# Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual

# Version 2.1

January 21st 2011

Notices

The information contained in this manual represents the definition of the SCE-MI as reviewed and released by Accellera in December 2010.

Accellera reserves the right to make changes to the SCE-MI and this manual in subsequent revisions and makes no warranties whatsoever with respect to the completeness, accuracy, or applicability of the information in this manual, when used for production design and/or development.

Accellera does not endorse any particular simulator or other CAE tool that is based on the SCE-MI.

Suggestions for improvements to the SCE-MI and/or to this manual are welcome. They should be sent to the SCE-MI email reflector or to the address below.

The current Working Group's website address is

http://www.accellera.org/apps/org/workgroup/itc

Information about Accellera and membership enrollment can be obtained by inquiring at the address below.

Published as: SCE-MI Reference Manual
Version 2.1 Release, December, 2010.

Published by: Accellera
1370 Trancas Street, #163
Napa, CA 94558
Phone: (707) 251-9977
Fax: (707) 251-9877

Printed in the United States of America.

# Contributions

The following individuals were major contributors to the creation of the original version of this standard: Duaine Pryor, Jason Andrews, Brian Bailey, John Stickley, Linda Prowse-Fossler, Gerard Mas, John Colley, Jan Johnson, and Andy Eliopoulos.

The following individuals contributed to the creation, editing and review of the SCE-MI Reference Manual Version 2.1. The companies associated with each individual are those that they were working for when the contribution was made.

| | | |
|---|---|---|
| Brian Bailey | Independent Consultant | ITC Workgroup Chair |
| Per Bojsen | AMD | |
| Pramod Chandraiah | Cadence | |
| Shabtay Matalon | Cadence | |
| Steve Seeley | Cadence | |
| John Stickley | Mentor Graphics | |
| Ying-Tsai Chang | Springsoft | |
| Amy Lim | Cadence | |
| Ajeya Prabhakar | Broadcom | |
| Ramesh Chandra | Qualcomm | |
| Russ Vreeland | Broadcom | |

The following individuals contributed to the creation, editing and review of the SCE-MI Reference Manual Version 2.0

| | | |
|---|---|---|
| Brian Bailey | Independent Consultant | ITC Workgroup Chair |
| Per Bojsen | AMD | |
| Shabtay Matalon | Cadence | |
| Duiane Pryor | Mentor Graphics | |
| John Stickley | Mentor Graphics | |
| Russell Vreeland | Broadcom | |
| Edmund Fong | AMD | |
| Jason Rothfuss | Cadence | |
| Bryan Sniderman | AMD | |

The following individuals contributed to the creation, editing, and review of SCE-MI Reference Manual Version 1.1.0

| | | |
|---|---|---|
| Jason Andrews | Axis | |
| Brian Bailey | Independent Consultant | ITC Workgroup Chair |
| Per Bojsen | Zaiq Technologies | |
| Dennis Brophy | Mentor Graphics | |
| Joseph Bulone | ST Microelectronics | |
| Andrea Castelnuovo | ST Microelectronics | |
| Fabrice Charpentier | ST Microelectronics | |

| | | |
|---|---|---|
| Damien Deneault | Zaiq Technologies | |
| Andy Eliopoulos | Cadence | |
| Vassilios Gerousis | Infineon | |
| Richard Hersemeule | ST Microelectronics | |
| Jan Johnson | Mentor Graphics | |
| Matt Kopser | Cadence | |
| Todd Massey | Verisity | |
| Shabtay Matalon | Cadence | |
| Richard Newell | Aptix | |
| Nish Parikh | Synopsys | |
| Duiane Pryor | Mentor Graphics | SCE-MI Subcommittee Chair |
| Joe Sestrich | Zaiq Technologies | |
| John Stickley | Mentor Graphics | |
| Russell Vreeland | Broadcom | |
| Irit Zilberberg | Cadence | |

or to prior versions of this standard.

## Revision history:

| | |
|---|---|
| Version 1.0 | 05/29/03 |
| Version 1.1 | 1/13/05 |
| Version 2.0 | 03/22/07 |
| Version 2.1 | 12/21/2010 |

# STATEMENT OF USE OF ACCELLERA STANDARDS

Accellera Standards documents are developed within Accellera and the Technical Committees of Accellera Organization, Inc. Accellera develops its standards through a consensus development process, approved by its members and board of directors, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of Accellera and serve without compensation. While Accellera administers the process and establishes rules to promote fairness in the consensus development process, Accellera does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an Accellera Standard is wholly voluntary. Accellera disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other Accellera Standard document.

Accellera does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or suitability for a specific purpose, or that the use of the material contained herein is free from patent infringement. Accellera Standards documents are supplied "AS IS."

The existence of an Accellera Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of an Accellera Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change due to developments in the state of the art and comments received from users of the standard. Every Accellera Standard is subjected to review periodically for revision and update. Users are cautioned to check to determine that they have the latest edition of any Accellera Standard.

