= 0))
         {
             *actualRev = 0;
              return this;
         }
         return NULL;
     }
};
```

4.4.1 CAPICallback::allocateTraceSegment()

This function allocates a trace segment. Profiling data might grow very large for long simulation. By setting an appropriate memory manager the CAPI client has full control of how much of the profile data resides in memory at a given time

```
virtual CAPITraceSegment_t* CAPICallback::allocateTraceSegment (
        CAPIStream_t * stream) { return NULL }
```

where:

`stream` is the stream that uses this segment.

Note

A profiling stream must have an active `CAPITraceSegment_t` at all times.The size of the trace must be:

`(stream->eventWidthInBytes) * (stream->eventsTraceSegmentSize).`

The default implementation returns a NULL pointer.

4.4.2 CAPICallback::deallocateTraceSegment()

Deallocates a trace segment that was previously allocated.

```
virtual void deallocateTraceSegment (const CAPIStream_t * stream,
            CAPITraceSegment_t * segment) {}
```

where:

`stream` is the stream for the trace segment.

`segment` is the trace segment.

4.4.3 CAPICallback::getTotalAllocatedMemory()

Return a count of the total memory allocated for profiling.

```
virtual uint64_t getTotalAllocatedMemory(void) {return 0;}
```

4.4.4 CAPICallback::getCurrentCycle()

This function returns the current simulation cycle.

```
virtual uint64_t CAPICallback::getCurrentCycle () {return (uint64_t)-1;}
```

A cycle represents the moment in time a profiled event is recorded. You can define a custom cycle or timing definition by providing an appropriate implementation for this callback.

4.5 CAPI data structures

This section describes the classes and data structures used by the CAPI interface.

4.5.1 The CAPIStream_t structure

The CAPIStream_t structure specifies the information for an individual profiling stream. Each CAPI interface implementation might contain one or more such streams.

Example 4-7 CAPIStream_t structure

```
class CAPIStream_t : public CAInterface
{
    CAPIStream_t(CAPI * _owner, CAPIStreamInfo_t * _info,
          uint64_t _segmentSize = CAPI_DEFAULT_SEGMENT_SIZE);

   inline static CAPIReturn_t recordEvent (CAPIStream_t *stream, ... );

   // Return the CAInterface name for this interface.
   static if_name_t IFNAME() { return "eslapi.CAPIStream_t2"; }

   // Specify the current minor revision for this interface.
   static if_rev_t IFREVISION() { return 0; }
   // Return interface if requested
   virtual CAInterface * ObtainInterface( if_name_t ifName, if_rev_t minRev,
        if_rev_t *   actualRev)
        {
           if((strcmp(ifName,"eslapi.CAPIStream_t2") == 0) && (minRev <= 0)){
                    *actualRev = 0;
                    return this;
            }
        return NULL;
        }

    CAPIStreamInfo_t * info;
    CAPI *owner;
    bool enabled;
    uint64_t eventWidthInBytes;
    uint64_t nrOfEvents;
    uint64_t nrOfEventsInLastSegment;
    CAPITraceSegment_t *eventHead;
    CAPITraceSegment_t *eventTail;
    uint64_t eventsTraceSegmentSize;
    void * userData;
};
```

4.5.2 The CAPIStreamInfo_t structure

The CAPIStreamInfo_t structure specifies the channel information for a profiling stream. Each stream can contain one or more channels. Use CAPIFindStream(*stream_name*) to return a stream with the specified name from streams[] array that contains all of the profiling streams for the component.

Example 4-8 CAPIStreamInfo_t structure

```
struct CAPIStreamInfo_t{
    char *streamName; //The ID Name of the stream
    char *streamLabel; //The name displayed in the profiling view of the stream
    char * description; //The description of the stream
    uint32_t nrOfChannels; //The number of channels
    CAPIChannelInfo_t **channels; //The channels data structure
};
```

4.5.3 The CAPIChannelInfo_t structure

The CAPIChannelInfo_t structure in Example 4-9 specifies the information for a profiling channel. Each profiling stream can contain one or more such channels.

Example 4-9 CAPIChannelInfo structure

```
struct CAPIChannelInfo_t{
    char *channelName; //the channel name
    char * description; //the channel description
    CAPIChannelType_t type; //the type of the data elements for this channel
    int count; //number of the data elements of this type (default is 1)

    //For type CAPI_CHANNEL_TYPE_SYMBOL:
    // symbol values and colors
    CAPIChannelSymbolInfo_t * symbolInfo;
    // The high-level (conceptual) information type that this channel
    // describes (e.g., an ADDRESS value, a delay, etc.).
    CAPIChannelInfoType_t infoType;
};
```


Caution

If you are profiling on a workstation running the Solaris operating system, a limitation of the system prevents misaligned data reads and writes. The channels declared in a stream must be in descending order of their byte size. For every CAPI stream, place channels of 64-bit size before channels of 32-bit size, and so forth.

4.5.4 The CAPIChannelType_t enumeration

The `CAPIChannelType` enumeration in Example 4-10 specifies the possible types for the profiling channels.

Example 4-10 CAPIChannelType_t

```
enum CAPIChannelType_t{
    CAPI_CHANNEL_TYPE_bool, //BOOL
    CAPI_CHANNEL_TYPE_SYMBOL, //A symbol with values 0, 1, 2, ..., and
                              // associated strings in CAPIChannelSymbolInfo_t
    CAPI_CHANNEL_TYPE_U8, //unsigned char
    CAPI_CHANNEL_TYPE_U16, //unsigned short
    CAPI_CHANNEL_TYPE_U32, //unsigned int
    CAPI_CHANNEL_TYPE_U64, //unsigned long long
    CAPI_CHANNEL_TYPE_S8, //signed char
    CAPI_CHANNEL_TYPE_S16, //signed short
    CAPI_CHANNEL_TYPE_S32, //signed int
    CAPI_CHANNEL_TYPE_S64  //signed long long
    CAPI_CHANNEL_TYPE_F32,  ///< floating point (32 bit)
    CAPI_CHANNEL_TYPE_F64  ///< floating point double (64 bit)
};
```

The channel types `BOOL`/`U8`/`U16`/… represent boolean, unsigned byte, half-word, or other types as represented by the host machine.

The channel type `CAPI_CHANNEL_TYPE_SYMBOL` is stored in memory as a byte, and represents possible enumeration values, starting with 0. Each such enumeration value has an associated symbol name and color. The symbol names and colors show the legend and the bar charts in the profiling view.

Note

The CAPI view windows support the following types of channels on the X and Y axes:

- The X axis supports CAPI channels of numeric types (for example, `CAPI_CHANNEL_TYPE_U8`, `CAPI_CHANNEL_TYPE_U16`, and so forth).
- The Y axis supports channels of numeric types and type `CAPI_CHANNEL_TYPE_SYMBOL`.

4.5.5 The CAPIChannelSymbolInfo_t structure

The `CAPIChannelSymbolInfo` specifies the values associated with a channel of type `CAPI_CHANNEL_TYPE_SYMBOL`.

Example 4-11 CAPIChannelSymbolInfo_t structure

```
struct CAPIChannelSymbolInfo_t{
    int nrSymbolValues; //the number of symbol values in the symbolValues
                        //array of values
    CAPISymbolValue_t *symbolValues; //the symbol values array, associated with
                        //values of 0, 1, 2, ...
};
```

4.5.6 The CAPISymbolValue_t structure

The `CAPISymbolValue` specifies the value names and colors associated with the values of a symbol, for channels of type `CAPI_CHANNEL_TYPE_SYMBOL`.

Example 4-12 CAPISymbolValue_t structure

```
struct CAPISymbolValue_t{
    char *valueName;   //The name of the values for this symbol channel
                       //(corresponding to the values of 0, 1, 2, ...)
    uint32_t valueColor; //CAPIColor_e defines the colors for this value
};
```

Note

The contents of `valueColor` give a hint to the color used. The `CAPIColor_e` enumeration is not used directly for binary compatibility reasons.

4.5.7 The CAPIColor_e enumeration

The CAPIColor_e specifies the colors available for values of channels of type CAPI_CHANNEL_TYPE_SYMBOL. Example 4-13 lists the first part of the enumeration, for the full list of colors supported, see the CAPIColors.h header file.

Example 4-13 CAPIColor_e enumeration

```
enum CAPIColor_e{
    CAPI_COLOR_WHITE,
    CAPI_COLOR_BLACK,
    CAPI_COLOR_RED,
    CAPI_COLOR_GREEN,
    CAPI_COLOR_BLUE,
          . . .
}
```

Use the getCAPIColorName() function to return the human-readable name for a color value:

```
const char* getCAPIColorName (CAPIColor_e color)
```

4.5.8 The CAPITraceSegment_t structure

This structure is used for nodes in the profile data list. Each node in the list, except for the last node, stores eventsTraceSegmentSize number of events. The last element in the list however, only stores nrOfEventsInLastSegment events.

Example 4-14 CAPITraceSegment_t structure

```
struct CAPITraceSegment_t{
        uint8_t *eventsTrace;     // Profile data block
        CAPITraceSegment_t *next; // Next node in list
    };
```

4.5.9 The CAPIReturn_t enumeration

This enumeration contains the return values for CAPI functions.

Example 4-15 CAPIReturn_t

```
enum CAPIReturn_t{
        CAPI_STATUS_OK,
        CAPI_STATUS_GeneralError,
        CAPI_STATUS_UnknownCommand,
        CAPI_STATUS_IllegalArgument,
        CAPI_STATUS_CmdNotSupported,
        CAPI_STATUS_ArgNotSupported,
        CAPI_STATUS_InsufficientResources
    };
```

4.6 Accessing CAPI

This section describes how to access the CAPI interfaces and record profiling information. It contains the following sections:

- *The CAPI derived class*
- *Memory Management* on page 4-26
- *Collecting the profiling information* on page 4-27.

4.6.1 The CAPI derived class

To support profiling, a model must implement a class inheriting from CAPI and specify the information regarding the information to be profiled such as the profiling streams and channels to be collected for each stream.

The derived CAPI class, called *CompName*_CAPI, must implement an empty constructor calling the base class CAPIBase constructor.

```
CompName_CAPI::CompName_CAPI(CASIModule *comp):CAPIBase(comp){};
```

The initialization and memory allocation for the CAPI interface of a model must be completed by the end of the init phase.

The CAPI interface implements the function CAPIGetProfilingStreams() that is used by CAPI clients to dynamically discover the description of the streams associated with the model.

If a CASI model implements a CAPI interface, obtain a pointer to the CAPI interface by calling:

```
CAPI * capi = model->getCAPI()
```

See *Example CAPI implementation* on page 4-28 for examples of constructing and initializing an object derived from the CAPI class.

The CAPI object is guaranteed to be fully constructed after the init phase of the model has completed.

The streams metadata can then be retrieved using :

```
uint32_t actualNrStreams;
CAPIStreamInfo_t streamsInfo [100]; // initialize the array with a maximum value
capi->CAPIGetProfilingStreams (100, & actualNrStreams, streamsInfo);
```

The function CAPIFindStream() returns a pointer to the CAPIStream_t data structure for a given stream name (the stream names must be unique for each component).

```
CAPIStream_t *CAPI::CAPIFindStream(const char *name);
```

The actual streams are obtained based on the metadata using a call to CAPIFindStream():

```
CAPIStream_t * stream0 = capi->CAPIFindStream(streamInfo[0].streamName);
```

During simulation, the actual profiling data can be directly accessed by inspecting the list of collected profiling events exposed by the eventHead and eventTail members of CAPIStreamInfo_t structure. To access the cycle value of the first collected event, use:

```
uint64_t cycle = * (uint64_t *) (stream->eventHead-> eventsTrace + 0);
```

4.6.2 Memory Management

For large simulations, the amount of profiling data collected can grow very large. The memory management for streams is under complete control of the client application. Clients must set the CAPICallback before the creation of any CAPI enabled model by using the function setCAPICallback(). All the streams actual content is recorded in data structures allocated through this callback. It is the responsibility of the memory manager to cleanup the allocated memory after the simulation ends.

The CAPIRecordEvent() function must ensure that there is space in the current eventsTrace array before recording data. If there is not sufficient space, the allocateTraceSegment() function must be called to create a new array. The allocateTraceSegment() function is accessed from the CAPICallback object.

The implementation of the allocateTraceSegment() function must allocate an object of type CAPITraceSegment_t, and the eventsTrace array (the actual event collection area). The size of the event collection area must be:

`stream->eventWidthInBytes * stream->eventsTraceSegmentSize`.

The memory manager can change the stream->eventsTraceSegmentSize to a new value, but must keep the stream->eventWidthInBytes unchanged.

It is illegal to completely remove all the CAPITraceSegment_t structures associated with a stream, at least one segment must be present in the list at all times.

The memory manager callback might be used during model construction time, so it is advisable to setup a custom memory manager before any CAPI enabled model is constructed. By default the CAPI support libraries do install a simple memory manager that only allocates new trace segments.

4.6.3 Collecting the profiling information

The `CAPIStream_t` pointers are recorded inside the component during the init phase and are used in subsequent calls to the `CAPIRecordEvent` function or macros. This reduces function calls to a minimum and reduces overhead.

The pointer to a `CAPIStream_t` is returned by the `CASIFindStream()` function (see *The CAPI derived class* on page 4-25).

For streams containing one or two channels, macros are provided for collecting the profiling information. ARM strongly encourages using the macros whenever possible because it significantly reduces the performance overhead of the profiling.

```
CAPIRecordEvent1(stream, arg1)
CAPIRecordEvent2(stream, arg1, arg2)
```

Note

Each event recorded for a stream must contain a data element for every channel included in that stream.

If `stream1` is composed of `channel_1` and `channel_2`, for example, then every event recorded at runtime must provide a value for both `channel_1` and `channel_2` in that stream.

4.7 Example CAPI implementation

This section describes a CAPI implementation in a typical component. It contains the following sections:

- *The CAPI object*
- *Initializing the channel objects* on page 4-29
- *Using the CAPIGetProfilingStreams() function* on page 4-29
- *Recording events* on page 4-30
- *Reading the recorded profile data* on page 4-31.

4.7.1 The CAPI object

Example 4-16 shows an implementation of the CAPI derived class for a cache component:

Example 4-16 CASICache_CAPI class

```
#include "CAPI.h"
class CASICache_CAPI: public CAPI {
public:
    Cache_CAPI(CASIModule *comp):CAPI(comp){}
    CAPIReturn_t CAPIGetProfilingStreams(uint32_t desiredNrStreams,
    uint32_t *actualNrStreams, CAPIStreamInfo_t *streams);
};
```

The cache collects one stream of information, called `Accesses`, containing two channels: `Address` and `Operation`. The `Address` channel is of type `CAPI_CHANNEL_TYPE_U16`, while the `Operation` channel is of type `CAPI_CHANNEL_TYPE_SYMBOL`.

The `CacheOpType` enumeration defines the different values possible for the symbol type channel, corresponding to cache actions such as hits, misses, or refills:

```
enum CacheOpType{
    COT_READ_MISS,
    COT_READ_HIT,
    COT_WRITE_MISS,
    COT_WRITE_HIT,
    COT_REFILL,
    COT_WRITEBACK
};
```


4.7.2 Initializing the channel objects

Declare and initialize the CAPIChannelSymbolInfo, CAPIChannel and CAPIStreamInfo objects as shown in Example 4-17:

Example 4-17 Declaring the channel objects

```
class Cache_CAPI: public CAPI {
public:
    Cache_CAPI(CASIModule *comp):CAPIBase(comp){}
    CAPIReturn_t CAPIGetProfilingStreams(uint32_t desiredNrStreams,
                 uint32_t *actualNrStreams, CAPIStreamInfo_t *streams);
private:
    CAPIChannelSymbolInfo_t cache_optype_symbols[1]; //One channel is a symbol
    CAPIChannel_t *cache_channels[2]; //two channels
    CAPIStreamInfo_t cache_streams[1]; //one stream
};
```

4.7.3 Using the CAPIGetProfilingStreams() function

The function Cache_CAPI::CAPIGetProfilingStreams initializes the symbolStrings and symbolColors fields for the Operation channel, and returns the streams in the streams argument (see Example 4-18).

Note

Objects and memory structures created with new, such as CAPIChannel_t, must be destroyed or freed in the terminate() stage.

Example 4-18 The CAPIGetProfilingStreams function

```
CAPIReturn_t Cache_CAPI::CAPIGetProfilingStreams(uint32_t desiredNrStreams,
                uint32_t *actualNrStreams,CAPIStreamInfo_t *streams)
//Init the channels
    cache_channels[0]=new CAPIChannel_t();
    cache_channels[0]->channelName="Address";
    cache_channels[0]->description="The address being accessed";
    cache_channels[0]->type=CAPI_CHANNEL_TYPE_U16;
    cache_channels[1]=new CAPIChannel_t();
    cache_channels[1]->channelName="Operation Type";
    cache_channels[1]->description="The cache operation type";
    cache_channels[1]->type=CAPI_CHANNEL_TYPE_SYMBOL;
    cache_channels[1]->count=1;
    cache_channels[1]->symbolInfo=&cache_optype_symbols[0];
```

```
//Init the streams
    cache_streams[0].streamName="Accesses";
    cache_streams[0].streamLabel="Accesses";
    cache_streams[0].description="The cache accesses stream";
    cache_streams[0].nrOfChannels=2;
    cache_streams[0].channels=&cache_channels[0];

//Init the symbols for the CacheOpType channel
    cache_optype_symbols[0].nrSymbolValues = 6;
    cache_optype_symbols[0].symbolValues = new CAPISymbolValue_t[6];
    cache_optype_symbols[0].symbolValues[0].valueName=strdup("Read - Miss");
    cache_optype_symbols[0].symbolValues[1].valueName=strdup("Read - Hit");
    cache_optype_symbols[0].symbolValues[2].valueName=strdup("Write - Miss");
    cache_optype_symbols[0].symbolValues[3].valueName=strdup("Write - Hit");
    cache_optype_symbols[0].symbolValues[4].valueName=strdup("Refill");
    cache_optype_symbols[0].symbolValues[5].valueName=strdup("Writeback");

    cache_optype_symbols[0].symbolValues[0].valueColor=CAPI_COLOR_RED;
    cache_optype_symbols[0].symbolValues[1].valueColor=CAPI_COLOR_GREEN;
    cache_optype_symbols[0].symbolValues[2].valueColor=CAPI_COLOR_RED;
    cache_optype_symbols[0].symbolValues[3].valueColor=CAPI_COLOR_BLUE;
    cache_optype_symbols[0].symbolValues[4].valueColor=CAPI_COLOR_YELLOW;
    cache_optype_symbols[0].symbolValues[5].valueColor=CAPI_COLOR_PURPLE;

//set the return variables
    streams[0] = cache_streams[0];
    *actualNrStreams = 1;
    return CAPI_STATUS_OK;
}
```

The Cache init phase initializes the CAPI interface for the model:

```
capi = new Cache_CAPI(this);
CAPI:CAPIRegisterInterface(capi);
cacheStream=capi->CAPIFindStream("Accesses");
```

The CAPI initialization code might be located, for example, in the initialization code for the CASI object for the component.

4.7.4 Recording events

The `cacheStream` stream pointer is used to collect cache information (hit, miss, refill, and so forth) during the cache operation:

```
if(hit) CAPIRecordEvent2(cacheStream, (uint16_t) addr, (uint8_t) COT_READ_HIT);
```


Note

`CAPIRecordEvent2()` is a macro that records an event with two data items.

Maintaining consistency between the selection of colors and the order of streams ensures that the Profile windows are easy to analyze.

In general, colors with a red hue are used to indicate something undesirable (for example a cache miss) and green colors indicate success (for example a cache hit).

After collecting the profiling information during simulation, the user can view the profiling by either:

- Using a third-party application to select the stream to view and the channels to represent on the X and Y axes of a profiling graph that is displayed in a Profile Window.
- writing a custom function that replays the profiling data (see *Reading the recorded profile data*).

4.7.5 Reading the recorded profile data

The simulation environment might include tools that replay the collected profile information in profile windows or output it to a file. It is also possible to create a custom routine that accesses the collected profile data directly from the main C++ program.

The functions that record profile data must be placed in the individual components (see *Collecting the profiling information* on page 4-27). The top-level program must include the following code to access the recorded profile stream:

CAPICallback

The top level program must implement a callback function to access the data recorded by the component (see Example 4-19 on page 4-32). The `CAPICallback` function manages the allocation of memory for the data collection functions of the profile interface.

CAPI display function

The top level program must implement a function to access the CAPI data stream for the component (see Example 4-20 on page 4-32).

Calling the display function

The top level program must initialize the CAPI interface and call the display function to output the recorded data (see Example 4-21 on page 4-33).

Example 4-19 Implementing CAPICallback

```
class TopCAPICallback : public CAPICallback
{
public:
    TopCAPICallback (CASIClockDriverRoot * _clockRoot)
        : clockRoot (_clockRoot) {}

    virtual ~TopCAPICallback () {}

    virtual CAPITraceSegment_t * allocateTraceSegment (CAPIStream_t * stream)
    {
        // minimal functionality
        CAPITraceSegment_t * result = new CAPITraceSegment_t ();
        result->eventsTrace = new uint8_t [(int) (stream->eventWidthInBytes *
                              stream->eventsTraceSegmentSize)];
        return result;
    }

    virtual uint64_t getCurrentCycle ()
    {
        return clockRoot->getCurrentCycle ();
    }

private:
    CASIClockDriverRoot * clockRoot;
};
```

Example 4-20 Function for reading profile information

```
void printCAPIEvents (eslapi::CAPIStream_t * stream)
{
    for (int i = 0; i < 10; ++ i)
    {
        printf (" Event %d ", i);
        uint64_t base = i * stream->eventWidthInBytes;
        uint64_t offset = 0;
        for (int j=0; j < (int) stream->info->nrOfChannels; ++ j)
        {
            switch (stream->info->channels[j]->type)
            {
                case eslapi::CAPI_CHANNEL_TYPE_U8:
                case eslapi::CAPI_CHANNEL_TYPE_BOOL:
                case eslapi::CAPI_CHANNEL_TYPE_SYMBOL:
                    printf ("0x%x ", (int) * ((uint8_t*) &
                        (stream->eventHead->eventsTrace[base + offset])));
```



```
                    offset += 1;
                    break;
                case eslapi::CAPI_CHANNEL_TYPE_U16:
                    printf ("0x%x ", (int) * ((uint16_t*) &
                        (stream->eventHead->eventsTrace[base + offset])));
                    offset += 2;
                    break;
                case eslapi::CAPI_CHANNEL_TYPE_U32:
                    printf ("0x%x ", (int) * ((uint32_t*) &
                        (stream->eventHead->eventsTrace[base + offset])));
                    offset += 4;
                    break;
                case eslapi::CAPI_CHANNEL_TYPE_U64:
                    printf ("0x%x ", (int) * ((uint64_t*) &
                        (stream->eventHead->eventsTrace[base + offset])));
                    offset += 8;
                    break;
            }
        }
        printf ("\n");
    }
}
```

Example 4-21 Calling printCAPIEvent() from main()

```
int sc_main (int argc, char *argv[])
{
    eslapi::CASIClockDriverRoot fast ("fast_clock", 1, SC_NS);
    eslapi::CASIClockDriver slow;
    fast.registerClockSlave (& slow, eslapi::CASI_PHASE_BOTH, 0, 4);
    // capi init
    TopCAPICallback eslapi::capiCallback ( & fast);
    CAPI::setCAPICallback (& eslapi::capiCallback);

    .
    .
    .