In publishing and making this document available, Accellera is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is Accellera undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other Accellera Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of Accellera, Accellera will initiate action to prepare appropriate responses. Since Accellera Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, Accellera and the members of its Technical Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of Accellera Standards are welcome from any interested party, regardless of membership affiliation with Accellera. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Accellera Organization
1370 Trancas Street #163
Napa, CA 94558
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. Accellera shall not be responsible for identifying patents for which a license may be required by an Accellera standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Accellera is the sole entity that may authorize the use of Accellera-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use must be granted by Accellera Organization, Inc., provided that permission is obtained from and any required fee is paid to Accellera. To arrange for authorization please contact Lynn Horobin, Accellera, 1370 Trancas Street #163, Napa, CA 94558, phone (707) 251-9977, e-mail lynn@accellera.org. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained from Accellera.

# Table of Contents

# 1. Overview

While the 1.0 version of the Standard Co-Emulation API (see bibliography [B3]) was a technical success in every one of its original goals, it has not managed to attract a large community of independent model developers. This is deemed a necessary step for the broader acceptance of emulation and is thus a priority for this new version of the standard.

The broad attempt of the standard is to create a modeling interface that is as close as possible to that likely to be used for simulation, so that transactor models could be easily migrated from simulation to emulation as long as they adhere to a number of restrictions imposed by the additional demands of emulators.

The Verilog language was extended to create SystemVerilog (see Bibliography [B4]) and as part of this new standard a new interface was created call the Direct Programming Interface (DPI). This interface is intended to allow the efficient connection of an HDL model with a C model, a very similar task compared to one of the goals of this standard. We have thus tried to adopt the DPI interface for SystemVerilog wherever we can and to add additional capabilities to facilitate the efficient connection of the host based code with an emulator through additional mechanism such as pipes.

It is customary for Accellera standards to have been verified in practice before a standard is established. This ensures that the basis for the standard has been tested in the field on a number of real cases and is thus likely to be reasonably stable. This is not the case with this standard, since it contains material devised and created as part of the operations of this standards group. Even though the contributors to the standard have taken very effort to ensure its completeness, correctness and future stability, we cannot at this time guarantee that changes will not be necessary.

## 1.1 Scope

The scope of this document shall be restricted to what is specifically referred to herein as **the Standard Co-Emulation API: Modeling Interface** (SCE-MI).

## 1.2 Purpose

There is an urgent need for the EDA industry to meet the exploding verification requirements of SoC design teams. While the industry has delivered verification performance in the form of a variety of emulation and rapid prototyping platforms, there remains the problem of connecting them into SoC modeling environments while realizing their full performance potential. Existing standard verification interfaces were designed to meet the needs of design teams of over 10 years ago. A new type of interface is needed to meet the verification challenges of the next 10 years. This standard defines a multi-channel communication interface which addresses these challenges and caters to the needs of verification (both simulation and emulation) end-users and suppliers and providers of verification IP. In many places in this document it will make reference to emulation as this was the focus of the macro-based version of this specification. The new function-based and pipes capabilities added in this version of the specification are equally applicable to both simulation and emulation environments but this will not always be spelled out because of the awkwardness of it.

The SCE-MI can be used to solve the following verification customer problems.

- Most emulators on the market today offer some proprietary APIs in addition to SCE-MI 1.1 API. The proliferation of APIs makes it very difficult for software-based verification products to port to the different emulators, thus restricting the solutions available to customers. This also leads to low productivity and low return on investment (ROI) for emulator customers who build their own solutions.
- The emulation “APIs” which exist today are oriented to gate-level and not system-level verification.
- The industry needs an API which takes full advantage of emulation performance.

- This enables the portability of transactor models between emulation systems, making it possible for IP providers to write a single model. In addition, with the extension for the 2.0 version of this standard, it enables transactor models to be migrated from a simulation environment into an emulation environment so long as certain restrictions are adhered to. Models will also migrate in the other direction without any necessary changes.

The SCE-MI can also be used to solve the following verification supplier problems.

- Customers are reluctant to invest in building applications on proprietary APIs.
- Traditional simulator APIs like programmable language interface (PLI) and VHDL PLI slow down emulators.
- Third parties are reluctant to invest in building applications on proprietary APIs.
- The establishment of a common API that supports both simulators and emulators will encourage more third part model developers to make transactor available that are also suitable for emulators.

## 1.3 Usage

This specification describes a modeling interface which provides multiple channels of communication that allow software models describing system behavior to connect to structural models describing implementation of a device under test (DUT). Each communication channel is designed to transport un-timed messages of arbitrary abstraction between its two end points or "ports" of a channel.

These message channels are not meant to connect software models to each other, but rather to connect software proxy models to message port interfaces on the hardware side of the design. The means to interconnect software models to each other shall be provided by a software modeling and simulation environment, such as SystemC, which is beyond the scope of this document.

Although the software side of a system can be modeled at several different levels of abstraction, including un-timed, cycle-accurate, and even gate-level, the focus of SCE-MI Version 1 and 2 is to interface purely un-timed software models with a register transfer level- (RTL) or gate-level DUT.

This can be summarized with the following recommendations regarding the API.

Do not use it to bridge event-based or sub cycle-accurate simulation environments with the hardware side.

- It is possible, but not ideal, to use this to bridge cycle accurate simulation environments.
- It is best used for bridging an un-timed simulation environment with a cycle-accurate simulation environment.

Note: — There are many references in the document to SystemC (see Bibliography [2]) as the modeling environment for un-timed software models. This is because, although SystemC is capable of modeling at the cycle accurate RTL abstraction level, it is also considered ideally suited for un-timed modeling. As such, it has been chosen for use in many of the examples in this document. However it should not be inferred that the only possible environment that SCE-MI supports is SystemC and could equally be ANSI C, C++, or a number of other languages.

## 1.4 Performance goals

While software side of the described interface is generic in its ability to be used in any C/C++ modeling environment, it is designed to integrate easily with non-preemptive multi-threaded C/C++ modeling environments, such as SystemC. Similarly, its hardware side is optimized to prevent undue throttling of an emulator during a co-modeling session run.

Throughout this document the term emulation or emulator is used to denote a structural or RTL model of a DUT running in an emulator, rapid prototype, or other simulation environment, including software HDL simulators.

That said, however, the focus of the design of this interface is to avoid communication bottlenecks which might become most apparent when interfacing software models to an emulator as compared to interfacing them to a slower software HDL simulator or even an HDL accelerator. Such bottlenecks can severely compromise

the performance of an emulator, which is otherwise very fast. Although some implementations of the interface can be more inefficient than others, there shall be nothing in the specification of the interface itself that renders it inherently susceptible to such bottlenecks.

For this reason, the communication channels described herein are message- or transaction-oriented, rather than event-oriented, with the idea that a single message over a channel originating from a software model can trigger dozens to hundreds of clocked events in the hardware side of the channel. Similarly, it can take thousands of clocked events on the hardware side to generate the content of a message on a channel originating from the hardware which is ultimately destined for an un-timed software model.

## 1.5 Document conventions

This standard uses the following documentation notations.

- Any references to actual literal names that can be found in source code, identifiers that are part of the API, file names, and other literal names are represented in `courier` font.
- Key concept words or phrases are **in bold type**. See Chapter five for further definitions of these terms.
- Due to the draft nature of this document, informative and normative text is intermixed to allow easier understanding of the concepts. The normative text is shown in regular types, and the *informative is shown in blue italicized type.*

## 1.6 Contents of this standard

The organization of the remainder of this standard is:

- Chapter 2 provides references to other applicable standards that are assumed or required for this standard.
- Chapter 3 defines terms used throughout this standard.
- Chapter 4 provides an overall description and use models for the SCE Modeling Interface (SCE-MI).
- Chapter 5 is a formal functional specification of the APIs themselves.
- Appendix A is a tutorial showing the macro-based interface in a simple application.
- Appendix B provides an example using dynamic callbacks to implement a user-defined blocking send function on top of the non-blocking functions.
- Appendix C provides a VHDL package which can be used to supply SCE-MI macro component declarations to an application.
- Appendix D provides a simple multi-clock, multi-transactor schematic example and its VHDL code listing.
- Appendix E Using transaction pipes compatibly with OSCI-TLM applications
- Appendix F (Sample header files for the SCE-MI) provides headers for both C and C++ implementations.
- Appendix G Sample Header File for Basic Transaction Pipes C-Side API
- Appendix H Bibliography - provides additional documents, to which reference is made only for information or background purposes.

# 2. References

This standard shall be used in conjunction with the following publications. When any of the following standards is superseded by an approved revision, the revision shall apply.

[1] IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual.

[2] IEEE Std 1364-2005, IEEE Standard for Verilog Hardware Description Language.

[3] IEEE Std 1800-2005: IEEE Standard for SystemVerilog.

[4] IEEE Std 1666-2005: IEEE Standard for SystemC.

# 3. Definitions

For the purposes of this standard, the following terms and definitions apply. *The IEEE Standard Dictionary of Electrical and Electronics Terms* (see Bibliography [B1]) should be referenced for terms not defined in this standard.

## 3.1 Terminology

This section defines the terms used in this standard.

### 3.1.1 abstraction bridge:

A collection of abstraction gasket components that disguise a bus-cycle accurate, register transfer level, device under test (BCA RTL DUT) model as a purely untimed model. The idea is that to the untimed testbench models, the DUT itself appears untimed (see Figure 4.3) when, in fact, it is a disguised BCA model (see Figure 4.4).

### 3.1.2 abstraction gasket:

A special model that can change the level of abstraction of data flowing from its input to output and vice versa. For example, an abstraction gasket might convert an un-timed transaction to a series of cycle accurate events. Or, it might assemble a series of events into a single message. BCASH (bus-cycle accurate shell) models and transactors are examples of abstraction gaskets.