    // use CAPI
    printf ("%s events\n", s1->getInstanceID ().c_str ());
    printCAPIEvents (s1->getCAPI ()->CAPIFindStream ("Events"));
    printf ("%s events\n", s2->getInstanceID ().c_str ());
    printCAPIEvents (s2->getCAPI ()->CAPIFindStream ("Events"));
}
```


Chapter 5
The CASI Memory Map Interface

This chapter describes the *Cycle Accurate Simulation Interface - Memory Map Interface* (CASIMMI) used to modify the memory map of components connected to a bus master. It contains the following sections:

5.1 CASIMMI interfaces

This section describes the CASIMMI classes and interfaces.

The CASI Memory Map Interface simplifies configuring the address spaces managed by a bus component:

1. A memory map describes how the address regions are assigned to the slaves present in the system.
2. The bus routes a given bus transaction based on the transaction address.
3. The address region containing the transaction address is used to determine the target slave.

Complex systems might define multiple memory maps for a single bus and change the active memory map from one cycle to the next. For example, the memory map active during reset might contain ROM modules that are replaced with RAM modules during normal execution. If the active memory map is changed, the bus must notify its slaves with the new address regions they are now assigned to.

The `CASIMMI.h` file (in the CASI `include` directory) contains the class and structure definitions. See the header file for more information on data structures used by this class and any design changes.

The memory maps are managed by the following steps:

1. The component defines the structure of its memory maps by interrogating the connected slaves.
2. The environment sets the values for each memory map present and establishes the ranges of the corresponding address spaces.

5.1.1 The CASIMMIMemoryMap structure

The `CASIMMIMemoryMap` and `CASIMMIMemoryMaps` structures define how the memory map details are stored:

- `CASIMMIMemoryMap` describes a single address space (see Example 5-1 on page 5-3).
- `CASIMMIMemoryMaps` Describes the address spaces associated with a bus. Only one address space can be active at any given time. During simulation, the bus address space can be reconfigured by selecting a different memory map.

Example 5-1 CASIMMIMemoryMap structures

```
struct CASIMMIMemoryMap{
    std::string memoryMapName;      // The identifier of this address map
    uint32_t num_address_regions;    // Total number of address regions
    // The regions in this memory map
    // (all following are arrays that contain one entry for every region)
    uint64_t* start;                // the start of the regions for this map
    uint64_t* size;                 // the size of the regions for this map
    string* name;                  // the names of the regions for this map
    string* slaveCompInstanceID;   // the Instance ID of the slave that each region correspond to
    string* slavePortInstanceName; //the Instance Name of the Slave Port that each region correspond to
    CASIMMIMemoryMap& operator= (CASIMMIMemoryMap& puRef); //Make a deep copy of this structure
}

struct CASIMMIMemoryMaps{
    CASIMMIMemoryMap* maps;      // An array with the address maps for this bus
    uint32_t numMaps;              // num of addr maps supported
    CASIMMIMemoryMaps& operator= ( CASIMMIMemoryMaps& puRef ); // deep copy
}
```

5.1.2 The CASIMMIMemoryMapRequest structure

The `CASIMMIMemoryMapRequest` structure is used by the environment to identify the memory maps structure.

Example 5-2 Request memory map details

```
struct CASIMMIMemoryMapRequest{
    uint32_t numMaps;             // num of addr maps supported
    uint32_t* memoryMapID;        // array with IDs of addr maps supported.
    std::string* mapNames;      // array with names of addr maps supported (numAddrMaps number)
    CASIMMIMemoryMapRequest& operator= ( CASIMMIMemoryMapRequest& puRef ); // deep copy
}
```

5.1.3 CASIMMI class definition

The CASSIMMI class describes the memory maps of a component. The memory maps are describe in a two-step process:

1. The component defines the structure of its memory maps by interrogating the connected slaves.
2. The environment sets the values for each memory map present and therefore establishes the ranges of the corresponding address spaces.

Example 5-3 CASIMMI class definition

```
class CASIMMI : public CAInterface
{
public:
   CASIMMI();
   virtual ~CASIMMI();

    // Return the CAInterface name for this interface.
    static if_name_t IFNAME() { return "eslapi.CASIMMI2"; }
    // Specify the current minor revision for this interface.
    static if_rev_t IFREVISION() { return 0; }

    // register interface
    static CASIStatus registerInterface(CASIMMI* puMMI, CASIModuleIF* component,
           std::string name, uint32_t numMasterPorts, CASIPortIF **masterPortList);
    // Retrieve the MMI interface which was previously registered with the environment.
    static CASIMMI *getInterface(CASIModuleIF* component);

    // MMI structure
    // Request the bus maps (number and their names)
    virtual CASIStatus requestMemoryMaps(CASIMMIMemoryMapRequest* req) = 0;
    virtual void setEnableMME(bool value);
    virtual bool  isMMEEnabled();

    // Reserved for future extensions.
    virtual CASIMMIDetails *getMMIDetails() {return ptrMMIDetails;}

    // Set the maps that are present in this bus for this system.
    virtual CASIStatus setMemoryMaps(CASIMMIMemoryMaps* maps);
    // Get the maps that are present in this bus for this system
    virtual CASIMMIMemoryMaps *getMemoryMaps();

    // Dynamic MMI, can be invoked during simulation
    // Set the CURRENT memory maps.
    virtual CASIStatus setCurrentMemoryMaps(uint32_t* ids, int numMaps);
    // Get the CURRENT memory maps.
```



```
    virtual uint32_t *getCurrentMemoryMaps(int &numMaps);

    // Return interface if requested
    virtual CAInterface *ObtainInterface(if_name_t ifName, if_rev_t minRev, if_rev_t* actualRev)
    {
       if((strcmp(ifName,"eslapi.CASIMMI2") == 0) && (minRev <= 0)){
            *actualRev = 0;
            return this;
        }
        return NULL;
    }

protected:
    // The maps that are present in this bus for this system (the actual regions for every map in
    // the bus). Value set by the setMemoryMaps() and returned by getMemoryMaps()
    CASIMMIMemoryMaps* MemoryMapList;

    // Map ID of the current Memory Maps
    // Value set by setCurrentMemoryMaps and returned by getCurrentMemoryMaps()
    uint32_t* currentMemoryMapIDs;
    int numCurrentMemoryMaps;

    CASIMMIDetails* ptrMMIDetails;
    // Bus Master Port should use the Address Regions set by the Memory Map Editor
    bool bMMEEnabled;

private:
    typedef std::map <CASIModuleIF  *, CASIMMI *> ComponentMMIMap;
    static ComponentMMIMap componentMMIMap;
}
```

5.1.4 CASIMMI::registerInterface()

This function registers the MMI interface

```
static CASIStatus registerInterface (CASIMMI* puMMI,CASIModuleIF* component,
       std::string name,uint32_t numMasterPorts,  CASIPortIF **masterPortList)
```

where:

`puMMI` is a pointer to a CASIMMI object.

`component` is the component that owns the ports.

`name` is the name of the interface.

`numMasterPorts`
is the number of master ports in the component.

`masterPortList`

is a pointer to an array of master ports.

5.1.5 CASIMMI::getInterface()

Retrieves the MMI interface that was previously registered with the environment.

```
static CASIMMI *getInterface (CASIModuleIF* component)
```

where:

`component` is the component that implements the MMI interface.

5.1.6 CASIMMI::requestMemoryMaps()

Requests the bus maps (both number and names).

```
virtual CASIStatus requestMemoryMaps(CASIMMIMemoryMapRequest *req) = 0
```

where:

`reg` is a pointer to a request structure.

5.1.7 CASIMMI::setEnableMME()

Enable the memory map editor.

```
virtual void setEnableMME(bool value)
```

5.1.8 CASIMMI::isMMEEnabled()

Determine if the memory map editor is enabled.

```
virtual bool isMMEEnabled()
```

5.1.9 CASIMMI::getMMIDetails()

Reserved for future use.

```
virtual CASIMMIDetails *getMMIDetails() {return ptrMMIDetails;}
```

5.1.10 CASIMMI::setMemoryMaps()

Set the maps that are present in this bus for this system, that is, set the actual regions for every map in the bus.

```
virtual CASIStatus setMemoryMaps(CASIMMIMemoryMaps* maps);
```


Note

This enables the environment to set the memory maps.

5.1.11 CASIMMI::getMemoryMaps()

Get the maps that are present in this bus for this system, as set by the previous `setMemoryMaps()` call.

```
virtual CASIMMIMemoryMaps *getMemoryMaps()
```

5.1.12 CASIMMI::setCurrentMemoryMaps()

A component might have a number of alternate memory maps. This call enables selecting a certain memory map for simulation. The call can be invoked dynamically during simulation.

```
virtual CASIStatus setCurrentMemoryMaps(uint32_t* ids, int numMaps)
```

where:

`ids` is the array of memory map ids.

`numMaps` is the number of maps to set.

5.1.13 CASIMMI::getCurrentMemoryMaps()

Get the current memory maps.

```
virtual uint32_t *getCurrentMemoryMaps(int &numMaps)
```

5.2 Sample implementation

The examples in this section describe adding the CASIMMI code to a component.

For more details on the CASIMMI class, see *The CASIMMIMemoryMap structure* on page 5-2 and the `CASIMMI.h` and `CASITransaction.h` files for information on the memory and transaction functions and data structures.

The *`CompBus`*`_CASIMMI` class is the component's implementation of the master CASIMMI class (*`CompBus`* is the name of the component that behaves like a bus. The name of the classes will of course be different for your own component.) For more detail, see:

- *CompBus_CASIMMI class Implementation details* on page 5-9
- *CompBus_CASIMMI.h file* on page 5-10
- *CompBus_CASIMMI::CASIMMI() constructor* on page 5-11
- *CompBus_CASIMMI::~CASIMMI destructor* on page 5-11
- *CompBus_CASIMMI::requestMemoryMaps* on page 5-11
- *CompBus_CASIMMI::getCurrentMemoryMaps()* on page 5-12.

The `slave_port_1_TS` class is a transaction slave port that is part of the implementation of a custom component. (`slave_port_1_TS` is the name of the port. The name of the classes will of course be different for your own port.) This class describes the Memory Map Interface classes and methods that are present in a component that provides a bus slave port. ARM recommends that new designs that implement buses use the CASIMMI memory map interface. For more details see:

- *slave_port_1_TS::slave_port_1()* on page 5-13
- *slave_port_1_TS::setAddressRegions()* on page 5-13
- *slave_port_1_TS::getMappingConstraints()* on page 5-14.

New designs that implement buses must use the new CASIMMI memory map interface.

5.2.1 CompBus::CompBus()

The class implementation for an example bus component initializes the CASIMMI interface.

An example of a constructor for the component is shown in Example 5-4.

Example 5-4 Constructor

```
CompBus::CompBus(CASIModuleIF* c, const std::string &s) : CASIModule(c, s)
{
    bm1_BMaster = new sc_port< CASITransactionIF, 0 >(this, "bm1", 0x1000, 0x10000 );
    initTransactionPort((CASIPortIF*)bm1_BMaster);
    registerPort( bm1_BMaster, "bm1" );
}
```

5.2.2 CompBus_CASIMMI class Implementation details

For master ports that do not have multiple memory maps, it is simpler to use the built-in `sc_port<casi_transaction_if,0>` class that contains a standard bus master. Instantiate this class to create the master port:

```
sc_port::sc_port<casi_transaction_if,0> (CASIModuleIF* _owner,
    const std::string& name,  uint64_t? blocksize, uint64_t? memsize,
    maxsim::CASITransactionProperties* prop)
```

For the examples in this section, *CompBus* is the name of the component that implements the CASIMMI interface. The name of the classes will of course be different for your own component. Create your own component class that inherits from CASIMMI if your component uses multiple memory maps.

If you use the Bus Master Port constructor below, the CASIMMI will not be available:

```
sc_port<CASITransactionIF,0>(const std::string& name, uint64_t blocksize,
    uint64_t memsize,CASITransactionProperties* prop)
```

If you are creating a bus master port that requires more than one memory map and require that the port supports the Memory Map Editor, you must create an object of a class derived from CASIMMI and use `registerInterface()` to associate the bus master port with that CASIMMI object.

Bus master ports in components that use the standard bus master port (`sc_port<CASITransactionIF, 0>`) have their own predefined implementation of the CASIMMI constructor, destructor, and `requestMemoryMaps()` functions. These are not user modifiable.

Older buses continue to work without modification, but the Memory Map Editor is not be available and memory regions must be edited from the Parameters window for each slave component.

5.2.3 CompBus_CASIMMI.h file

The *CompBus*_CASIMMI.h file contains the class definition that provides the interface to the memory map functionality. An example of a .h file is shown in Example 5-5.

Example 5-5 CompBus_CASIMMI.h file

```
class CompBus;
class CompBus_CASIMMI : public CASIMMI
{
public:

    CompBus_CASIMMI(CompBus* c, std::string& MemSpaceName, uint32_t numPort,
                    CASIPortIF **masterPortList);
    virtual ~CompBus_CASIMMI();

public:

    // Implemented by the BUS
    // Request the bus maps (number and their names) to create the different tabs in the MME).
    // These are the maps supported by this bus
    CASIStatus requestMemoryMaps(CASIMMIMemoryMapRequest* req);

private:

    CompBus*    target;
    casi_port_base** target_portlist;
    CASIMMIMemoryMapRequest* MemMapReq;

};
#endif
```

5.2.4 CompBus_CASIMMI::CASIMMI() constructor

An example of a *CompBus*_CASIMMI constructor is shown in Example 5-6.

Example 5-6 Constructor

```
CompBus_CASIMMI::CompBus_CASIMMI(CompBus* c, std::string& MemSpaceName, uint32_t numPort,
       CASIPortIF **masterPortList) : target(c), target_portlist(masterPortList){
    CASIModuleIF* puModule = dynamic_cast<CASIModuleIF *> (c);
    MemMapReq =  new CASIMMIMemoryMapRequest;
    MemMapReq->numMaps = 2;
    MemMapReq->memoryMapID = 0x0;
    MemMapReq->mapNames = new std::string[MemMapReq->numMaps];
    MemMapReq->mapNames[0] = std::string("MapX");
    MemMapReq->mapNames[1] = std::string("MapY");
    registerInterface(this, puModule, MemSpaceName, 1, masterPortList);
}
```

5.2.5 CompBus_CASIMMI::~CASIMMI destructor

An example of a *CompBus*_CASIMMI destructor is shown in Example 5-7.

Example 5-7 CASIMMI destructor

```
CompBus_CASIMMI::~CompBus_CASIMMI()
{
    delete[] MemMapReq->mapNames;
    delete MemMapReq;
}
```

5.2.6 CompBus_CASIMMI::requestMemoryMaps

An example of the `requestMemoryMaps()` function is shown in Example 5-8 on page 5-12.

Note

For master ports that do not have multiple memory maps, it is simpler to use the built-in `sc_port` class that contains a standard bus master. Ports generated by the component master support only one memory map.

Example 5-8 requestMemoryMaps()

```
//Request the bus maps (number and their names) to create the different tabs in the MME).
//These are the maps supported by this bus

CASIStatus CompBus_CASIMMI::requestMemoryMaps(CASIMMIMemoryMapRequest* req)
{
    req->mapNames = new std::string[3];
    req->mapNames[0] = “Map1”;
    req->mapNames[1] = “Map2”;
    req->mapNames[2] = “Map3”;
    return CASI_STATUS_OK;
}
```

5.2.7 CompBus_CASIMMI::getCurrentMemoryMaps()

Some components can support multiple active memory map at the same time. This requires that an array pointer and the number of maps is passed as parameters when setting the current maps. An example of a `getCurrentMemoryMaps()` from the CASIMMI class is shown in Example 5-7 on page 5-11.

Note

Implementation of this function is not required. A default implementation is provided in the `CASIMMI` base class.

The function is provided to enable the simulation environment to perform dynamic remapping.

Example 5-9 getCurrentMemoryMaps

```
uint32_t *CASIMMI::getCurrentMemoryMaps(int &numMaps) {
    numMaps = 1;
    uint32_t* currentMemoryMapIDs = new uint32_t[1];
    currentMemoryMapIDs[1] = mapid;
    return currentMemoryMapIDs;
}
```


5.2.8 slave_port_1_TS::slave_port_1()

The slave port property value for `casi_version` must be `CASI_VERSION_1_1` or higher. For example:

Example 5-10 Setting the CASI version

```
slave_port_1_TS::slave_port_1_TS(CompSlave1* _owner) :
        CASITransactionSlave( _owner, "slave_port_1_TS"), owner(_owner)
{
    CASITransactionProperties prop;
    memset(&prop,0,sizeof(prop));
    prop.casiVersion = CASI_VERSION_1_1; // must be version 6 or higher
}
```

5.2.9 slave_port_1_TS::setAddressRegions()

Example 5-11 shows generated code that has been updated.

Example 5-11 setAddressRegions

```
/* Method to set the address regions */
//Sets the value of the address regions for this slave port
void slave_port_1_TS::setAddressRegions(uint64_t* start, uint64_t* size, std::string* name)
{
    if (start && size && name)
    {
        string port_name = this->getPortInstanceName();
        owner->message(CASI_MSG_INFO, "Set Address Region Called for Port %s", port_name.c_str());
        int i = 0;
        while (size[i] != 0) // empty string indicates end of array
        {
            owner->message(CASI_MSG_INFO, "Address Region: start= 0x%I64x size = 0x%I64x Name= %s",
                        start[i], size[i], name[i].c_str());
            i++;
        }
    }

    // TODO: Add your code here.
}
```

5.2.10 slave_port_1_TS::getMappingConstraints()

Example 5-12 shows an example of updating the `getMappingConstraints()` code for a transaction slave. Example 5-13 on page 5-15 shows the definitions of data structures.

Example 5-12 getMappingConstraints

```
CASIMemoryMapConstraints* slave_port_1_TS::getMappingConstraints()
{
    puMemoryMapConstraints.minRegionSize = 0x1000;
    puMemoryMapConstraints.maxRegionSize = 0xffffffffffffffff;
    puMemoryMapConstraints.minAddress = 0x0;
    puMemoryMapConstraints.maxAddress = 0xffffffffffffffff;
    puMemoryMapConstraints.minNumSupportedRegions = 0x2;
    puMemoryMapConstraints.maxNumSupportedRegions = 0xa;
    puMemoryMapConstraints.alignmentBlockSize = 0x80 ;
    puMemoryMapConstraints.numFixedRegions = 0; // reserved for future use
    puMemoryMapConstraints.fixedRegionList = NULL; // reserved for future use

    return &puMemoryMapConstraints;
}
```

Note

The `minNumSupportedRegions` and `maxNumSupportedRegions` values determine how many regions can be present. These constraints must be used by an external editor that deletes or adds regions.

Example 5-13 Data structures