### 3.1.3 behavioral model:

See: untimed model.

### 3.1.4 bridge netlist:

The bridge netlist is the top level of the user-supplied netlist of components making up the hardware side of a co-modeling process. The components typically found instantiated immediately under the bridge netlist are transactors and the DUT. By convention, the top level netlist module the user supplies to the infrastructure linker is called Bridge and, for Verilog (see Reference [2])[1], is placed in a file called Bridge.v.

### 3.1.5 co-emulation:

A shorthand notation for co-emulation modeling, also known as co-modeling. See also: co-modeling.

### 3.1.6 co-modeling:

Although it has broader meanings outside this document, here co-modeling specifically refers to transaction-oriented co-modeling in contrast to a broader definition of co-modeling which might include event-oriented co-modeling. Also known as co-emulation modeling, transaction-oriented co-modeling describes the process of modeling and simulating a mixture of software models represented with an un-timed level of abstraction, simultaneously executing and inter-communicating through an abstraction bridge, with hardware models represented with the RTL level of abstraction, and running on an emulator or a simulator. Figure 3.1 depicts such a configuration, where the Standard Co-Emulation API - Modeling Interface (SCE-MI) is being used as the abstraction bridge. See section 3.2 for the definitions of the acronyms used here.

[1] For more information on references, see Chapter 2.

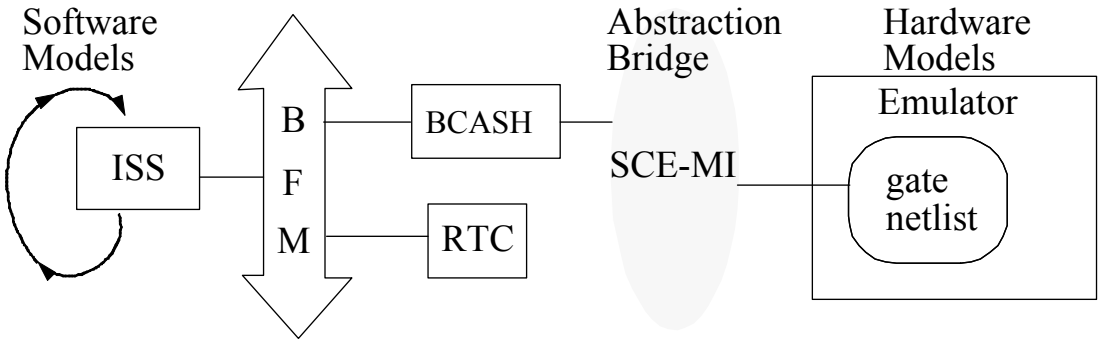



Figure 3.1 Using the SCE-MI as an abstraction bridge

Another illustration can be seen in Figure 4.2.

### 3.1.7 controlled clock (cclock):

A clock defined in the macro-based interface that drives the DUT and can be disabled by any transactor during operations which would result in erroneous operation of the DUT when it is clocked. When performing such operations, any transactor can "freeze" controlled time long enough to complete the operation before allowing clocking of the DUT to resume. The term cclock is often used throughout this document as a synonym for controlled clock.

### 3.1.8 controlled time:

Time defined in the macro-based interface which is advanced by the controlled clock and frozen when the controlled clock is suspended by one or more transactors. Operations occurring in uncontrolled time, while controlled time is frozen, appear between controlled clock cycles.

### 3.1.9 co-simulation:

The execution of software models modeled with different levels of abstraction that interact with each other through abstraction gaskets similar to BCASH (bus-cycle accurate shell) models. Figure 3.2 illustrates such a configuration. (See section 3.2 for definitions of the acronyms used here.)

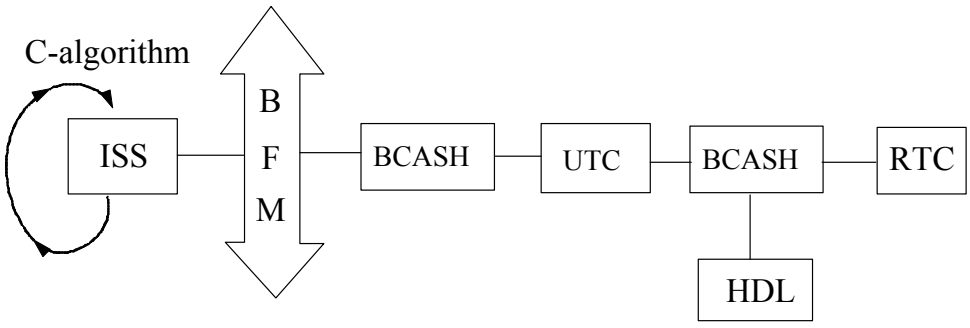



Figure 3.2 Modeling abstraction gaskets

The key difference between co-simulation and co-emulation is the former typically couples software models to a traditional HDL simulator interface through a proprietary API, whereas the latter couples software models to an emulator through an optimized transaction oriented interface, such as SCE-MI.