```
struct CASIMemRegion
{
    uint64_t start;
    uint64_t size;
    std::string name;
};

struct CASIMemoryMapConstraints
{
    uint64_t minRegionSize;
    uint64_t maxRegionSize;
    uint64_t minAddress;
    uint64_t maxAddress;
    //min number of supported regions by this slave at any given point in time
    uint32_t minNumSupportedRegions;
    //max number of supported regions by this slave at any given point in time
    uint32_t maxNumSupportedRegions;
    //Allignment requirement, the min block size where this slave's regions can be mapped to
    // (4k for AHB)
    uint64_t alignmentBlockSize;
    CASIMemoryMapConstraintsDetails *details; //Reserved
    uint32_t numFixedRegions;
    CASIMemRegion* fixedRegionList;        // only if numFixedRegions > 0
};
```


Appendix A
Static Scheduling of Communication Functions

This appendix describes the static scheduling mechanism that is used to enable combinatorial support for communication functions. It contains the following sections:

- *Introduction to combinatorial path scheduling* on page A-2
- *Specifying the combinatorial path* on page A-5
- *Error checking* on page A-6
- *Example implementation* on page A-7.

Note

Non-combinatorial path scheduling is typically used for components. See *Cycle based scheduling* on page 1-17 and *The clock interface classes* on page 2-41 for more information on the standard non-combinatorial communication process.

A.1 Introduction to combinatorial path scheduling

The simple cycle-based model described in *Cycle based scheduling* on page 1-17 is adequate for simple models. Some complex models, however, require multiple combinatorial paths to be handled in the same simulation cycle. A set of combinatorial paths means that there are multiple functions in a component that must be called in a specific order during the communication phase.

Creating a specific order for the functions in the combinatorial paths enables:

- protocols that require cyclic paths between components to manage arbitration
- communication between transactor and RTL components
- custom components that contain combinatorial paths between subcomponents
- optimizing simulation speed by enabling or disabling clock registration.

Figure A-1 shows a typical communicate and update sequence. The `communicate()` and `update()` functions for each component are not called in a predictable order. From the perspective of the component developers, the order must be considered as non-deterministic.

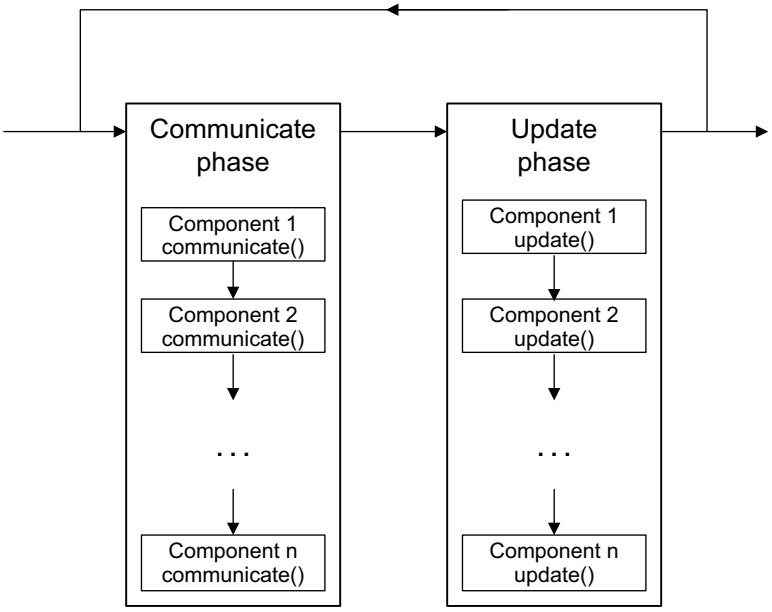

Figure A-1 Standard communicate and update phases

Figure A-2 shows a system where there are multiple communications functions for Component6, Component7, and Component8 that must be called in a specific order.

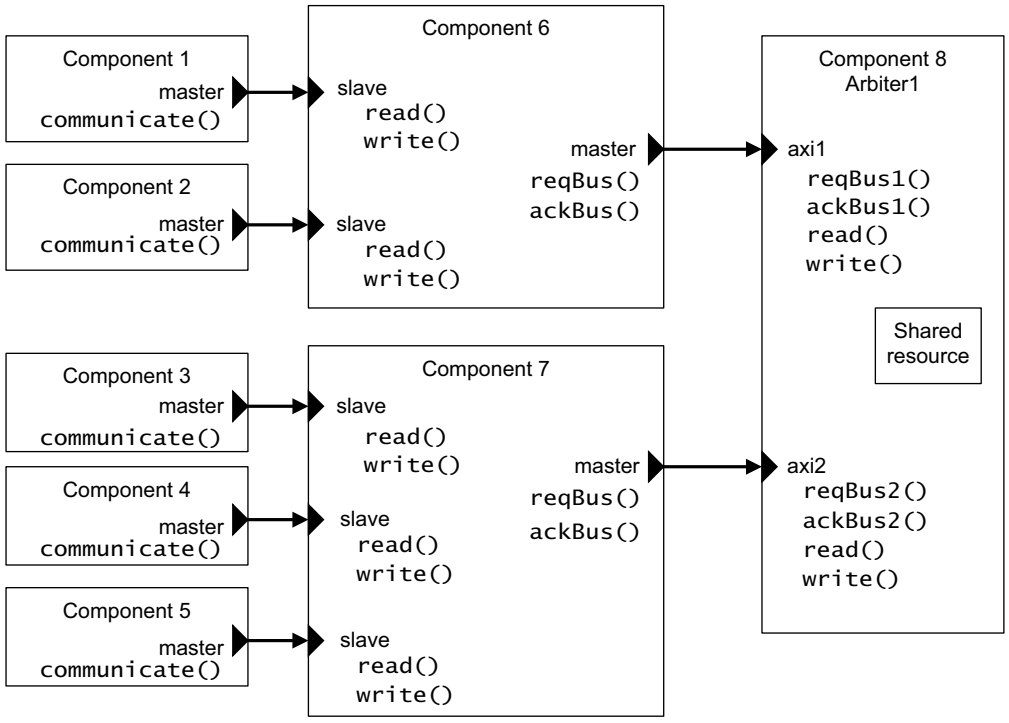

Figure A-2 System requiring ordered component communication

In this system, the remaining components use standard `communicate()` functions and the order they are called is not specified by the designer. The `read()` and `write()` functions in component 8 are not called by a `communicate()` function in the master of components 6 and 7. The order of function calls for these functions is not specified in the figure. They might, for example, be called by the `ackBus()` functions.

Figure A-3 shows the calling order for the communication functions.

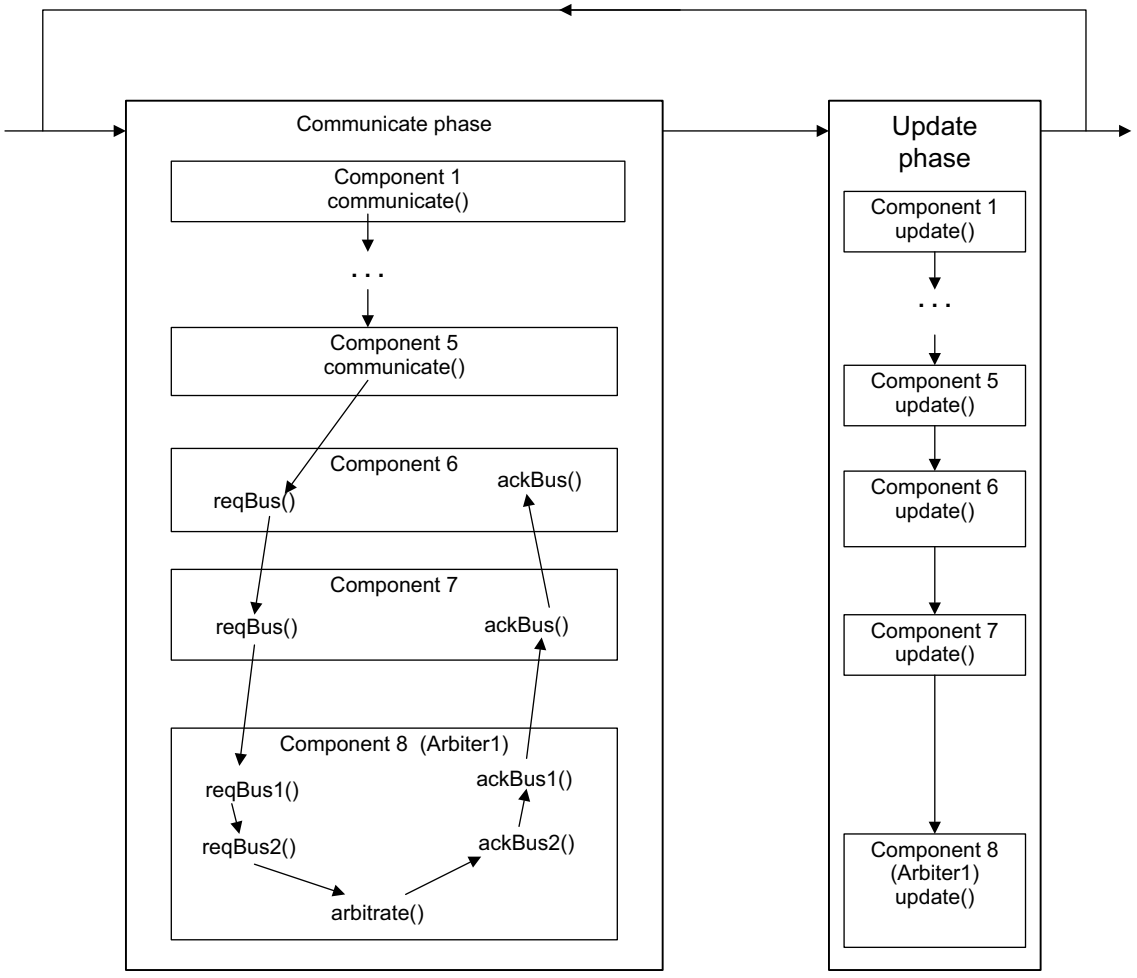

Figure A-3 Statically scheduled communicate and update phases

Note

There are no dependencies between Component6 and Component7, so the scheduling order for these components is determined randomly by the scheduling software and not by the system designer.

Statically scheduling the communication functions does not affect the scheduling order for the update functions.

A.2 Specifying the combinatorial path

The steps required to implement combinatorial paths with a static schedule are:

1. Identify the combinatorial paths that exist for your components and the specified protocol.
2. Identify all of the `communicate()` functions that are required to implement the combinatorial paths. (The individual `communicate()` functions are referred to as nodes in the following text.)
3. Identify the order of combinatorial nodes inside each component.
4. Use explicit `addDependency()` calls for nodes inside the component to specify the order between nodes.
5. Identify the required order for communication between component ports and use the `addDependency()` calls to specify the order for combinatorial paths across components.

   Note

   Any non-combinatorial reads or writes that go across a cycle, that is, are latched and updated, must not be specified as a dependency.

   Writes are always combinatorial because they occur in the same cycle.

   Reads, however, might or might not be latched an must be considered non-deterministic. The execution order cannot be relied upon.

6. Ensure that each port that has a dependency has a matching dependency in the corresponding port in the other component. Connected ports with dependencies must use the same protocol.
7. Do not use calls to `registerClock()` and `unregisterClock()` to provide speed optimizations. If clocking is turned on and off, use the `enableClock()` and `disableClock()` functions instead.

The function prototypes for the static scheduling are described in the following sections:

- *CASIClockMaster::disableClockSlave()* on page 2-46
- *CASIClockMaster::enableClockSlave()* on page 2-46
- *CASIClockMaster::registerRealCommunicate()* on page 2-46
- *CASIClockMaster::registerRealUpdate()* on page 2-47
- *CASIClockMaster::addDependency()* on page 2-48.

A.3 Error checking

The following actions result in an error when the system is compiled or simulated:

- There is a combinatorial read dependency and missing dependencies on other side.
- There is a dependency graph is present on both sides of a port, but does not match one-to-one on two sides of a port.
- The `registerClock()` or `unregisterClock()` functions are used dynamically for a clock port that has dependencies.
- There are cycles in dependency graph (the function at the start of the graph is dependent on a function at the end of the graph).
- The clock domain is different for components contributing to the same directed acyclic graph (DAG).
- Both the old `registerClockSlave()` and new `registerCommunicate()` and `registerUpdate()` mechanisms are used for the same component.
- Two functions with the same name have been registered from a component.
- The function names are not registered for a component and `addDependency()` is called for the component.
- There is an attempt to register more than one update. (This might result in a warning instead of an error.)

A.4 Example implementation

This section provides an example implementation of a arbitration component and two bus ports. Figure A-4 shows the system interconnections.

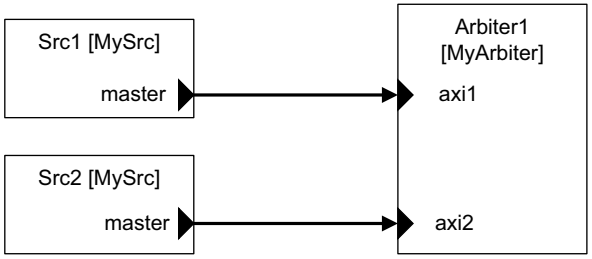

Figure A-4 Component connections

Figure A-5 shows the order of the dependency paths between the combinatorial communication functions for the components.

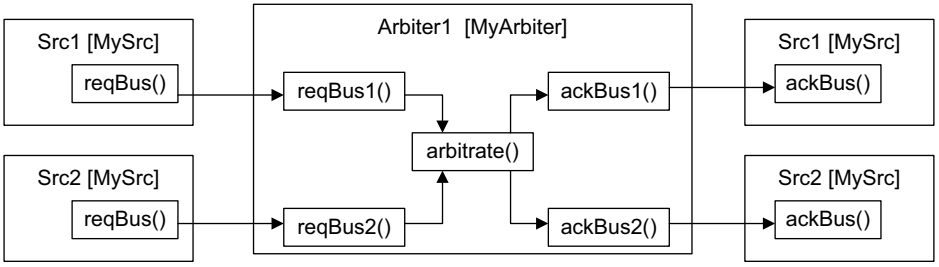

Figure A-5 Component function dependency graph

Example A-1 on page A-8 is a code fragment for the MyArbiter component

Example A-1 Arbiter component

```
// component class:
class MyArbiter: public casi_module
{
    MyArbiter (casi_m_base* c, const string &s);
    void reqBus1() {...}
    void reqBus2() {...}
    void ackBus1() {...}
    void ackBus2() {...}
    void arbitrate() {...}
    // normal communicate can also be registered
    // (not necessary, but works the same way as before)
    void communicate(){...}
    void update(){...}
    ...
}

MyArbiter::MyArbiter (casi_m_base* c, const string &s) : casi_module(c, s)
{
    ...
    registerPort( Port1, "axi1");
    registerPort( Port2, "axi2");
    ...
}

MyArbiter::interconnect(void){
    // add combinatorial logic
    clockMaster->registerCommunicate(this, &MyArbiter::reqBus1,"reqBusFrom1");
    clockMaster->registerCommunicate(this, &MyArbiter::reqBus2,"reqBusFrom2");
    clockMaster->registerCommunicate(this, &MyArbiter::ackBus1,"ackBusTo1");
    clockMaster->registerCommunicate(this, &MyArbiter::ackBus2,"ackBusTo2");
    clockMaster->registerCommunicate(this, &MyArbiter::arbitrate,"arbitrate");

    // add dependency edges
    // IMPOTANT: order of dependencies for a port matters! They must match
    // the dependencies one-to-one on the other component connected
    // to this port.
    clockMaster->addDependency (this, Port1,"reqBusFrom1");
    clockMaster->addDependency (this, "actBusTo1",Port1);
    clockMaster->addDependency (this, Port2,"reqBusFrom2");
    clockMaster->addDependency (this, "actBusTo2",Port2);

    clockMaster->addDependency (this, "reqBusFrom1","arbitrate");
    clockMaster->addDependency (this, "reqBusFrom2","arbitrate");
    clockMaster->addDependency (this, "arbitrate","ackBusTo1");
    clockMaster->addDependency (this, "arbitrate","ackBusTo2");
}
```

Example A-2 is a code fragment for the component with the bus master port.

Example A-2 Source component with bus port

```
// source component
class MySrc: public casi_module
{
    MySrc (casi_m_base* c, const string &s);

    void reqBus() {...}
    void ackBus() {...}

    //normal communicate can still be present, but not necessary
    void communicate(){...}
    void update(){...}

    ...

}

MySrc:: MySrc(casi_m_base* c, const string &s) : casi_module(c, s)
{
    ...
    registerPort( Port, "master");
    ...

}

MySrc::interconnect(void){
    // add combinatorial logic
    clockMaster->registerCommunicate(this, &MySrc::reqBus,"reqBus");
    clockMaster->registerCommunicate(this, &MySrc::ackBus,"ackBus");

    // add dependency edges
    // IMPORTANT: the order of these dependencies matters!
    clockMaster->addDependency (this, "reqBus", Port);
    clockMaster->addDependency (this, Port,"ackBus");
}
```

Appendix B
AMBA™ AHB TLM Specification for CASI

This appendix describes the *AMBA High-performance Bus* (AHB) *Transaction Level Model* (TLM) specification for the ESL APIs. It contains the following sections:

- *Introduction* on page B-2
- *AHB control signals* on page B-3
- *Implementation details for AHB interfaces* on page B-6.

Note

See the *AMBA Specification Rev 2.0* (ARM IHI0011) for detailed specifications of the AMBA AHB bus. AMBA is the *Advanced Microcontroller Bus Architecture* specification.

B.1 Introduction

A transaction is the exchange of a set of information between a master and a slave interface.

If there are multiple AHB masters, a bus arbiter must determine which master has control of the bus by transferring control information between the masters, slaves, and bus arbiter.

The data in the AHB slave can be accessed by synchronous or asynchronous functions:

Synchronous access functions

The `read()` and `write()` functions enable synchronous access between different components. These functions are expected to return immediately (in the same cycle where they were initiated) and return the status of the transaction. Because the functions are synchronous, the master must first obtain control of the slave by successfully calling the `requestAccess()` and `checkForGrant()` functions.

The read/write functions can implement also multi-cycled transactions. If, for example, in the first cycle they return `CASI_STATUS_WAIT`, then the initiating component will call the read/write function again in subsequent cycles, until it receives the `CASI_STATUS_OK` representing the end of this transaction.

The `readDbg()` and `writeDbg()` provide debug accesses and enable debuggers to, for example, read the desired information without advancing the simulation.

Asynchronous access functions

The asynchronous `readReq()` and `writeReq()` functions enable a communication model where the initiator master component provides a callback function pointer to the slave component.

When the slave component is ready to serve the transaction, it calls the callback function notifying the master that the data is ready.

B.2 AHB control signals

The control information and slave response information are encapsulated in the following data structures:

- AHB signal structure
- The CASIMMI structures (such as `CASIMemoryMapConstraints`) for the master and connected slaves.
- the `AHBTransactionProperties` structure (see the `CASITypes.h` and `AHB_Transaction_CASI.h` files)
- the `ctrl` array passed as a parameter with the access functions.

The AHB signal structure (declared in the `AHB_Transaction.h` header file) makes the AHB signals available to all bus participants independently of their involvement in the currently active AHB transfer:

```
typedef struct TAHBSignals
{
    uint32_t hclk; // not modeled, always 1
    uint32_t hreset; // not modeled, always 1
    uint32_t haddr; // can safely be read in update()
    uint32_t htrans; // can safely be read in update()
    uint32_t hwrite; // can safely be read in update()
    uint32_t hsize; // can safely be read in update()
    uint32_t hburst; // can safely be read in update()
    uint32_t hprot; // can safely be read in update()
    uint32_t hwdata[4]; // can safely be read in update()
    uint32_t hrdata[4]; // can safely be read in update()
    uint32_t hsel; // not modeled, always 0
    uint32_t hready; // can safely be read in update()
    uint32_t hresp; // not modeled, always 0
    uint32_t hbusreq[AHB_MAX_MASTERS]; // can safely be read in update()
    uint32_t hlock[AHB_MAX_MASTERS]; // can safely be read in update()
    uint32_t hgrant; // can safely be read in communicate()
    uint32_t hmaster; // can safely be read in communicate()
    uint32_t hmastlock; // can safely be read in update()
    uint16_t hsplit[AHB_MAX_MASTERS]; // not modeled, always 0
    uint32_t hready_z1; // hready delayed by one, can safely be read
            // in communicate(), maintained by AHB_casi
    uint32_t hgrant_z1; // hgrant delayed by one, can safely be read
            // in committed(), maintained by AHB_casi
} TAHBSignals;
```

The signals structure can be accessed by all masters and slaves connected to the bus. Use the `readReq()` function to pass a pointer to the data structure to bus masters and slaves.

During the reset simulation stage. the AHB_casi component calls `readReq()` for all connected slaves and makes the data structure available to the slaves. See the *AMBA Specification Rev 2.0* for details on the individual AHB signals.

B.2.1 AHB bus state machine

A typical implementation of an AHB master uses a simple state machine to access the AHB slave:

1. Start in the `arbitration` state and use the `requestAccess()` and `checkForGrant()` function to obtain a lock on the bus to the slave. If the bus is granted and ready, the `address` state is entered.
2. The master issues a `read()` or `write()` call with dummy data to determine the status of the slave. The `data` state is entered.
3. The `read()` or `write()` call is repeated to transfer the actual data. If the data is transferred, the state returns to `arbitration`.

Note

If there are multiple slaves connected to an AHB master, or the slaves do not have a fixed memory map, the memory map for the slave components must be created before the AHB master accesses the slaves.

For more information on memory map functions, see *The CASITransactionIF interface* on page 2-56.

Figure B-1 on page B-5 shows the sequence of calls to perform a single write. In this example, a single wait state has been inserted by the slave during the read data phase.

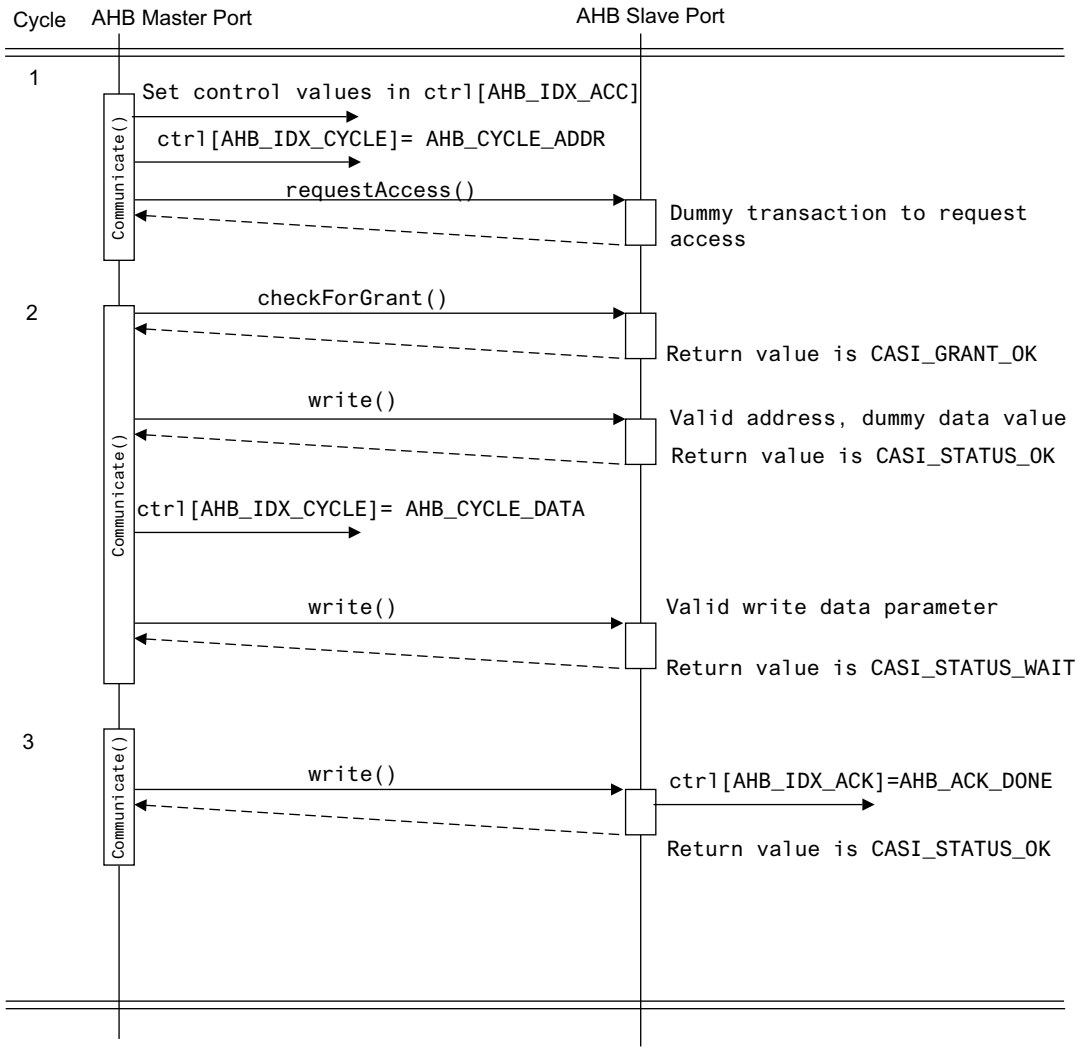

Figure B-1 AHB write

B.3 Implementation details for AHB interfaces

This section describes how the transaction interface functions are implemented for typical AHB masters and slaves.

B.3.1 read() and write()

Perform synchronous read or writes on the slave.

The slave must drive the acknowledgement field, `ctrl[AHB_IDX_ACK]` to proper values. The return value of the data stage read/write call represents the hready signal: `CASI_STATUS_OK` means that `hready` is 1, `CASI_STATUS_WAIT` means `hready` is 0 (wait state occurred). In the address stage no wait states can be inserted.

```
CASIStatus read(uint64_t addr, uint32_t* value, uint32_t* ctrl)
CASIStatus write(uint64_t addr, uint32_t* value, uint32_t* ctrl)
```

where:

`addr` is the transaction address.

`value` is an array of `uint32_t` for the value being read or written.

`ctrl` is an array of `uint32_t`, representing the control fields for the transaction:

- `ctrl[AHB_IDX_CYCLE]` is `AHB_CYCLE_ADDR` for address cycles or `AHB_CYCLE_DATA` for data cycles.
- `ctrl[AHB_IDX_ACC]` contains the control information of the access, `hlock` (falling edge), `hburst`, `htrans`, `hprot` and `hsize`.
- `ctrl[AHB_IDX_ACK]` is the acknowledgement field and must be set to AHB_ACK_DONE by the slave when the transfer is complete.

Example B-1 lists an implementation of `read()` from the `AHB_casi_TSPort.cpp` file:

Example B-1 read()

```
CASIStatus AHB_casi_TSPort::read(uint64_t address, uint32_t *dataPtr,
                     uint32_t *ctrlPtr)
{
    owner->doUpdate = true;
    CASIStatus slaveRet, ret = CASI_STATUS_OK;
    assert(portID < AHB_NUM_MASTERS);
    switch (ctrlPtr[AHB_IDX_CYCLE])
    {
        case AHB_CYCLE_ADDR:
            if (owner->nrSlavesInAddr > 0)
            {
```



```
            if (owner->enableDebugMessages) {
                cout << sc_time_stamp() << ", " << m_name
                     << sc_string::to_string(
                      ": WARNING, port %d read CASI_CYCLE_ADDR", portID)
                     << sc_string::to_string(" addr:0x%08x",
                                             (uint32_t) address)
                     << sc_string::to_string("%d slaves in addr phase.",
                                            owner->nrSlavesInAddr+1)
                     << endl;
            }
            assert(owner->nrSlavesInData == 0);
        }
        assert(owner->nrSlavesInAddr == 0);
        owner->nrSlavesInAddr++;
        owner->acc = ctrlPtr[AHB_IDX_ACC];
        owner->signals.hmastlock = AHB_ACC_DECODE_HLOCK(owner->acc);
        owner->checkHLockForFallingEdge(portID, owner->signals.hmastlock);
        owner->signals.hwrite = 0;
        owner->signals.haddr = (uint32_t)(address & 0xffffffff);
        owner->signals.hmaster = portID;
        owner->decodeControl();
        // The granted master has started the transfer,
        // there is no longer a requirement to check the backoff cycles
        if (portID == owner->signals.hgrant)
        // everything else would be a protocol violation
        {
            owner->currBackoffCycle = 0;
        }
        if (owner->signals.htrans != AHB_TRANS_IDLE)
        {
            owner->bmaster->read(address, dataPtr, ctrlPtr);
        }
        break;
    case AHB_CYCLE_DATA:
        if (owner->enableDebugMessages) {
            cout << sc_time_stamp() << ", " << m_name
                 << ": port " << portID << " read"
                 << sc_string::to_string(" addr:0x%08x", (uint32_t) address)
                 << sc_string::to_string(" ctrl:0x%x", ctrlPtr[AHB_IDX_ACC])
                 <<    endl;
        }
        if (owner->nrSlavesInData > 0)
        {
            if (owner->enableDebugMessages) {
                                cout << sc_time_stamp() << ", " << m_name
               << ": WARNING, port " << portID << " read CASI_CYCLE_DATA"
               << sc_string::to_string(" addr:0x%08x", (uint32_t) address)
               << sc_string::to_string("%d slaves in addr phase.",
                 owner->nrSlavesInAddr+1) << endl;
            }
```

```
                assert(owner->nrSlavesInData == 0);
            }
            owner->nrSlavesInData++;
            owner->dataStagePortID = portID; // for cadi display only
            assert(owner->signals.hready_z1 || (owner->waitPortID == portID));
            slaveRet = owner->bmaster->read(address, dataPtr, ctrlPtr);
            switch (slaveRet)
            {
                case CASI_STATUS_OK:
                    //assert(owner->hready != 0);
                    owner->signals.hready = 1;
                    owner->signals.hrdata[0] = dataPtr[0];
                    owner->signals.hresp = AHB_ACK_DONE;
                    // currently only OK supported
                    if (owner->p_dataWidth > 32)
                    {
                        owner->signals.hrdata[1] = dataPtr[1];
                        if (owner->p_dataWidth > 64)
                        {
                            owner->signals.hrdata[2] = dataPtr[2];
                            owner->signals.hrdata[3] = dataPtr[3];
                        }
                    }
                    break;
                case CASI_STATUS_WAIT:
                    //owner->hready = false;
                    owner->signals.hready = 0;
                    owner->waitPortID = portID;
                    ret = CASI_STATUS_WAIT;
                    break;
                default:
                    break;
            }
            break;
        default:
            assert(0);
            break;
    } // end switch
    return ret;
}
```


B.3.2 readDbg()

This function performs synchronous debug read transaction operations.

Similar to `read()`, except that the slave state must not change.

```
CASIStatus AHB_casi_TSPort::readDbg(uint64_t address, uint32_t *dataPtr,
                                    uint32_t *type)
{
    CASIStatus ret;
    ret = owner->bmaster->readDbg(address, dataPtr, type);
    return ret;
}
```

B.3.3 writeDbg()

This function performs synchronous debug write transaction operations.

Similar to `write()`, except that the slave must not change state.

```
CASIStatus AHB_casi_TSPort::writeDbg(uint64_t address, uint32_t *dataPtr,
                                     uint32_t *type)
{
    CASIStatus ret = CASI_STATUS_OK;
    ret = owner->bmaster->writeDbg(address, dataPtr, type);
    return ret;
}
```

B.3.4 readReq()

AHB Masters can obtain a pointer to the data structure that represents the AHB signal set. The pointer to this data structure is passed to bus masters by the `readReq()` function. It must be called in the `reset` simulation stage. In contrast to the slave interface, the master can also receive its port ID.

```
CASIStatus AHB_casi_TSPort::readReq(uint64_t , uint32_t* value, uint32_t* ctrl,
        casi_transaction_callback_if* )
{
    CASIStatus ret = CASI_STATUS_NOTSUPPORTED;
    if (ctrl)
    {
        owner->pSignals = (TAHBSignals*) ctrl;
        ret = CASI_STATUS_OK;
    }
    return ret;
}
```

where:

ctrl	is a pointer to the AHB signals structure.
value	if not NULL, the ID of the port is be stored at (*Value). This is useful for masters that evaluate the **hmaster** signal.

B.3.5 requestAccess()

The bus masters connected to the slave ports of the bus should call requestAccess() at the rising and falling edges of the hreq signal and/or on the rising edge of hlock.

The value of hreq is encoded into bit 0 of the address parameter of this function (1 = hreq asserted, 0 = hreq deasserted)

The value of hlock is coded in bit 1 (1 = hlock asserted, 0 = hlock deasserted). The falling edge event of hlock is not handled by this call due to its different timing (it is set during the address stage).

The changed semantics of requestAccess() means that this function is not called if hreq and hlock are asserted and unchanged from the previous cycle. The return value of requestAccess() can be ignored and always return CASI_GRANT_DENIED.

It is possible to deassert hlock by driving an IDLE cycle.

```
AHB_casi_TSPort::requestAccess(uint64_t address)
{
    assert(portID < owner->max_ports);
    // set the request flag for the respective port ID and lock
    // (coded in bit 1 (2^1))
    owner->registerRequest(portID, address);
    owner->doUpdate = true;

    return  CASI_GRANT_DENIED;
}
```

B.3.6 checkForGrant()

Call checkForGrant() from the masters to retrieve the value of hgnt. The timing of checkForGrant() is such that if it is called in cycle n, it gets the value of hgnt that has been calculated by the AHB_casi component during its update() phase of cycle n–1. This the value must be used to identify if the master will own the bus in cycle n+1.

Caution

The master must include the status of hready when evaluating hgnt.


```
AHB_casi_TSPort::checkForGrant(uint64_t addr)
{
    CASIGrant grant;
    UNUSEDARG(addr);
    owner->doUpdate = true;
    grant = (owner->signals.hgrant == portID) ?
                    CASI_GRANT_OK : CASI_GRANT_DENIED;
    return grant;
}
```

B.3.7 getAddressRegions()

This function returns the structure of the memory address regions supported by this interface. The address regions are returned as start value and block size in addition to a region name.

Note

More than one range can be specified. The number of expected regions is given by the function `getNumRegions()` and you must allocate the required memory for the parameters.

This function returns the default address mapping.

```
void AHB_casi_TSPort::getAddressRegions(uint64_t* start, uint64_t* size,
                                        const char* name[])
{
    start[0] = owner->Base;
    size[0]  = owner->Size;
    name[0]  = "AHB_casi";
}
```

where:

`start` is the memory address region start addresses array.

`size` is the memory address region sizes array.

`start` is the memory address region names array.

Appendix C
AMBA® AXI TLM Specification for CASI

This appendix describes the *AMBA Advanced eXtensible Interface* (AXI) *Transaction Level Model* (TLM) specification for the ESL APIs. It contains the following sections:

- *Introduction to AXI* on page C-2
- *Introduction to the CASI TLM for AXI* on page C-9
- *ESL API implementation of the AXI TLM* on page C-44.

Note

See the *AMBA AXI Protocol v1.0 Specification* (ARM IHI0022) for detailed specifications of the AMBA AXI bus. AMBA is the *Advanced Microcontroller Bus Architecture* specification.

C.1 Introduction to AXI

AXI is a point to point connection between a single master and a single slave interface.

Note

Multiple masters can, however, communicate with multiple slaves by using a matrix component that manages arbitration, multiplexing, and address decoding.

The AXI protocol is implemented by five independent channels:

Write address channel (AW)

This channel communicates the address from the master to the slave for write requests. This channel also communicates information about the write access (for example, write burst count and word size).

Write data channel (W)

This channel communicates the write data from the master to the slave.

Write response channel (B)

This channel communicates the status from the slave to a write request from the master and indicates whether the write attempt was successful. The AXI slave drives the status (typically OKAY) onto the response channel.

Read address channel (AW)

This channel communicates the address from the master to the slave for read requests. The channel also communicates information about the read access (for example, read burst count and word size).

Read data channel (R)

This channel communicates the data and status from the slave to the master for read requests. The AXI slave drives the data and status onto the data channel.

A block diagram of two components that communicate over an AXI bus is shown in Figure C-1 on page C-3:

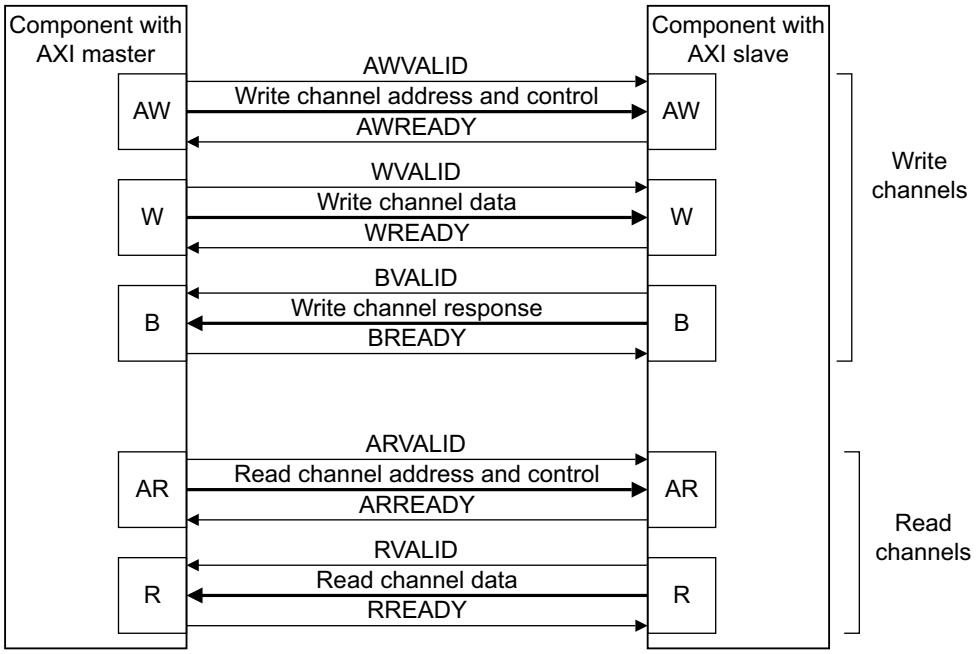

Figure C-1 Block diagram of master and slave components connected over an AXI bus

C.1.1 Control and data signals on the AXI channels

The signals on each of the five AXI channels are listed in Table C-1 through Table C-5 on page C-7. Refer to *AMBA AXI Protocol v1.0 Specification* for more information on the signals.

Table C-1 Signals on the write address channel (AW)

Signal	Source	Description
AWID[3:0]	Master	Write address ID. This signal is the identification tag for the write address group of signals.
AWADDR[31:0]	Master	Write address. The write address bus gives the address of the first transfer in a write burst transaction. The associated control signals are used to determine the addresses of the remaining transfers in the burst.
AWLEN[3:0]	Master	Burst length. The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address.

Table C-1 Signals on the write address channel (AW) (continued)

Signal	Source	Description
AWSIZE[2:0]	Master	Burst size. This signal indicates the size of each transfer in the burst. Byte lane strobes indicate exactly which byte lanes to update.
AWBURST[1:0]	Master	Burst type. The burst type, coupled with the size information, details how the address for each transfer within the burst is calculated.
AWLOCK[1:0]	Master	Lock type. This signal provides additional information about the atomic characteristics of the transfer.
AWCACHE[3:0]	Master	Cache type. This signal indicates the bufferable, cacheable, write-through, write-back, and allocate attributes of the transaction.
AWPROT[2:0]	Master	Protection type. This signal indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access.
AWVALID	Master	Write address valid. If this signal is 1, valid write address and control information are available. The address and control information remain stable until the address acknowledge signal, **AWREADY**, goes HIGH.
AWREADY	Slave	Write address ready. If this signal is 1, the slave is ready to accept an address and associated control signals.

Table C-2 Signals on the write data channel (W)

Signal	Source	Description
WID[3:0]	Master	Write ID tag. This signal is the ID tag of the write data transfer. The WID value must match the AWID value of the write transaction.
WDATA[31:0]	Master	Write data. The write data bus can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.
WSTRB[3:0]	Master	Write strobes. This signal indicates which byte lanes to update in memory. There is one write strobe for each eight bits of the write data bus. Therefore, WSTRB[n] corresponds to WDATA[(8 n) + 7:(8 n)].
WLAST	Master	Write last. This signal indicates the last transfer in a write burst.
WVALID	Master	Write valid. If this signal is 1, valid write data and strobes are available.
WREADY	Slave	Write ready. If this signal is 1, the slave can accept the write data.

Table C-3 Signals in the write response channel (B)

Signal	Source	Description
BID[3:0]	Slave	Response ID. The identification tag of the write response. The BID value must match the AWID value of the write transaction to which the slave is responding.
BRESP[1:0]	Slave	Write response. This signal indicates the status of the write transaction. The allowable responses are OKAY, EXOKAY, SLVERR, and DECERR.
BVALID	Slave	Write response valid. If this signal is 1, a valid write response is available.
BREADY	Master	Response ready. If this signal is 1, the master can accept the response information.

Table C-4 Signals on the read address channel (AR)

Signal	Source	Description
ARID[3:0]	Master	Read address ID. This signal is the identification tag for the read address group of signals.
ARADDR[31:0]	Master	Read address. The read address bus gives the initial address of a read burst transaction. Only the start address of the burst is provided and the control signals that are issued alongside the address detail how the address is calculated for the remaining transfers in the burst.
ARLEN[3:0]	Master	Burst length. The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address.
ARSIZE[2:0]	Master	Burst size. This signal indicates the size of each transfer in the burst.
ARBURST[1:0]	Master	Burst type. The burst type, coupled with the size information, details how the address for each transfer within the burst is calculated.
ARLOCK[1:0]	Master	Lock type. This signal provides additional information about the atomic characteristics of the transfer.
ARCACHE[3:0]	Master	Cache type. This signal provides additional information about the cacheable characteristics of the transfer.
ARPROT[2:0]	Master	Protection type. This signal provides protection unit information for the transaction.
ARVALID	Master	Read address valid. If this signal is 1, the read address and control information is valid and will remain stable until the address acknowledge signal, **ARREADY**, is high.
ARREADY	Slave	Read address ready. If this signal is 1, the slave is ready to accept an address and associated control signals.

Table C-5 Signals in the read data channel (R)

Signal	Source	Description
RID[3:0]	Slave	Read ID tag. This signal is the ID tag of the read data group of signals. The RID value is generated by the slave and must match the ARID value of the read transaction to which it is responding.
RDATA[31:0]	Slave	Read data. The read data bus can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.
RRESP[1:0]	Slave	Read response. This signal indicates the status of the read transfer. The allowable responses are OKAY, EXOKAY, SLVERR, and DECERR.
RLAST	Slave	Read last. This signal indicates the last transfer in a read burst.
RVALID	Slave	Read valid. If this signal is 1, the required read data is available and the read transfer can complete.
RREADY	Master	Read ready. If this signal is 1, the master can accept the read data and response information.

C.1.2 Hardware flow control signals

All of the AXI channels have versions of **READY** and **VALID** signals that control the flow between the master and slave. (The read data channel for example has **RVALID** and **RREADY**.) For systems implemented in hardware, the handshaking is managed by monitoring the state of **READY** and **VALID** on the rising clock edge as shown in Figure C-2 on page C-8:

Note

Figure C-2 on page C-8 shows **READY** low until **VALID** is high. The **VALID** and **READY** signals can, however, go high in either order. The only requirement for signifying that the transfer is complete is that both signals are high on the rising edge of the clock.

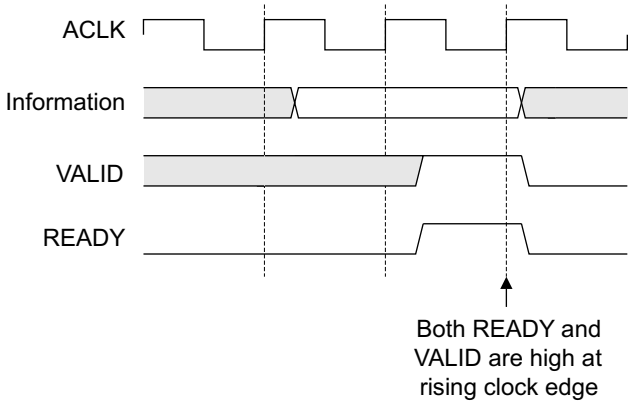

Figure C-2 READY and VALID handshake signals

C.2 Introduction to the CASI TLM for AXI

This section describes how the CASI TLM for AXI uses data structures and access functions to provide a simulation equivalent to the AXI hardware bus.

The TLM API does not contain five distinct pairs of master/slave ports that were shown in Figure C-1 on page C-3 for the hardware implementation. The communication between the master and slave ports is done by passing a transaction info structure between the `driveTransaction()` and `notifyEvent()` functions. The signals that are managed by the hardware channels in a physical system are managed by multiple calls to `driveTransaction()` and `notifyEvent()` in the TLM API.

Figure C-3 shows a simplified block diagram of a system consisting of:

- a component named `AXI_Master_casi` that contains an AXI master port.
- a component named `AXI_Slave_casi` that contains an AXI slave port.

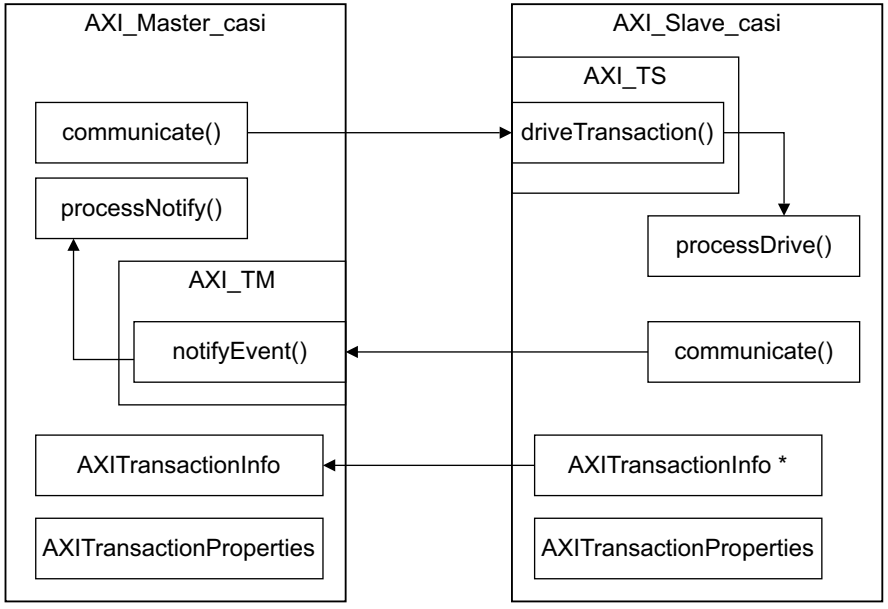

Figure C-3 Simplified AXI TLM block diagram

The classes and structures used to implement the AXI TLM are shown in Figure C-4 on page C-10:

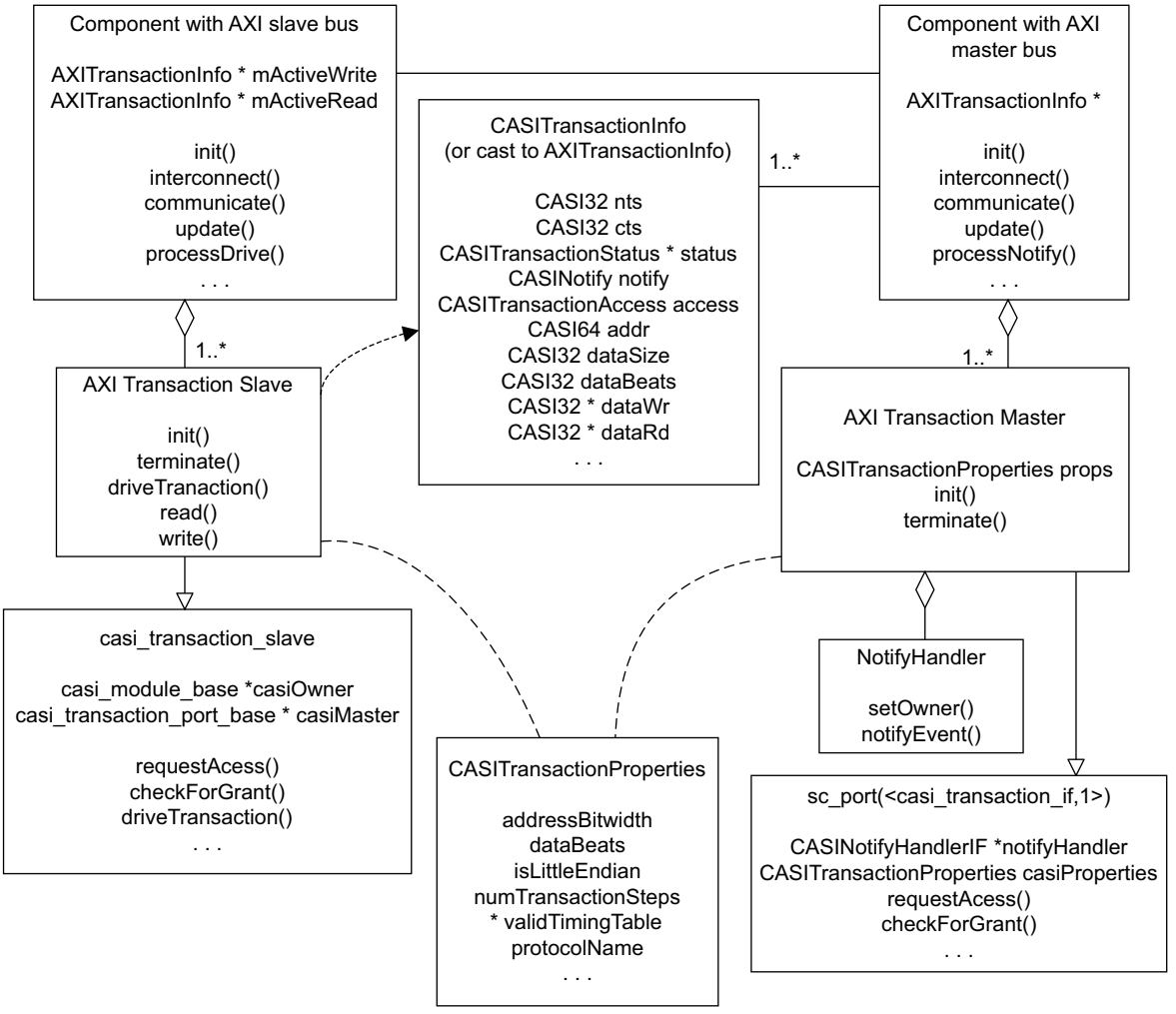

Figure C-4 AXI classes

C.2.1 CASITransactionInfo structure

There would be considerable overhead for a SystemC model to attempt to manage each of the individual signals in the five channels. A Transaction Level Model, however, groups the individual signals into a data structure that is passed as a parameter by the high-level access functions.

The data and control information for an AXI transaction is managed by a transaction info structure that is shared by all five channels.

For AXI hardware, the channels can be active at the same time. For the TLM implementation, different channel actions are managed by separate transaction steps that reads and modifies the data structure to reflect the behavior of the corresponding AXI signals.

Example C-1 lists the contents of the `CASITransactionInfo` structure that is defined in `CASITypes.h`.

Example C-1 CASI transaction info structure

```
struct CASITransactionInfo
{
/* Transaction Control */
    // pointer to the initiator of the transaction
    CASITransactionMasterIF* initiator;
    // total number of transaction steps for this transaction
    uint32_t nts;
    // current transaction step of transaction
    uint32_t cts;
    // array containing current status for each step in the transaction
    CASITransactionStatus* status;
    // use notify request (CASI_NOTIFY_YES/CASI_NOTIFY_NO)
    CASINotify notify;

/* Pre-defined fixed Transaction Elements */
    // data direction (CASI_ACCESS_READ, CASI_ACCESS_WRITE)
    CASITransactionAccess access;
    // address of shared resource to be accessed
    uint64_t addr;
    // size of the data transfer (per beat) in MAUs (e.g. number of bytes)
    uint32_t dataSize;
    // number of data beats to be transferred (burst transfers)
    uint32_t dataBeats;
    // write data array
    uint32_t* dataWr;
    // read data array
    uint32_t* dataRd;

/* User-defined Transaction Elements */
    // flags/sideband/control signals controlled by the sender (master)
    uint32_t* masterFlags;
    // flags/sideband/control signals controlled by the receiver (slave)
    uint32_t* slaveFlags;
    …
}
```

In addition to the expected bus-related signals (for example `dataWr` for the write data signals **WDATA[31:0]**), there are also structure members that are used by the interface to manage the progress of the transaction:

`initiator` is the component that initiated the AXI transaction.

`nts` is the number of transaction steps required for the current transaction. All AXI operations require multiple transaction steps to complete.

`cts` is the current transaction step.

`status` is an array with one element for each of the transaction steps. The contents of `status[`*`n`*`]` indicates the status of transaction step *n*.

`notify` determines how the slave port responds to the step.

The structure can be extended with user-defined elements that can be used by the sender or receiver to provide additional information that is not part of the AXI protocol:

`masterFlags` is an array that can be filled in by master. These flags are valid for step n when `status[n]` is greater than or equal to `CASI_MASTER_READY`.

`slaveFlags` is an array that can be filled in by slave. These flags are valid for step n when `status[n]` is greater than or equal to `CASI_SLAVE_READY`.

… indicates that additional user-defined elements might be added to the structure in the future.

C.2.2 CASITransactionInfo.masterFlags array

The masterFlags array contains flags that are set by the AXI master to describe the transaction. The elements of the array can be accessed by the definitions in the `AXITransactionMasterFlagEnum`:

```
enum AXITransactionMasterFlagEnum
{
    AXI_MF_ID = 0,      // transaction ID
    AXI_MF_BURST_TYPE,  // address auto increment mode
    AXI_MF_LOCK,        // lock signals
    AXI_MF_CACHE,       // cache flag signals
    AXI_MF_PROTECTION,  // protection flag signals
    AXI_MF_DATA_STROBE, // data strobe signals
    AXI_MF_RESPONSE,    // response signal
    AXI_MF_CHANNEL,     // non-architectural: short cut to analyzing access,
                        // cts and respective status
    AXI_MF_AW_INFO,     // non-architectural: additional info for channel aw
    AXI_MF_AUSER,       // user flag for address channel (read/write)
    AXI_MF_DUSER,       // user flag for data channel (read/write)
```



```
    AXI_MF_BUSER,       // user flag for write response channel
    AXI_MF_LAST         // total number of Master Flags
};
```

CASITransactionInfo.masterFlags[AXI_MF_CHANNEL] indicates the channel that is active for the current step in the transaction. It can have one of the values from AXIChannelEnum:

```
enum AXIChannelEnum
{
    AXI_CHANNEL_AR     =  1,
    AXI_CHANNEL_R      =  2,
    AXI_CHANNEL_AW     =  4,
    AXI_CHANNEL_W      =  8,
    AXI_CHANNEL_B      = 16,
    AXI_TRANSFER_EMPTY = 32,    // Indicates a 'dummy' driveTransaction() or
        // notifyEvent() call. There is not a 'valid' transfer in a channel.
        // this value is used for ACI
    AXI_TRANSFER_LAST  = 64     // Indicates the last call for the port
        // (last driveTransaction() or last notifyEvent())
        // so the number of dummy calls can be reduced
};
```

C.2.3 AXITransactionInfo class

The AXI_Transaction.h file contains the definition of the AXITransactionInfo class that extends the CASITransactionInfo structure for AXI transactions. The class definition is listed in Example C-2:

Example C-2 AXITransactionInfo class

```
class AXITransactionInfo : public CASITransactionInfo
{
    public:
        AXITransactionInfo();
        void init (uint32_t dataBitwidth);

        // for compatibility
        // pointer to something used exclusively by the master port
        void*           m_pExclusiveMasterPortRelated;
        // pointer to something used exclusively by the slave port
        void*           m_pExclusiveSlavePortRelated;
        // next free p is required by SimpleHeap, must be public
        AXITransaction* nextFreeP;
        AXITransaction* m_pACIrelatedSPortInfoInt;
        AXITransaction* m_pACIrelatedMPortInfoExt;
        uint32_t         m_ACISPortIdxInt, m_ACIMPortIdxInt;
        uint32_t         m_ACISPortIdxExt, m_ACIMPortIdxExt;
```

```
        void*           m_pPad;     // for preliminary internal extensions
        // implementations can adapt their behavior according to
        // the value of the pad version
        uint32_t         m_padVersion;
        uint32_t         m_dataBitwidth;
        uint32_t         m_datauint32width;      // = (dataBitwidth + 31)/32
        uint32_t         m_nextCts;
        uint32_t         m_nextStrb;
};
```

Communication between the master and slave port is done by:

- The master calling the `driveTransaction()` function in the slave.
- The slave calling the `notifyEvent()` function in the master.

An AXITransactionInfo object is passed as a parameter for both functions.

C.2.4 AXI_TM::notifyEvent()

An example of a AXI transaction master port (`AXI_TM`) using the `notifyEvent()` function to process notifications from the slave port is listed in Example C-3:

Example C-3 Processing the AXITransactionInfo structure in the master port

```
inline void AXI_TM::NotifyHandler::notifyEvent (CASITransactionInfo* info)
{
    mOwner->processNotify(static_cast<AXITransactionInfo*>(info));
}
```

Note

The code examples in this appendix are based on the `AXI_1m1s` system.

The code in `AXI_Master_casi.cpp` is the source for the component that owns the AXI master port. It creates the new port as shown in Example C-4.

Example C-4 Creating the master port

```
AXI_Master_casi::AXI_Master_casi(sc_module_name name): CASIModule(name, NULL)
{
    mPort = new AXI_TM(this, "axi_m");
}
```


The constructor for the port is in AXI_TM.cpp and it sets the owner and notify handler as shown in Example C-5:

Example C-5 Setting the master port owner

```
AXI_TM::AXI_TM (AXI_Master_casi *owner, const std::string& name) :
                 sc_port<casi_transaction_if, 1>(owner, name)
{
    mNotifyHandler.setOwner(owner); // sets local mOwner=owner
    setNotifyHandler(&mNotifyHandler);
}
```

The processNotify() function is implemented in the component that owns the AXI master port.

Example C-6 processNotify() example

```
void AXI_Master_casi::processNotify (AXITransactionInfo* axi)
{
    if (axi->masterFlags[AXI_MF_CHANNEL] & AXI_CHANNEL_B)
    {
       axi->status[axi->cts] = CASI_SLAVE_READY;
       printf("AXI_Master: Cycle %4i: Channel B  received - Write finished\n\n", mCycleCount);
    }
    else if (axi->masterFlags[AXI_MF_CHANNEL] & AXI_CHANNEL_W)
    {
       mAxiState = (mAxi.cts == mAxi.dataBeats ? STATE_B : mAxiState);
       printf("AXI_Master: Cycle %4i: Channel W  received ready\n", mCycleCount);
    }
    else if (axi->masterFlags[AXI_MF_CHANNEL] & AXI_CHANNEL_R)
    {
      axi->status[axi->cts] = CASI_SLAVE_READY;
      if (axi->cts < axi->nts-1)
        {
          printf("AXI_Master: Cycle %4i: Channel R  received - data %x\n", mCycleCount,
                  axi->dataRd[axi->cts-1]);
        }
        else
        {
           printf("AXI_Master: Cycle %4i: Channel R  received - data %x - Read finished\n\n",
                    mCycleCount, axi->dataRd[axi->cts-1]);
        }
    }
}
```

C.2.5 AXI_TS::driveTransaction()

The AXI transaction slave port (AXI_TS) uses the driveTransaction() function to process transaction requests from the master port as listed in Example C-7:

Example C-7 Processing the AXITransactionInfo structure in the slave port

```
void AXI_TS::driveTransaction (AXITransactionInfo* info)
{
    mOwner->processDrive(static_cast<AXITransactionInfo*>(info));
}
```

mOwner is a pointer to the component that owns the slave port and is set in the port constructor as shown in Example C-8.

Example C-8 Slave port constructor

```
AXI_TS::AXI_TS (AXI_Slave_casi* owner, const std::string& name) :
                            casi_transaction_slave(owner, name)
{
    mOwner = owner;
}
```

The processDrive() function is implemented in the component that owns the AXI slave port and mOwner is a pointer to this component. Example C-9 shows the processDrive() function from the AXI_Slave_casi.cpp file that defines the component that owns the AXI slave port.

Example C-9 processDrive() example

```
void AXI_Slave_casi::processDrive (AXITransactionInfo* axi)
{
    if (!(axi->masterFlags[AXI_MF_CHANNEL] & AXI_TRANSFER_EMPTY))
    {
        if ((axi->masterFlags[AXI_MF_CHANNEL] & AXI_CHANNEL_AW) &&
                    mActiveWrite == NULL)
        {
            axi->status[0] = CASI_SLAVE_READY;
            mActiveWrite = axi;

            printf("AXI_Slave:  Cycle %4i: Channel AW received - address %x\n",
                       mCycleCount, axi->addr);
```

```
        }
        if ((axi->masterFlags[AXI_MF_CHANNEL] & AXI_CHANNEL_AR) &&
                    mActiveRead == NULL)
        {
            axi->status[0] = CASI_SLAVE_READY;
            mActiveRead = axi;

            printf("AXI_Slave:  Cycle %4i: Channel AR received - address %x\n",
                       mCycleCount, axi->addr);
        }
        if (axi->masterFlags[AXI_MF_CHANNEL] & AXI_CHANNEL_W)
        {
            axi->status[axi->cts] = CASI_SLAVE_READY;
            printf("AXI_Slave:  Cycle %4i: Channel W  received - data %x\n",
                       mCycleCount, axi->dataWr[axi->cts-1]);
        }
    }
}
```

Note

The `processDrive()` and `processNotify()` functions have a parameter of type `CASITransactionInfo`, but this is cast to type `AXITransactionInfo` for use by the processing functions in the components that contain the master and slave ports.

C.2.6 Transaction properties structure

AXI ports can be implemented with different characteristics such as bit width, minimum addressable unit (byte or word for example), or specific interface functions. The `CASITransactionProperties` structure is associated with an AXI master port and specifies the properties for the master. This structure is also used to ensure that the attached slave port uses a compatible implementation. The structure is listed in Example C-10.

Example C-10 CASITransactionProperties structure

```
// port properties
struct CASITransactionProperties
{
    // the transaction version used for this transaction
    CASIVersion casiVersion;
    bool useMultiCycleInterface; // driveTransaction() is used
                                 // instead of read() or write()
    uint8_t addressBitwidth; // address bit width
    uint32_t mauBitwidth;    // minimal addressable unit
```