### 3.1.10 cycle stamping:

A process defined in the macro-based interface where messages are tagged with the number of elapsed counts of the fastest controlled clock in the hardware side of a co-modeled design.

### 3.1.11 don’t care duty cycle:

A posedge active don’t care duty cycle is a way of specifying a duty cycle in the macro-based interface where the user only cares about the posedge of the clock and does not care about where in the period the negedge falls, particularly in relation to other cclocks in a functional simulation. In such a case, the DutyHi parameter is given as a 0. The DutyLo can be given as an arbitrary number of units which represent the whole period such that the Phase offset can still be expressed as a percentage of the period (i.e., DutyHi+DutyLo). See 5.2.4.1 for more details. A negedge active don’t care duty cycle is a way of specifying a duty cycle in the macro-based interface where the user only cares about the negedge of the clock and does not care about where in the period the posedge falls, particularly in relation to other cclocks in a functional simulation. In such a case, the DutyLo parameter is given as a 0. The DutyHi can be given as an arbitrary number of units that represent the whole period such that the Phase offset can still be expressed as a percentage of the period (i.e., DutyHi+DutyLo). See 5.2.4.1 for more details.

### 3.1.12 device or design under test (DUT):

A device or design under test that can be modeled in hardware and stimulated and responded to by a software testbench through an abstraction bridge such as the SCE-MI shown in Figure 4.2.

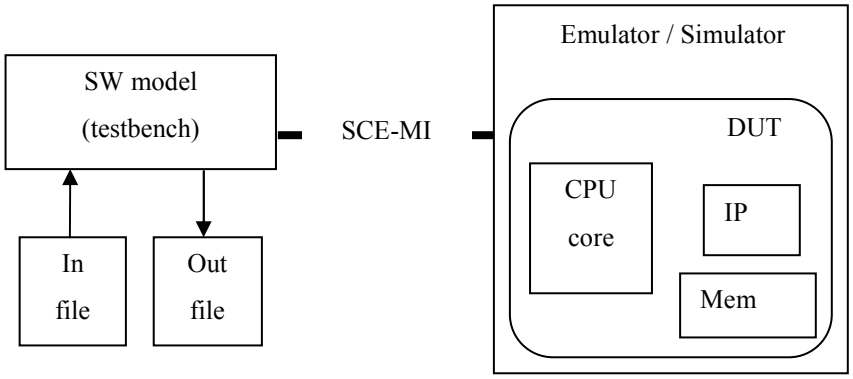



Figure 3.3 Modeling a DUT via an abstraction bridge

### 3.1.13 DUT proxy:

A model or collection of models that presents (to the rest of the system) an interface to the design under test which is un-timed. This is accomplished by a translation of un-timed messages to cycle-accurate pin activity. A DUT proxy contains one or more abstraction bridges which perform this function. If the abstraction bridge is SCE-MI, the un-timed communication is handled by message port proxy interfaces to the message channels. See Figure 4.4 for an illustration of DUT proxies.

### 3.1.14 Fastest Clock:

If the user instantiates a 1/1 cclock without a don't care duty cycle in the macro-based interface, then that becomes the fastest clock in the system, although it limits performance to be only half as fast as the uclock, since in this case, both edges must be scheduled on posedges of uclock.

### 3.1.15 hardware model:

A model of a block that has a structural representation (i.e., as a result of synthesis or a gate netlist generated by an appropriate tool) which is mapped onto the hardware side of a co-modeling process (i.e., an emulator, other hardware simulation platform or a simulator). It can also be real silicon (i.e., a CPU core or memory chip) plugged into an emulator or simulation accelerator.

### 3.1.16 hardware side:

See: software side.

### 3.1.17 infrastructure linkage process:

The process defined in the macro-based interface that reads a user description of the hardware, namely the source or bridge netlist describing the interconnect between the transactors, the DUT, and the SCE-MI interface components, and compiles that netlist into a form suitable for executing in a co-modeling session. Part of this compile process can include adding more structure to the bridge netlist it properly interfaces the user-supplied netlist to the SCE-MI infrastructure implementation components.

### 3.1.18 macros:

These are implementation components provided by a hardware emulator solution to implement the hardware side of the SCE-MI infrastructure in the macro-based interface, examples include: SceMiMessageInPort, SceMiMessageOutPort, SceMiClockControl, and SceMiClockPort.

### 3.1.19 message:

A data unit of arbitrary size and abstraction to be transported over a channel. Messages are generally not associated with specific clocked events, but can trigger or result from many clocks of event activity. For the most part, the term message can be used interchangeably with transaction. However, in some contexts, transaction could be thought of as including infrastructure overhead content in addition to user payload data (and handled at a lower layer of the interface), whereas the term message denotes only user payload data.

### 3.1.20 message channel:

A two-ended conduit of messages between the software and hardware sides of an abstraction bridge.

### 3.1.21 message port:

The hardware side of a message channel. Transactors use these ports to gain access to messages being sent across the channel to or from the software side.

### 3.1.22 message port proxy:

The software side of a message channel. DUT proxies or other software models use these proxies to gain access to messages being sent across the channel to or from the hardware side.

### 3.1.23 negedge:

This refers to the falling edge of a clock in the macro-based interface.

### 3.1.24 posedge:

This refers to the rising edge of a clock in the macro-based interface.

### 3.1.25 service loop:

This function or method call in the macro-based interface allows a set of software models running on a host workstation to yield access to the SCE-MI software side so any pending input or output messages on the channels can be serviced. The software needs to frequently call this throughout the co-modeling session in order to avoid backup of messages and minimize the possibility of system deadlock. In multi-threaded environments, place the service loop call in its own continually running thread. See 5.4.3.7 for more details.

### 3.1.26 software model:

A model of a block (hardware or software) that is simulated on the software side of a co- modeling session (i.e., the host workstation). Such a model can be an algorithm (C or C++) running on an ISS, a hardware model that is modeled using an appropriate language environment, such as SystemC, or an HDL simulator.

### 3.1.27 software side:

This term refers to the portion of a user's design which, during a co-modeling session, runs on the host workstation, as opposed to the portion running on the emulator (which is referred to as the hardware side). The SCE-MI infrastructure itself is also considered to have software side and hardware side components.

### 3.1.28 structural model:

A netlist of hardware models or other structural models. Because this definition is recursive, by inference, structural models have hierarchy.

### 3.1.29 transaction:

See: message.

### 3.1.30 transactor:

A form of an abstraction gasket. A transactor decomposes an un-timed transaction to a series of cycle-accurate clocked events, or, conversely, composes a series of clocked events into a single message.

### 3.1.31 uncontrolled clock (uclock):

A free-running system clock defined in the macro-based interface, generated internally by the SCE-MI infrastructure, which is used only within transactor modules to advance states in uncontrolled time. The term uclock is often used throughout this document as a synonym for uncontrolled clock.

### 3.1.32 uncontrolled reset:

This is the system reset defined in the macro-based interface, generated internally by the SCE-MI infrastructure, which is used only with transactor modules. This signal is high at the beginning of simulated time and transitions to low an arbitrary (implementation-dependent) number of uclocks later. It can be used to reset a transactor. The controlled reset is generated exactly once by the SCE-MI hardware side infrastructure at the very beginning of a co- modeling session.

### 3.1.33 uncontrolled time:

Time defined in the macro-based interface that is advanced by the uncontrolled clock, even when the controlled clock is suspended (and controlled time is frozen).

### 3.1.34 untimed model:

A block that is modeled algorithmically at the functional level and exchanges data with other models in the form of messages. An un-timed model has no notion of a clock. Rather, its operation is triggered by arriving messages and it can, in turn, trigger operations in other un-timed models by sending messages.

## 3.2 Acronyms and abbreviations

This section lists the acronyms and abbreviations used in this standard.

| | |
|---|---|
| API | Application Programming Interface |
| BCA | Bus-Cycle Accurate model - sometimes used interchangeably with RTL model |
| BCASH | Bus-Cycle Accurate SHell model |
| BFM | Bus Functional Model |
| BNF | extended Backus-Naur Form |
| DPI | SystemVerilog Direct Programming Interface |
| DUT | Device or Design Under Test |
| EDA | Electronic Design Automation |
| HDL | Hardware Description Language |
| HVL | Hardware Verification Language<br>High-level Verification Language |
| IP | Intellectual Property |

| | |
|---|---|
| ISS | Instruction Set Simulator |
| PLI | Programmable Language Interface |
| RTC | Register Transfer Level C model |
| RTL | Register Transfer Level |
| SCE-API | Standard Co-Emulation API |
| SCE-MI | Standard Co-Emulation API - Modeling Interface |
| UT or UTC | Untimed C model |
| VHDL | VHSIC Hardware Description Language |

# 4. Use models

SCE-MI directly supports three primary use models for connecting a model written in HDL to a model running on a workstation. Each of these use-models is enabled by a corresponding interface. The software side of the interface allows access from the workstation side, while the hardware side of the interface allows access from the HDL side. The three interfaces are a (message-passing) macro-based interface which was standardized in the previous SCE-MI 1.1 version of this standard and has been updated and extended in the SCE-MI 2 release, secondly a new function-based interface based on the SystemVerilog DPI (see section 4.7) and thirdly, a new pipes-based interface (see section 4.8). These are shown in Figure 4.1.