```
    uint32_t dataBitwidth;   // maximum data bit width (data bus)
    uint32_t dataBeats;      // maximum data beats in a burst transaction
    bool isLittleEndian;    // alignment of MAUs
    bool isOptional;        // if true this port can be disabled
    bool supportsAddressRegions; // M/S can negotiate address mapping
    uint32_t numCtrlFields;   // # of ctrl elements used
    uint32_t numSlaveFlags;   // # of slave flag elements used
    uint32_t numMasterFlags;  // # of master flag elements used
    uint32_t numTransactionSteps; // # of transaction steps (maximum)
    uint32_t* validTimingTable; // lookup table providing transaction
                                // step of validity
    uint32_t protocolID;      // magic number of the protocol
                              // (Vendor/Protocol-ID)
    bool supportsNotify;      // M/S do event based execution upon
                              // notify request
    bool supportsBurst;       // burst transfer capability
                              // (true/false)
    bool supportsSplit;       // split transfer capability
                              // (true/false)
    bool isAddrRegionForwarded;
    CASITransactionMasterIF * forwardAddrRegionToMasterPort;

    /* Visualization and Protocol Extension Area */
    CASITransactionDetails* details; // for future expansion
};
```

The contents of the `CASITransactionProperties` structure for the bus master is initialized and destroyed by the following macros that are defined in the `AXI_Transaction.h` file:

- `AXI_INIT_TRANSACTION_PROPERTIES(PROP__, DATA_BITWIDTH__)`
- `AXI_DESTRUCT_TRANSACTION_PROPERTIES(PROP__)`

C.2.7 AXITransactionStepEnum

The data members of the `CASITransactionInfo` structure are filled in by different steps in the transaction sequence. The step types are shown in Example C-11:

Example C-11 AXITransactionStepEnum

```
enum AXITransactionStepEnum
{
   AXI_STEP_ADDRESS = 0, // Read or Write address channel
   AXI_STEP_DATA0,       // Read or Write data channel beat 0
   AXI_STEP_DATA1,       // Read or Write data channel beat 1
   AXI_STEP_DATA2,       // Read or Write data channel beat 2
   AXI_STEP_DATA3,       // Read or Write data channel beat 3
```



```
   AXI_STEP_DATA4,       // Read or Write data channel beat 4
   AXI_STEP_DATA5,       // Read or Write data channel beat 5
   AXI_STEP_DATA6,       // Read or Write data channel beat 6
   AXI_STEP_DATA7,       // Read or Write data channel beat 7
   AXI_STEP_DATA8,       // Read or Write data channel beat 8
   AXI_STEP_DATA9,       // Read or Write data channel beat 9
   AXI_STEP_DATA10,      // Read or Write data channel beat 10
   AXI_STEP_DATA11,      // Read or Write data channel beat 11
   AXI_STEP_DATA12,      // Read or Write data channel beat 12
   AXI_STEP_DATA13,      // Read or Write data channel beat 13
   AXI_STEP_DATA14,      // Read or Write data channel beat 14
   AXI_STEP_DATA15,      // Read or Write data channel beat 15
   AXI_STEP_RESPONSE,    // Write response channel for 16 beat transfer only
   AXI_STEP_LAST
};
```

The number of steps actually taken by a transaction depend on the transaction type. A one word read requires a minimum of two steps (address out and data in), while a 16-word write requires at least 18 steps (address out, 16 times data out, response back) The following variables are set or read by different steps:

`nts` This transaction info member indicates the maximum number of steps required to complete the transaction sequence.

`cts` This transaction info member indicates the current step in the transaction sequence.

`*status` This transaction info member array indicates the status for each of the steps in the transaction sequence. See *CASITransactionStatus* on page C-20.

`*validTimingTable`

This array, that is part of the transaction properties structure, enables a generic probe (a monitor, for example) to determine whether an entry in the data structure is valid or should be ignored. If, for example, the contents of `validTimingTable[3]` is `AXI_STEP_ADDRESS` this indicates that the third predefined transaction element (that is, `dataSize`) must be valid in the transaction step that transfers the read or write address. This array, and the other members of the transaction properties structure, are filled in by the `AXI_INIT_TRANSACTION_PROPERTIES` macro.

Caution

The `AXI_INIT_TRANSACTION_PROPERTIES` macro must be modified if the AXI bus for the component has different characteristics than those used in the definition of the macro in `AXI_Transaction.h`.

C.2.8 CASITransactionStatus

The CASITypes.h file contains an enumeration for the possible states of the master and slave ports. The CASITransactionStatus enumeration is listed in Example C-12:

Example C-12 CASITransactionStatus enumeration

```
enum CASITransactionStatus
{
  //  Master status responses.
  // Transaction step does not contain valid information.
  CASI_MASTER_WAIT = 0,
  // Transaction step does contain valid info by master.
  CASI_MASTER_READY = 1,

  //  Slave status responses
  // Transaction step does not contain valid response.
  CASI_SLAVE_WAIT = 2,
  // Transaction step does contain valid response by Slave.
  CASI_SLAVE_READY = 3,
  // Slave acknowledges "cancelTransaction(?)".
  CASI_SLAVE_READY_CANCEL = 4,
  // Slave breaks transaction, can be continued by master.
  CASI_SLAVE_READY_SPLIT = 5,
  // Slave asks for restart of same transaction.
  CASI_SLAVE_READY_RETRY = 6,
  // Slave reports transaction error.
  CASI_SLAVE_READY_ERROR = 7
};
```

For the AXI implementation, the mapping of the handshake signals map onto the status signals is listed in Table C-6:

Table C-6 Handshake signals and status

Status	VALID	READY	Description
CASI_MASTER_WAIT	0	x	The sender does not have valid data.
CASI_MASTER_READY	1	0	The sender has valid information to transfer and is waiting for the receiver to become ready.
CASI_SLAVE_READY	1	1	Information has been transferred.

The negotiation of **VALID** and **READY** occurs for each channel that is required for the transfer. The correspondence between the current transaction step (`cts`) and the channel is listed in Table C-7 and Table C-8:

Table C-7 Current transaction and channels for reads

cts	Channel	Description
0	AR	The address to read from is transferred to slave.
1 to (`nts`-1)	R	Read data is transferred. If a read burst is being performed, multiple transaction steps are required and `nts` is greater than 1.

Table C-8 Current transaction and channels for writes

cts	Channel	Description
0	AW	Address to write to is transferred to slave. **Note** The data channel can transport information in advance, of or in parallel to, the write address channel. The slave must test for the completion of the address transfer by testing: `status[AXI_STEP_ADDRESS] >= CASI_SLAVE_READY`
1 to (`nts`-2)	W	Write data is transferred. If a write burst is being performed, multiple transaction steps are required.
`nts`-1	B	The response to the write is transferred from the slave to the master. The response step is active when: `(cts == (nts - 1)) && (status[cts] >=CASI_MASTER_READY).`

Note

Advancing to the next transaction step might require several clock cycles because of wait states issued by the receiver.

C.2.9 AXI transfer sequences

This section describes the different sequences of events that can occur during AXI read or write operations.

Table C-9 lists the cycles for an example single-word write operation with waits on each channel:

Table C-9 Cycle by cycle activity for a write transaction

Cycle	cts	status[0:2]	Description
0	0	CASI_MASTER_WAIT, CASI_MASTER_WAIT, CASI_MASTER_WAIT	No action. AXI slave function driveTransaction() is not called by AXI master.
1	0	-	Null transaction (optional). Master calls the slave with an empty transaction. This might be used for arbitration.
2	0	CASI_MASTER_READY, CASI_MASTER_WAIT, CASI_MASTER_WAIT	Address transfer. Master calls the slave with a valid transaction. The slave does not change the status on return from driveTransaction().
3	0	CASI_SLAVE_READY, CASI_MASTER_WAIT, CASI_MASTER_WAIT	Master calls the slave again. Slave accepts address by changing status.
4	1	CASI_SLAVE_READY, CASI_MASTER_READY, CASI_MASTER_WAIT	Write data transfer. Master calls slave with a transaction with the data elements filled in. The slave does not change the status on return.
5	1	CASI_SLAVE_READY, CASI_SLAVE_READY, CASI_MASTER_WAIT	Master calls the slave again. Slave accepts data by changing status.
6	2	CASI_SLAVE_READY, CASI_SLAVE_READY, CASI_MASTER_READY	Write response transfer. Slave calls notifyEvent() in AXI master with the response elements filled in. The master does not change the status on return.
7	2	CASI_SLAVE_READY, CASI_SLAVE_READY, CASI_MASTER_READY	Slave calls the master again. Master accepts data by changing status.
8	0	CASI_MASTER_WAIT, CASI_MASTER_WAIT, CASI_MASTER_WAIT	No action (optional). If no additional writes are pending, all of the channels are inactive and driveTransaction() is not called from master.

The actual transfer sequence can differ considerably from the sequence shown in Table C-9:

- If wait states are not required, the receiving channel can set CASI_SLAVE_READY before returning rather than waiting for the next call. For example, cycle 3 can be combined with cycle 2.
- The write address and write data can be valid in the same cycle.

For cycle 2 in Table C-9 on page C-22, if the first three elements of the status array were CASI_MASTER_READY, CASI_MASTER_READY, CASI_MASTER_WAIT, the slave could accept both of these elements and set the array contents to CASI_SLAVE_READY, CASI_SLAVE_READY, CASI_MASTER_WAIT on return.

- Cycles 4 and 5 transfer a single word. If a burst-write operation is occurring, multiple cycles are required for the transfer. For a burst operation, the number of transaction steps (nts) is larger.

C.2.10 AXI_Master_casi::communicate()

The communicate() function in the AXI master uses a local state variable (mAxiState) and the transaction info structure (mAXI for this example) to control the sequencing of the various transaction steps. Example C-13 shows an example from AXI_Master_casi.cpp:

Example C-13 AXI master communicate()

```
void AXI_Master_casi::communicate ()
{
    switch (mAxiState)
    {
        case STATE_AW:
            if (mAxi.status[0] != CASI_MASTER_READY)
            {
                mAxi.reset();
                mAxi.access                         = CASI_ACCESS_WRITE;
                mAxi.masterFlags[AXI_MF_CHANNEL]    = AXI_CHANNEL_AW | AXI_TRANSFER_LAST;
                mAxi.status[0]                      = CASI_MASTER_READY;
                mAxi.masterFlags[AXI_MF_ID]         = 0;
                mAxi.addr                           = 0;
                mAxi.dataBeats                      = 8;
                mAxi.nts                            = 1+mAxi.dataBeats+1;
                mAxi.dataSize                       = (1 << AXI_BURST_SIZE_4);
                mAxi.masterFlags[AXI_MF_BURST_TYPE] = AXI_BURST_INCR;
                mAxi.masterFlags[AXI_MF_LOCK]       = AXI_LOCK_NORMAL;
                mAxi.masterFlags[AXI_MF_PROTECTION] = 0;
                mAxi.masterFlags[AXI_MF_CACHE]      = 0;
                mAxi.masterFlags[AXI_MF_AUSER]      = 0;
            }
            printf("AXI_Master: Cycle %4i: Channel AW driven   - address %x\n", mCycleCount,
                     mAxi.addr);

            mPort->driveTransaction(&mAxi);
            mAxiState = (mAxi.status[AXI_STEP_ADDRESS] >= CASI_SLAVE_READY ?
                   STATE_W : mAxiState);
            break;
```

```
case STATE_W:
    if (mAxi.status[mAxi.cts] >= CASI_SLAVE_READY)
    {
        mAxi.cts++;

        uint32_t dataToWrite = mAxi.cts-1;
        mAxi.dataWr[mAxi.cts-1]             = dataToWrite;
        mAxi.masterFlags[AXI_MF_CHANNEL]    = AXI_CHANNEL_W | AXI_TRANSFER_LAST;
        mAxi.status[mAxi.cts]               = CASI_MASTER_READY;
        mAxi.masterFlags[AXI_MF_DATA_STROBE]= 0xF;
        mAxi.masterFlags[AXI_MF_DUSER]      = 0;
    }
    printf("AXI_Master: Cycle %4i: Channel W  driven   - data %x\n", mCycleCount,
             mAxi.dataWr[mAxi.cts-1]);
    mPort->driveTransaction(&mAxi);
    mAxiState = (((mAxi.status[mAxi.cts] >= CASI_SLAVE_READY) && (mAxi.cts == mAxi.dataBeats))
                    ? STATE_B : mAxiState);
    break;
case STATE_B:
    mPort->driveTransaction(&mEmptyTransaction);
    break;

case STATE_AR:
    if (mAxi.status[0] != CASI_MASTER_READY)
    {
        mAxi.reset();
        mAxi.access                         = CASI_ACCESS_READ;
        mAxi.masterFlags[AXI_MF_CHANNEL]    = AXI_CHANNEL_AR | AXI_TRANSFER_LAST;
        mAxi.status[0]                      = CASI_MASTER_READY;
        mAxi.masterFlags[AXI_MF_ID]         = 0;
        mAxi.addr                           = 0;
        mAxi.dataBeats                      = 8;
        mAxi.nts                            = 1+mAxi.dataBeats;
        mAxi.dataSize                       = (1 << AXI_BURST_SIZE_4);
        mAxi.masterFlags[AXI_MF_BURST_TYPE] = AXI_BURST_INCR;
        mAxi.masterFlags[AXI_MF_LOCK]       = AXI_LOCK_NORMAL;
        mAxi.masterFlags[AXI_MF_PROTECTION] = 0;
        mAxi.masterFlags[AXI_MF_CACHE]      = 0;
        mAxi.masterFlags[AXI_MF_AUSER]      = 0;
    }
    printf("AXI_Master: Cycle %4i: Channel AR driven   - address %x\n", mCycleCount,
        mAxi.addr);

    mPort->driveTransaction(&mAxi);
    mAxiState = (mAxi.status[AXI_STEP_ADDRESS] >= CASI_SLAVE_READY
                 ? STATE_R : mAxiState);
    break;

case STATE_R:
```



```
            mPort->driveTransaction(&mEmptyTransaction);
            break;
    }
}
```

C.2.11 AXI_Master_casi::update()

The `update()` function in the AXI master sets the a local state variable (`mAxiState`) to indicate that response or read data step has finished. Example C-14 shows an example from `AXI_Master_casi.cpp`:

Example C-14 AXI master update()

```
void AXI_Master_casi::update ()
{
    if (mAxiState == STATE_B && mAxi.status[mAxi.nts-1] >= CASI_SLAVE_READY)
    {
        mAxiState = STATE_AR;
    }
    else if (mAxiState == STATE_R &&
             mAxi.status[mAxi.nts-1] >= CASI_SLAVE_READY)
    {
        mAxiState = STATE_AW;
    }
    mCycleCount++;
}
```

C.2.12 AXI_Slave_casi::communicate()

Inside the `communicate()` function, the AXI slave determines if a write response or read data is to be returned to the AXI master. Example C-15 shows the `communicate()` function from `AXI_Slave_casi.cpp`. The function uses local state control variables `mActiveWriteDoB` and `mActiveReadDoR` to indicate the action to take.

Example C-15 AXI slave communicate()

```
void  AXI_Slave_casi::communicate()
{
    if (mActiveWriteDoB)
    {
        if (mActiveWrite->cts < mActiveWrite->nts-1)
        {
            mActiveWrite->cts++;
```

```
            mActiveWrite->status[mActiveWrite->cts]     = CASI_MASTER_READY;
            mActiveWrite->masterFlags[AXI_MF_CHANNEL]   = AXI_CHANNEL_B;
            mActiveWrite->masterFlags[AXI_MF_RESPONSE]  = AXI_RESP_OKAY;
        }
        printf("AXI_Slave:  Cycle %4i: Channel B  driven\n", mCycleCount);
        mPort->getMaster()->getNotifyHandler()->notifyEvent(mActiveWrite);
    }
    if (mActiveReadDoR)
    {
        if (mActiveRead->status[mActiveRead->cts] != CASI_MASTER_READY)
        {
            uint32_t dataToRead;
            mActiveRead->cts++;
            mActiveRead->status[mActiveRead->cts]       = CASI_MASTER_READY;
            mActiveRead->masterFlags[AXI_MF_CHANNEL]    = AXI_CHANNEL_R;
            mActiveRead->masterFlags[AXI_MF_RESPONSE]   = AXI_RESP_OKAY;
            dataToRead = mActiveRead->cts-1;
            mActiveRead->dataRd[mActiveRead->cts-1] = dataToRead;
        }
        printf("AXI_Slave:  Cycle %4i: Channel R  driven   - data %x\n",
                mCycleCount, mActiveRead->dataRd[mActiveRead->cts-1]);

        mPort->getMaster()->getNotifyHandler()->notifyEvent(mActiveRead);
    }
}
```

C.2.13 AXI_Slave_casi::update()

The AXI slave uses `update()` to set the flags that indicate that the communicate() function must send a write response or read data to the AXI master. shows the `update()` function from `AXI_Slave_casi.cpp`. The function uses local state control variables `mActiveWriteDoB` and `mActiveReadDoR` to indicate that the B response channel or the R data channel is active.

Example C-16 AXI slave update()

```
void AXI_Slave_casi::update()
{
    if (mActiveWrite != NULL)
    {
        if (mActiveWrite->status[mActiveWrite->nts-1] >= CASI_SLAVE_READY)
        {
            mActiveWrite = NULL;
            mActiveWriteDoB = false;
        }
        else if ((mActiveWrite->cts == mActiveWrite->dataBeats) &&
```



```
                (mActiveWrite->status[mActiveWrite->cts] >= CASI_SLAVE_READY))
        {
            mActiveWriteDoB = true;
        }
    }
    if (mActiveRead != NULL)
    {
        if (mActiveRead->status[mActiveRead->nts-1] >= CASI_SLAVE_READY)
        {
            mActiveRead = NULL;
            mActiveReadDoR = false;
        }
        else
        {
            mActiveReadDoR = true;
        }
    }
    mCycleCount++;
}
```

C.2.14 Example AXI write transaction sequences

This section describes typical AXI write sequences.

Write with no wait states

Figure C-5 on page C-30 shows a UML diagram of a write transaction that:

- has no wait states
- uses separate cycles for the AW and W information transfer.

In `AXI_Master_casi::update()`, a write transaction is requested by initializing the transaction container and setting the master flag `mAxiState` to be `STATE_AW`. The actions that occur in each cycle following the requested write are:

1. `AXI_Master_casi::communicate()` is called:
   a. The value of `mAxiState` is tested and found to be `STATE_AW` and a write address transfer is initiated.
   b. The master port calls `AXI_Slave_casi::driveTransaction()` in the connected slave with the following values set in the transaction container:

```
mAxi.cts=0
mAxi.status[0]=CASI_MASTER_READY
mAxi.addr=0
mAxi.masterFlags[AXI_MF_CHANNEL]= AXI_CHANNEL_AW | AXI_TRANSFER_LAST
```

```
mAxi.dataBeats=1
mAxi.nts= 1 + mAxi.dataBeats + 1
mAxi.masterFlags[AXI_MF_BURST_TYPE]= AXI_BURST_FIXED
```

c. The slave acknowledges with ready by using its reference to the transaction container and changing the status:
`mActiveWrite->status[0]=CASI_SLAVE_READY`

d. The slave returns from the call to `driveTransaction()`.

e. The value of `mActiveWrite->status[AXI_STEP_ADDRESS]` is tested and found to be `CASI_SLAVE_READY` and the transfer for the AW channel is finished.

f. The state variable `mAxiState` is set to `STATE_W`.

g. In `update()`, the transaction remains unchanged, but both the master and slave test for any changes in the transaction container.

2. `AXI_Master_casi::communicate()` is called:

a. The value of `mAxiState` is tested and found to be `STATE_W` and a write data sequence is initiated.

b. The value of `mAxi.status[AXI_STEP_ADDRESS]` is tested and found to be `CASI_SLAVE_READY` and the transfer for the channel continues.

Note

If the slave is not ready, the transaction info is not be updated and another driveTransaction() is issued to test the ready status.

c. `mAxi.cts` is incremented and is now 1.

d. The master port calls `AXI_Slave_casi::driveTransaction()` of the connected slave with the following values set in the transaction container:

```
mAxi.status[mAxi.cts]=CASI_MASTER_READY
mAxi.masterFlags[AXI_MF_CHANNEL]= AXI_CHANNEL_W | AXI_TRANSFER_LAST
mAxi.dataWr=0
mAxi.masterFlags[AXI_MF_DATA_STROBE]= 0xF
mAxi.masterFlags[AXI_MF_DUSER] = 0
```

e. The slave acknowledges with ready by changing the transaction container
`mActiveWrite->status[mAxi.cts]=CASI_SLAVE_READY`

f. The slave returns from the call to `driveTransaction()` and the transfer for the W channel is finished.

Note

If a burst write had been requested instead of a single write, the write data sequence is repeated for each of the data beats.

g. In update(), the transaction remains unchanged, but both the master and slave test for any changes in the transaction container.

3. AXI_Master_casi::communicate() and AXI_Slave_casi:communicate() are called:

   a. In AXI_Master_casi::communicate(), the value of mAxiState is tested and found to be STATE_B and the response sequence active. The only action in the master is to call driveTransaction() with an empty transaction.

   b. In AXI_Slave_casi::communicate(), the slave determines that the last step of the write transaction has been reached.

   c. mAxi.cts is incremented and is now 2.

   d. The slave port calls AXI_Master_casi::notifyEvent() of the master with the following values set in its reference to the transaction container:

```
mActiveWrite->status[mActiveWrite->cts]=CASI_MASTER_READY
mActiveWrite->masterFlags[AXI_MF_CHANNEL]= AXI_CHANNEL_B
mActiveWrite->masterFlags[AXI_MF_RESPONSE] = AXI_RESP_OKAY
```

   e. The master acknowledges with ready by changing the transaction container

```
mAxi.status[mAxi.cts]=CASI_SLAVE_READY
```

   f. The slave returns from the call to notifyEvent() and the transfer for the B channel is finished.

   g. In AXI_Master_casi::update(), the transaction remains unchanged. If there is another request pending, the master mAxiState is set to indicate the next transaction type.

   h. In AXI_Slave_casi::update(), the transaction remains unchanged, but the mActiveWrite->status is found to be CASI_SLAVE_READY, so the mActiveWriteDoB flag for response is set to false.

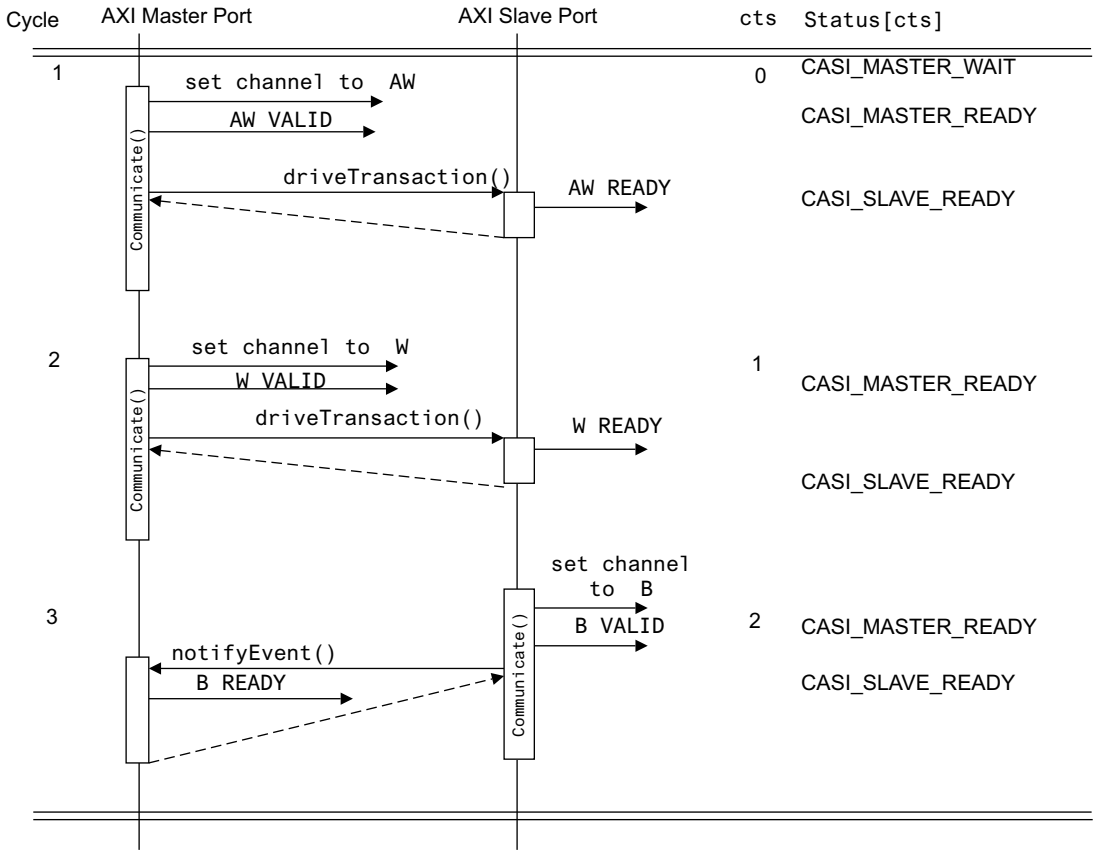

Figure C-5 Write with no wait states

Write with wait state in address channel

Figure C-6 shows a UML diagram of a write transaction that:

- has one wait states for the address transfer cycle
- uses separate cycles for the AW and W information transfer.

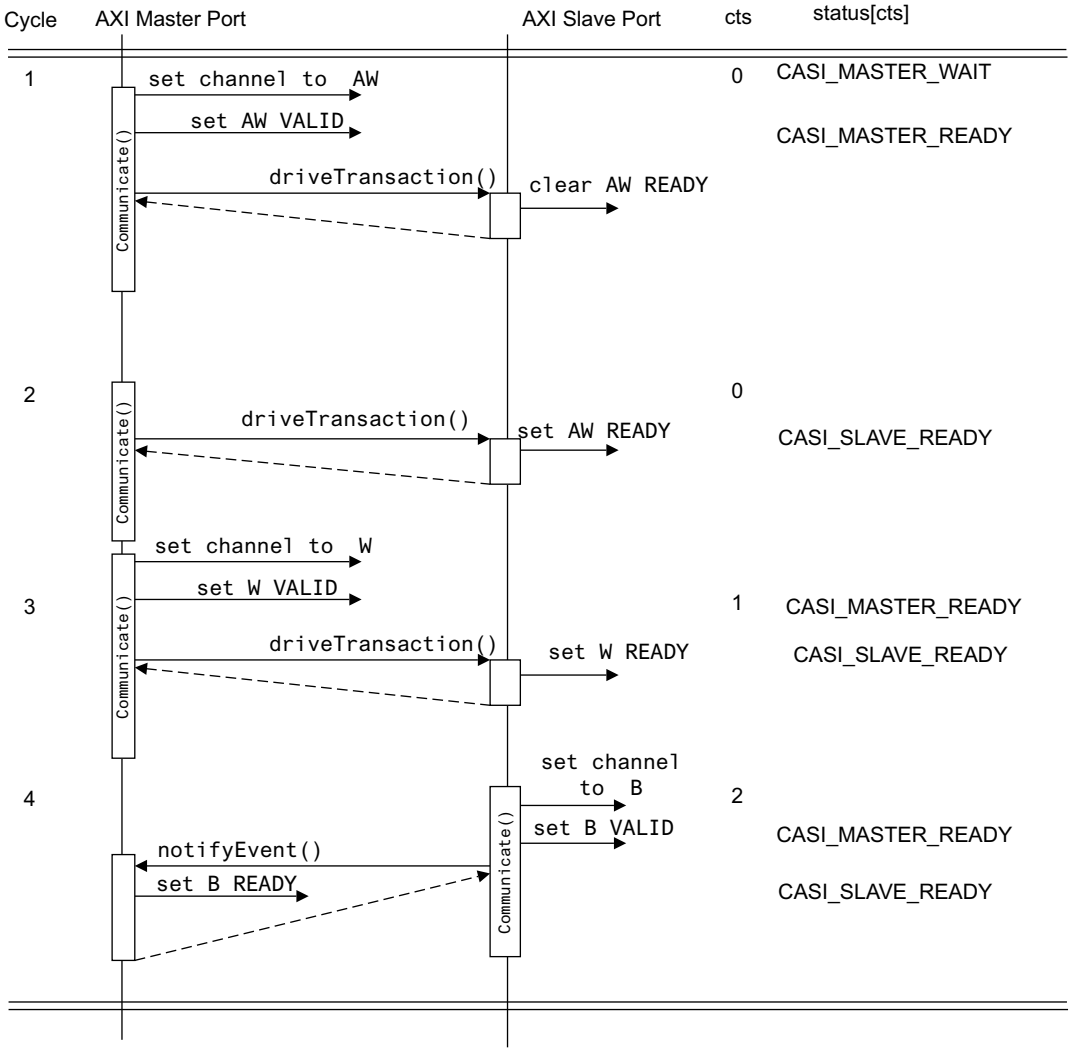

Figure C-6 Write with wait state in address channel

Write with combined AW and W channels

Figure C-7 shows a UML diagram of a write transaction that:

- has no wait states
- combines the AW and W information transfer in one cycle by setting CASI_MASTER_READY for both the address and write data steps.

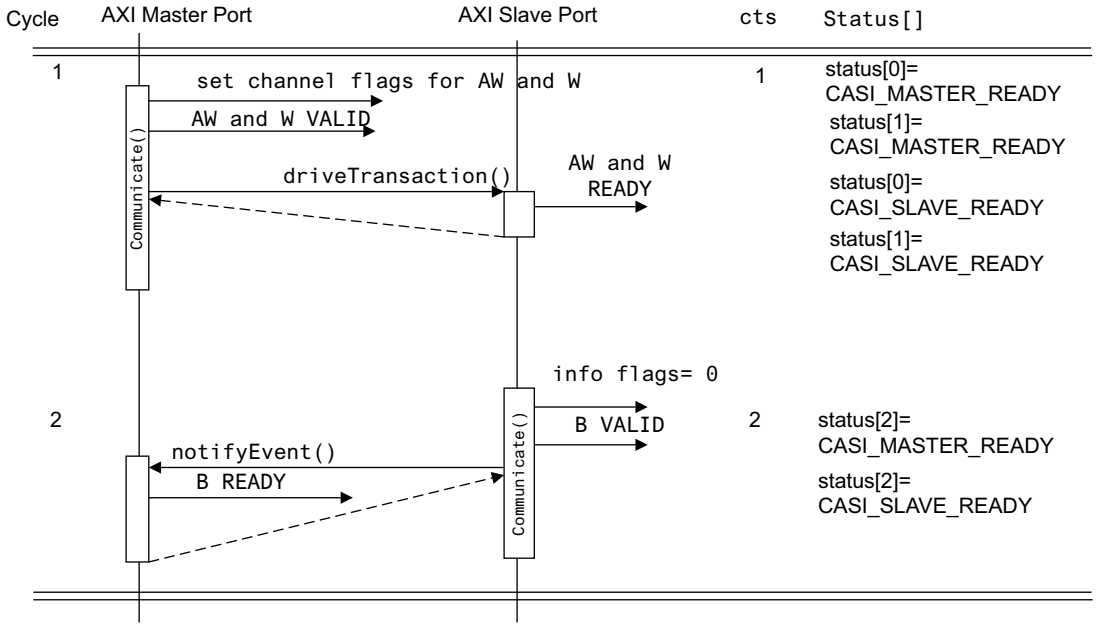

Figure C-7 Write with simultaneous AW and W, no wait states

Write with combined channels and two calls to driveTransaction()

Figure C-8 shows a UML diagram of a write transaction that:

- has no wait states
- combines the AW and W information transfer in one cycle calling `driveTransaction()` twice in the same cycle.

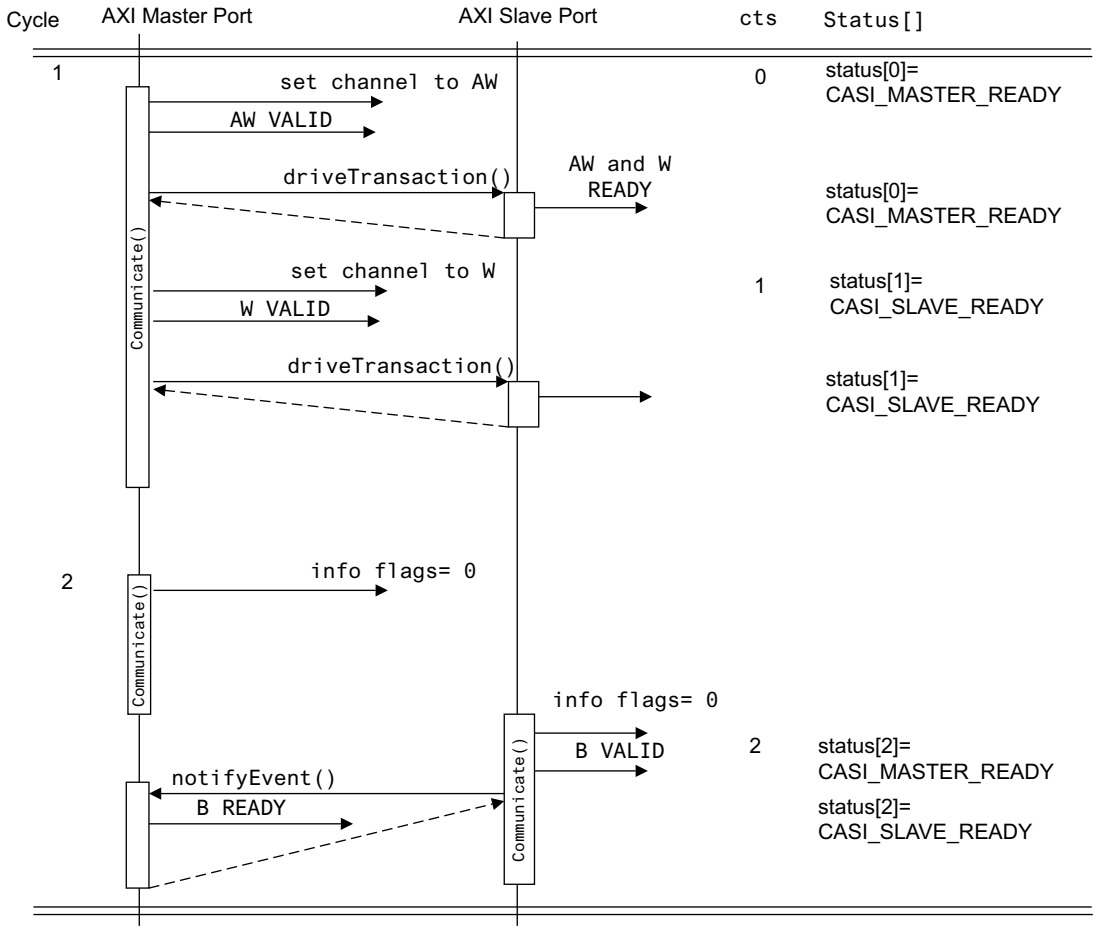

Figure C-8 Write with consecutive AW and W, no wait states

Write with combined AW and W channels and wait states

Figure C-9 shows a UML diagram of a write transaction that:

- has a wait state for the W data.
- combines the AW and W information.

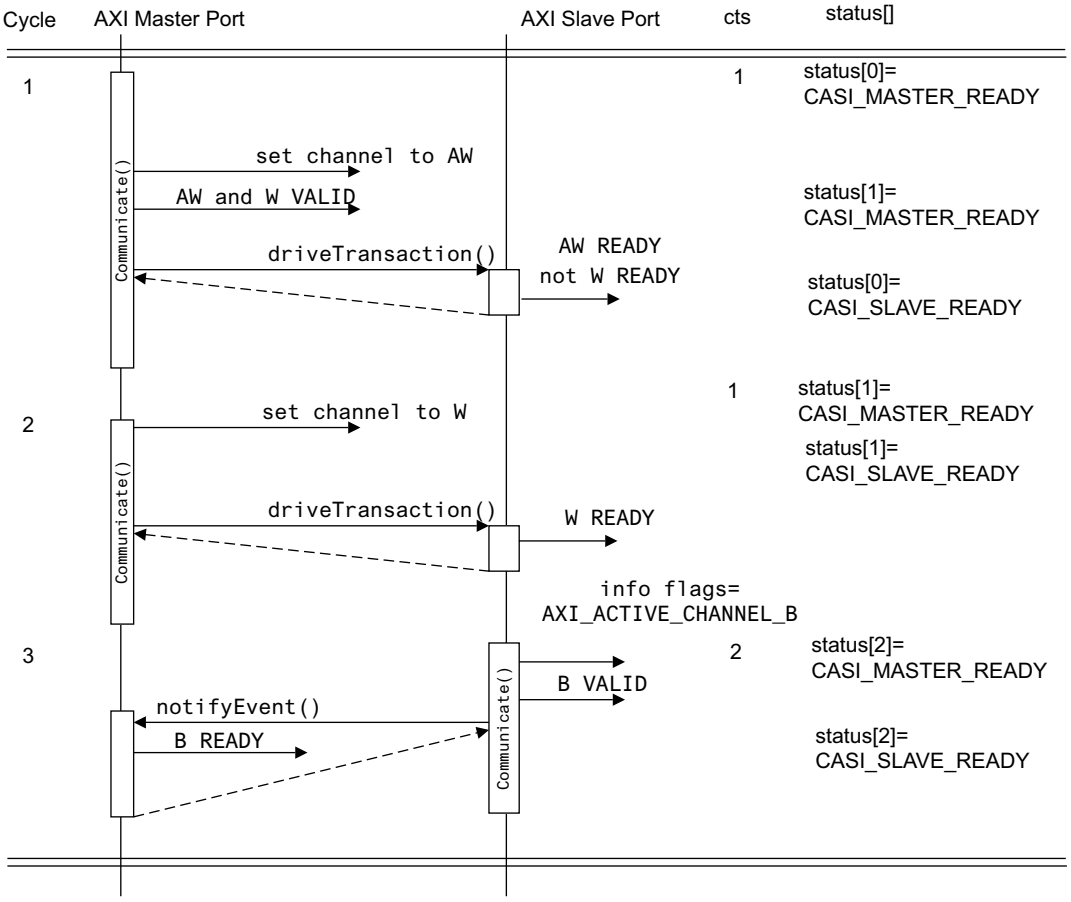

Figure C-9 Write with simultaneous AW and W and wait states

C.2.15 Example AXI read transaction sequences

This section describes typical AXI read sequences.

Read with no wait states

Figure C-10 on page C-36 shows a UML diagram of a read transaction that has no wait states on either the AR or R channels.

1. In communicate of cycle 1 the master port activates channel AR and calls `driveTransaction()` of the connected slave with transaction container fields:

   ```
   masterFlags[AXI_MF_CHANNEL]= AXI_CHANNEL_AR | AXI_TRANSFER_LAST
   cts=0
   status[cts]=CASI_MASTER_READY
   ```

2. The slave acknowledges with ready by modifying the transaction container:

   ```
   status[cts]=CASI_SLAVE_READY
   ```

3. The slave returns from the call and the AR transfer stage is finished. In `update()`, there are no changes to the transaction.

4. In communicate of cycle 2 the slave port activates channel R and calls `notifyEvent()` of the connected master with modified transaction container fields:

   ```
   masterFlags[AXI_MF_CHANNEL]= AXI_CHANNEL_R
   cts=1
   status[cts]=CASI_MASTER_READY
   ```

5. The master acknowledges with ready by modifying the transaction container:

   ```
   status[cts]=CASI_SLAVE_READY
   ```

6. The master returns from the call and the R transfer is finished.

   This also means that the entire transaction is finished. In `update()`, there are no changes to the transaction.

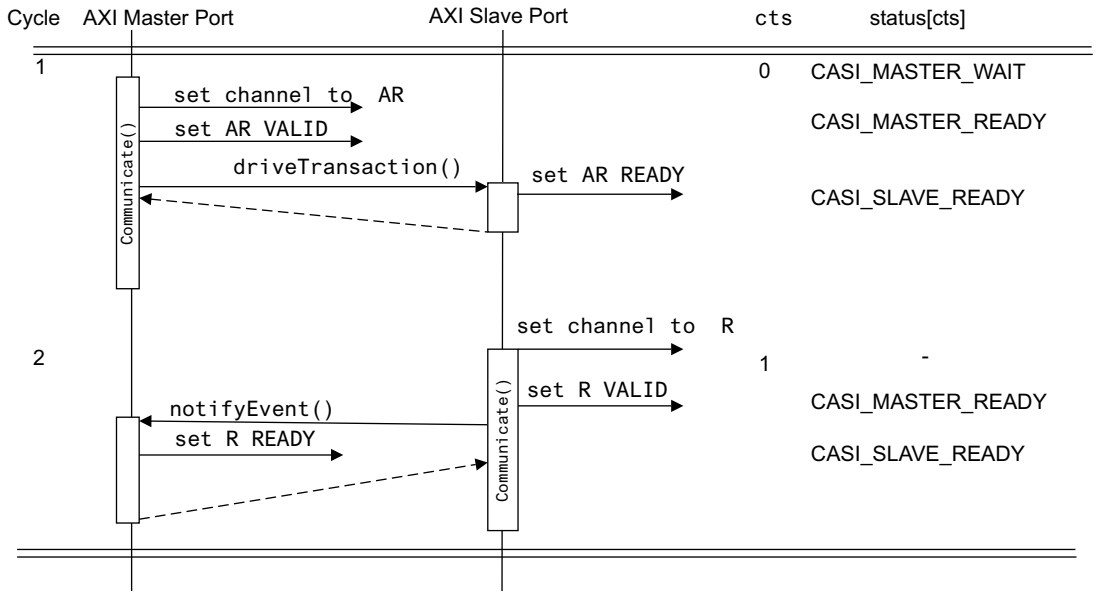

Figure C-10 Read with no wait states

Read with one wait state

Figure C-11 on page C-38 shows a UML diagram of a read transaction that has one wait states on the AR channel.

1. In communicate of cycle 1 the master port activates channel AR and calls `driveTransaction()` of the connected slave with transaction container fields:

   ```
   masterFlags[AXI_MF_CHANNEL]= AXI_CHANNEL_AR | AXI_TRANSFER_LAST
   cts=0
   status[cts]=CASI_MASTER_READY
   ```

2. The slave returns from the call but does not acknowledge ready by modifying the transaction container.

3. In communicate of cycle 2, the master port repeats the `driveTransaction()` call.

4. The slave acknowledges ready by changing the transaction container:

   ```
   status[cts]=CASI_MASTER_READY
   ```

5. The slave returns from the call and the AR transfer stage is finished. In `update()`, there are no changes to the transaction.

6. In cycle 2, the transaction does not become active for the R channel because the slave inserts a wait state. The slave does not call `notifyEvent()` in the master, but it does increment `cts`.

7. In communicate of cycle 4 the slave port activates channel R and calls `notifyEvent()` of the connected master with modified transaction container fields:

   ```
   masterFlags[AXI_MF_CHANNEL]= AXI_CHANNEL_R
   cts=1
   status[cts]=CASI_MASTER_READY
   ```

8. The master acknowledges with ready by modifying the transaction container:

   ```
   status[cts]=CASI_SLAVE_READY
   ```

9. The master returns from the call and the R transfer is finished.

   This also means that the entire transaction is finished. In `update()`, there are no changes to the transaction.

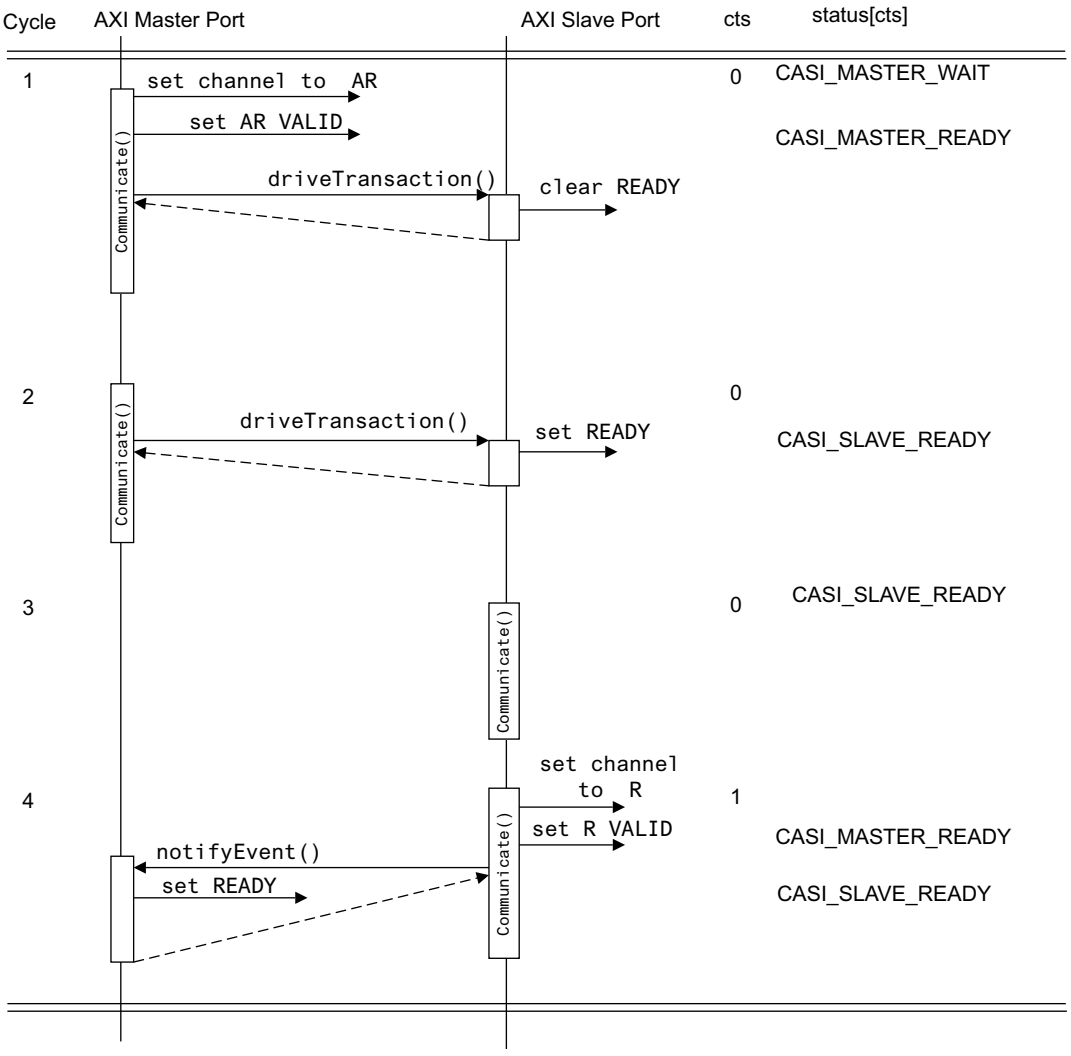

Figure C-11 Read with single wait state on address step

Read with wait states on both AR and R channels

Figure C-12 on page C-40 shows a UML diagram of a read transaction that wait states on both the AR and R channels.

1. In communicate of cycle 1 the master port activates channel AR and calls `driveTransaction()` of the connected slave with transaction container fields:

   ```
   masterFlags[AXI_MF_CHANNEL]= AXI_CHANNEL_AR | AXI_TRANSFER_LAST
   cts=0
   status[cts]=CASI_MASTER_READY
   ```

2. The slave returns from the call but does not acknowledge ready by modifying the transaction container.

3. In communicate of cycle 2, the master port repeats the `driveTransaction()` call.

4. The slave acknowledges ready by changing the transaction container:

   ```
   status[cts]=CASI_MASTER_READY
   ```

5. The slave returns from the call and the AR transfer stage is finished. In `update()`, there are no changes to the transaction.

6. In cycle 2, the transaction does not become active for the R channel because the slave inserts a wait state. The slave does not call `notifyEvent()` in the master, but it does increment `cts`.

7. In communicate of cycle 4 the slave port activates channel R and calls `notifyEvent()` of the connected master with modified transaction container fields:

   ```
   masterFlags[AXI_MF_CHANNEL]= AXI_CHANNEL_R
   cts=1
   status[cts]=CASI_MASTER_READY
   ```

8. In communicate of cycle 5, the slave port repeats the `notifyEvent()` call.

9. The master acknowledges with ready by modifying the transaction container:

   ```
   status[cts]=CASI_SLAVE_READY
   ```

10. The master returns from the call and the R transfer is finished.

    This also means that the entire transaction is finished. In `update()`, there are no changes to the transaction.

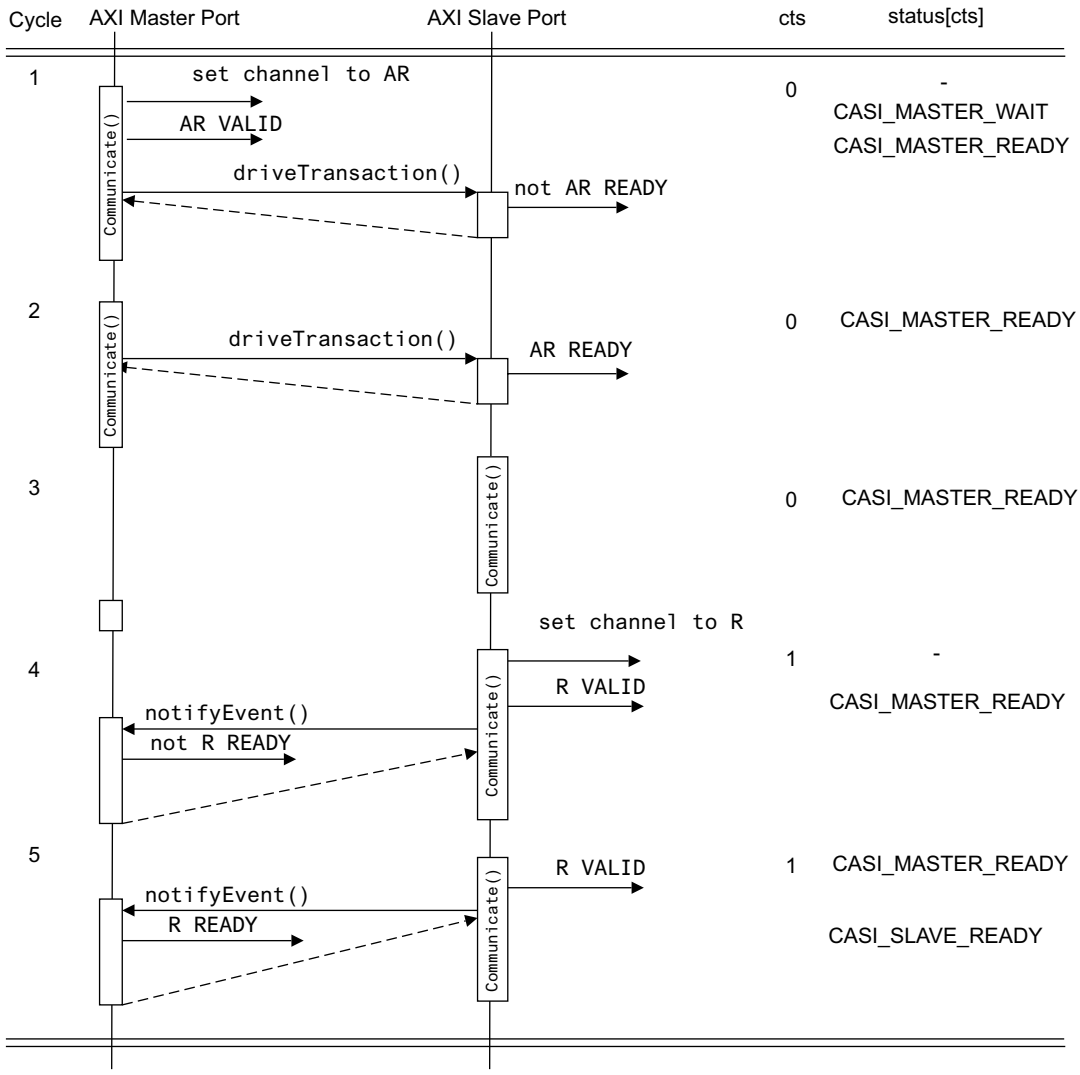

Figure C-12 Read with wait states on both address and data steps

C.2.16 Using the AXI Configurable Interconnect

AXI is bus specification that connects one master bus to one slave bus. Creating a bus that has multiple AXI masters or multiple slaves requires an *AXI Configurable Interconnect* (ACI) component that:

- provides one of its own AXI slaves for each of the external AXI masters
- provides one of its own AXI masters for each of the external AXI slaves
- manages arbitration requests from the different masters
- decodes the address from the master to select the appropriate slave port
- routes the data between the master that has control of the bus and the slave.

Hardware write access decoding is managed by combinatorial logic. The hardware logic determines the winning ACI slave port and the routing to the according ACI master port is established immediately. Within a single cycle, the initial winner can be displaced due to detecting the rising edge of **wvalid** from a second ACI slave port. The second winner can, in turn, be displaced by a third ACI slave port the winner can change again. For a hardware implementation, it is sufficient that all signals are stable by the end of the cycle.

In the CASI model of the ACI, it is not possible to defer the winner until the end of the cycle by waiting for **wvalid** signals. Every master port that has active transactions must issue requests to the corresponding slave port. All channels in the port (AW, AR and W) must send a request. If one channel in the port is not active, an empty request must be created and sent. The ACI counts the number of requests for the W channel and compares the count with the number of transactions registered internally for the w channel. When the counts are equal, the ACI can calculate the winner and continue accordingly.

A master without active transactions in any of the AW, AR, or W channels is not required to send requests to the slave. If at least one channel is active, however, the master must send empty requests for each of the inactive channels.

Note

As an alternative to sending empty requests, the master can set the `last` flag at the last normal request. Setting the flag is a shortcut and is recommended as it eliminates the requirement to send empty requests.

There is no arbitration for the R and B channels. Requests at the ACI master ports are forwarded immediately to the corresponding ACI slave port. A slave is only required to send normal requests for its active R or B channels. There is no requirement on the slave to send empty requests or set the `last` flag.

C.2.17 Example of ACI arbitration for the write channel

Figure C-13 on page C-43 shows the communication between two AXI masters (mp0 and mp1), the ACI, and an AXI slave port (sp0). In this example, mp1 has been granted access to sp0.

1. communicate() for Master 0 (mp0) is called first and sends a request for channel W to the connected ACI slave port a_sp0 by calling driveTransaction() of a_sp0.

   The ACI cannot determine at this stage whether mp0 is the winner since it is unknown whether Master 1 (mp0) will request channel W. The call returns with an unchanged transaction status.

2. Master 1 (mp1) is called next in communicate() and also sends a request for channel W to the connected ACI slave port a_sp1 by calling driveTransaction() of a_sp1.

3. Now the ACI can determine that the winner is slave port a_sp1. (The details of the arbitration mechanism are not considered for this example.) The request from a_sp1 is forwarded.

4. The ACI master port a_mp sends a request to the connected slave sp0 by calling driveTransaction() of sp0.

5. Slave sp0 acknowledges that it is ready by changing the transaction status and returning from the driveTransaction() call.

6. The ACI propagates the ready acknowledge to mp1 by changing the transaction status.

7. The ACI returns from the driveTransaction() call from mp1.

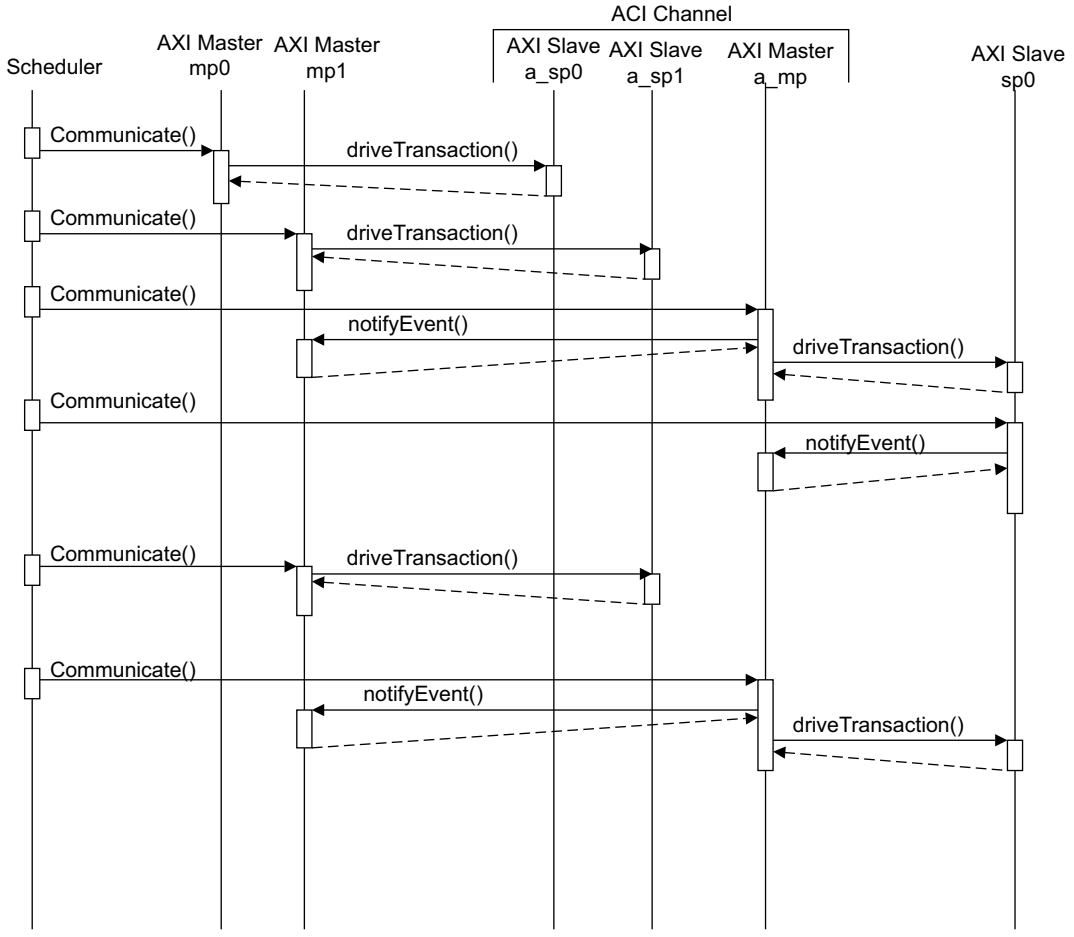

Figure C-13 Bus write with ACI

C.3 ESL API implementation of the AXI TLM

This section describes an example implementation that includes:

- a component with an AXI transaction master port
- a component with an AXI transaction slave port
- a system that connects the two components.

For more detail, see the example code supplied with the ESL API package.

The base classes for the AXI interface are described in *The transaction interface classes* on page 2-52.

C.3.1 Top-level system

The top-level system is implemented with the code in Example C-17:

Example C-17 Top.cpp