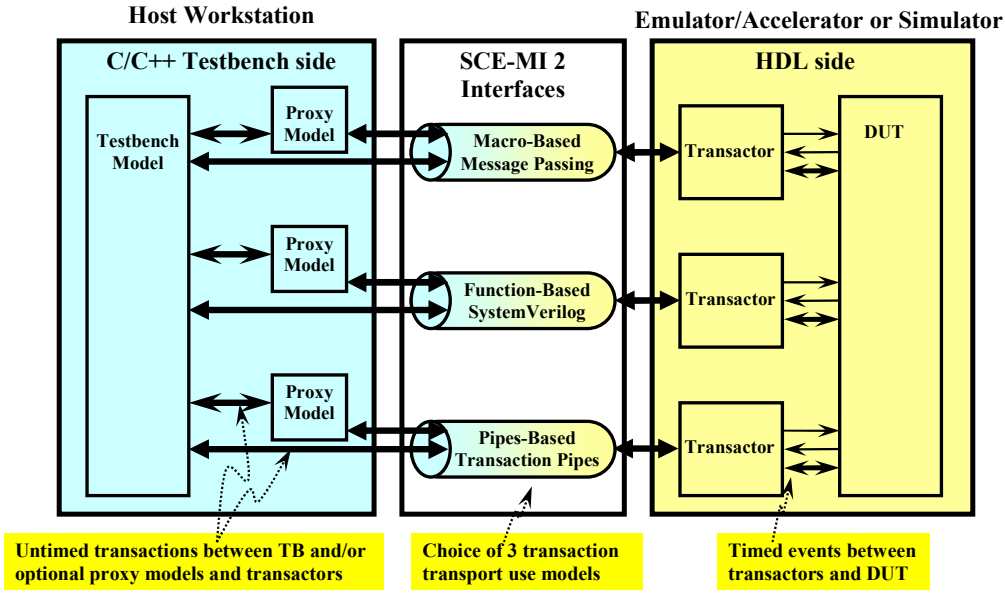



Figure 4.1 Three Interfaces

Each of these three interfaces are self contained such that a model that utilizes any one of them will be able to communicate with a model on the other side of an interface so long as the implementation of the interface also supports that interface. Models that use multiple interfaces will require complete support for each interface plus any additional requirements brought about by the interface compatibility issues.

A significant difference between the function-based interface and the two others is that an implementation running on a simulator (or on an emulator) that supports a larger subset of DPI than the one defined by SCE-MI 2 function-based interface is still compatible with the function-based interface standard. In addition, the function-based implementation on a simulator is not required to flag errors as long as it is compliant with DPI as defined by the SystemVerilog LRM. This is contrary to the macro-based and pipe-based interfaces when all implementations must support the same set of features define by these interfaces on any implementation. For SCE-MI 2 a subset of the DPI was chosen that is more easily synthesized to support current emulation technology. As time goes on, it may become feasible for emulators to support a broader subset of the DPI and the standard may be expanded. The standard thus defines the minimum sub-set of DPI features that should be supported by the SCE-MI 2 function-based interface to be deemed SCE-MI 2 compliant, but each implementation is free to add additional features of the SystemVerilog DPI standard to their function-based interface implementation.

However, this impacts what it means for a model to be portable and SCE-MI 2 function-based compliant. To be portable, the model can only utilize the function-based DPI features defined in this standard. Using any additional features assumed to be available by a specific implementation make it non-portable to other

implementations. It also impacts model compliance with SCE-MI 2 function-based interface as a model that uses the additional DPI features makes it SCE-MI 2 function-based non-compliant model.

Each of the three interfaces constitutes a corresponding use model carrying the same name described in the subsequent sections of the SCE-MI 2 specification.

## 4.1 Macro-based Message Passing Interface

This section of the document will describe the macro-based message passing environment. The description of the function call mechanism will be given in section 4.7 and the pipes-based interface in section 4.8. This message passing interface is intended to be used in several different use models and by several different groups of users.

### 4.1.1 High-level description

Figure 4.2 shows a high-level view of how SCE-MI interconnects untimed software models to structural hardware transactor and DUT models.