```
int sc_main (int argc , char *argv[])
{
    printf("\nNative CASI Master/Slave Example.\n\n");
    casi_clock_driver_root my_clk("MY_CLK", 1, SC_NS);

    //Instantiate the modules
    AXI_Master_casi *MasterComp = new AXI_Master_casi("Master");
    MasterComp->connect(&my_clk);
    AXI_Slave_casi *SlaveComp = new AXI_Slave_casi("Slave");
    SlaveComp->connect(&my_clk);
    MasterComp->init();  // call init() for master component
    SlaveComp->init();   // call init() for slave component

    //Connect the master port in MasterComp to the slave port in SlaveComp
    CONNECT_T(MasterComp->mPort, SlaveComp->mPort);
    MasterComp->interconnect();   // call interconnect() for master component
    SlaveComp->interconnect();    // call interconnect() for slave component

    MasterComp->reset(CASI_RESET_HARD,NULL);  //Call reset for master component
    SlaveComp->reset(CASI_RESET_HARD,NULL);   //Call reset for slave component

    sc_start(95);    // simulate for 95 cycles

    MasterComp->terminate();
    SlaveComp->terminate();
    return 0;
}
```

C.3.2 AXI_Master_casi component

The class definition for the sample component that contains an AXI transaction master, AXI_Master_casi, is listed in Example C-18:

Example C-18 AXI_Master_casi class definition

```
class AXI_Master_casi: public CASIModule
{
    friend class AXI_TM;

    public:
        SC_HAS_PROCESS(AXI_Master_casi);
        AXI_Master_casi(sc_module_name name);
        ~AXI_Master_casi();
         /* Functions for the different stages of simulation */
        void  init();
        void  interconnect();
        void  configure();
        void  reset(CASIResetLevel level, const CASIFileMapIF* filelist);
        void  terminate();
        /* Functions for Component parameters (user defined) */
        void  setParameter(const std::string& key, const std::string& value);
        std::string  getParameter    (const std::string& key);
        void  communicate(void);
        void  update(void);
        std::string         getName();
        CASIInterfaceType   getType);
        std::string         getProperty(CASIPropertyType property);
        CADI*               getCADI();
        CAPI*               getCAPI();
        // processNotify is called from the slave port
        void                processNotify(AXITransactionInfo* axi);
    public:
       // pointer to AXI transaction master port that belongs to this component
        AXI_TM*         mPort;

    private:
        // indicates which channel is in use
        enum AxiState { STATE_AW, STATE_W, STATE_B, STATE_AR, STATE_R };
        AxiState            mAxiState;

        // transaction structures used by port
        AXITransactionInfo  mAxi;
        AXITransactionInfo  mEmptyTransaction;
        uint32_t             mCycleCount;
};
```

C.3.3 AXI_Slave_casi component

The class definition for the sample component that contains an AXI transaction slave, `AXI_slave_casi`, is listed in Example C-19:

Example C-19 AXI_Slave_casi class

```
class AXI_Slave_casi: public CASIModule
{
    friend class AXI_TS;
    public:
        SC_HAS_PROCESS(AXI_Slave_casi);
        AXI_Slave_casi(sc_module_name name);
        ~AXI_Slave_casi();

        /* Functions for the different stages of simulation */
        void  configure();
        void  init();
        void  interconnect();
        void  reset(CASIResetLevel level, const CASIFileMapIF* filename);
        void  terminate();

        /* Functions for Component parameters (user defined) */
        void  setParameter(const std::string& key, const std::string& value);
        std::string  getParameter(const std::string& key);
        void  communicate(void);
        void  update(void);
        std::string         getName();
        CASIInterfaceType   getType();
        std::string         getProperty(CASIPropertyType property);
        CADI*    getCADI();
        CAPI*    getCAPI();

        //processDrive and processDriveDbg are called from the master port
        void        processDrive (AXITransactionInfo* axi);
        CASIStatus  processDriveDbg(AXITransactionInfo* axi);
    public:
       // pointer to AXI transaction slave port that belongs to this component
        AXI_TS*     mPort;
    private:
        AXITransactionInfo* mActiveWrite;  // structure used to write slave data
        bool                mActiveWriteDoB;  // current transaction is write
        AXITransactionInfo* mActiveRead;  // structure used to read slave data
        bool                mActiveReadDoR;   // current transaction is read
        uint32_t             mCycleCount;
};
```