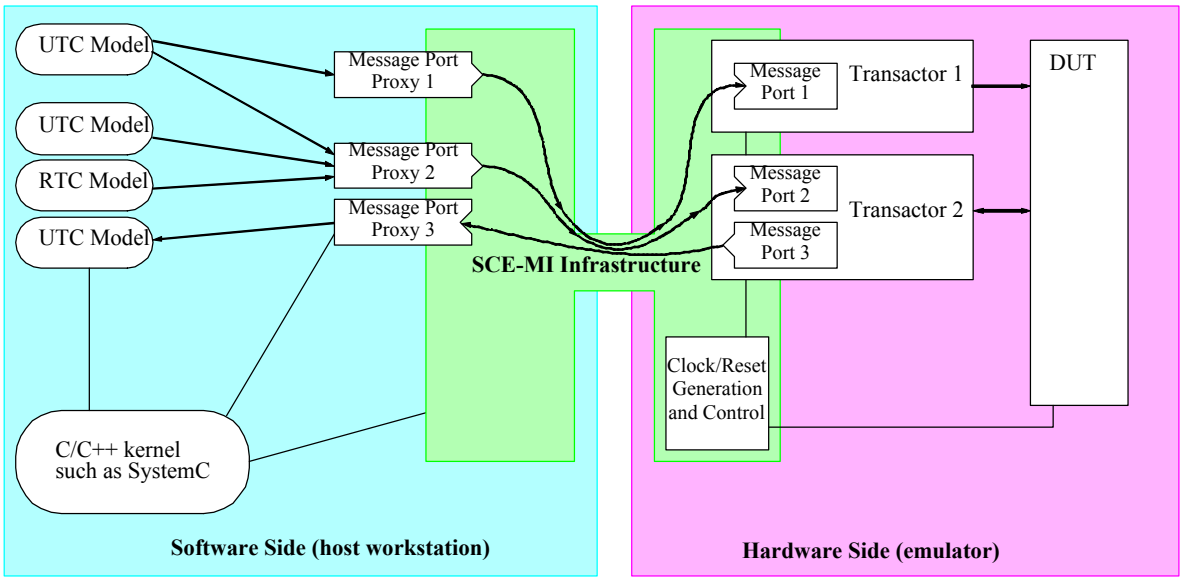



Figure 4.2 High-level view of run-time components

The SCE-MI provides a transport infrastructure between the emulator and host workstation sides of each channel, which interconnects *transactor* models in the emulator to C (untimed or RTL) models on the workstation. For purposes of this document, the term *emulator* can be used interchangeably with any simulator capable of executing RTL or gate-level models, including software HDL simulators.

These interconnects are provided in the form of message channels that run between the *software side* and the *hardware side* of the SCE-MI infrastructure. Each message channel has two ends. The end on the software side is called a *message port proxy*, which is a C++ object or C function that gives API access to the channel. The end on the hardware side is a *message port*, which is instantiated inside a transactor and connected to other components in the transactor. Each message channel is either an input or an output channel with respect to the hardware side.

Note: While all exposition in this standard is initially given using C++, C equivalents exist for all functionality. See Chapter 5 for more details.

*Message channels* are not unidirectional or bidirectional busses in the sense of hardware signals, but are more like network sockets that use message passing protocols. It is the job of the transactors to serve as *abstraction*

*gaskets* and decompose messages arriving on input channels from the software side into sequences of cycle-accurate events which can be clocked into the DUT. For the other direction of flow, transactors recompose sequences of events coming from the DUT back into messages to be sent via output channels to the software side.

In addition, the SCE-MI infrastructure provides clock (and reset) generation and shared clock control using handshake signals with the transactor in the macro-based use model. This allows the transactor to "freeze" *controlled time* while performing message composition and decomposition operations.

## 4.2 Support for environments

The SCE-MI provides support for both single and multi-threaded environments.

### 4.2.1 Multi-threaded environments

The SCE-MI is designed to couple easily with multi-threaded environments, such as SystemC, yet it also functions just as easily in single-threaded environments, such as simple C programs. SCE-MI macro-based interface provides a special *service loop* function (see 5.4.3.7), which can be called from an application to give the SCE-MI infrastructure an opportunity to service its communication channels. Calls to service loop result in the sending of queued input messages to hardware and the dispatch of arriving output messages to the software models.

While there is no thread-specific code inside the service loop function (or elsewhere in the SCE-MI), this function is designed to be called periodically from a dedicated thread within a multi-threaded environment, so the interface is automatically serviced while other threads are running.

When only using the function-based or pipes-based use model, calling the service loop is not required.

### 4.2.2 Single-threaded environments

In a single-threaded environment, calls to the service loop function in the macro-based use model can be "sprinkled" throughout the application code at strategically placed points to frequently yield control of the CPU to the SCE-MI infrastructure so it can service its messages channels.

## 4.3 Users of the interface

A major goal of this specification is to address the needs of three target audiences, each with a distinct interest in using the interface. The target audiences are:

- end-user
- transactor implementor
- SCE-MI infrastructure implementor

### 4.3.1 End-user

The *end-user* is interested in quickly and easily establishing a bridge between a software testbench which can be composed of high-level, *untimed*, algorithmic software models, and a hardware DUT which can be modeled at the RTL, cycle-accurate level of abstraction.

While end-users might be aware of the need for a "gasket" that bridges these two levels of abstraction, they want the creation of these *abstraction bridges* to be as painless and automated as possible. Ideally, the end-users are not required to be familiar with the details of SCE-MI API. Rather, on the *hardware side*, they might wish to rely on the *transactor implementer* (see 4.3.2) to provide predefined *transactor* models which can directly interface to their DUT. This removes any requirement for them to be familiar with any of the SCE-MI hardware-side interface definitions. Similarly, on the *software side*, the end-users can also rely on the transactor implementers to furnish them with *plug-and-play* software models, custom-tailored for a software modeling environment, such as SystemC. Such models can encapsulate the details of interfacing to the SCE-MI software side and present a fully *untimed*, easy- to-use interface to the rest of the software testbench.

### 4.3.2 Transactor implementer

The transactor implementer is familiar with the SCE-MI, but is not concerned with its implementation. The transactor implementer provides plug-and-play transactor models on the *hardware side* and proxy models on the *software side* which *end-users* can use to easily bridge their untimed software models with their RTL-represented DUT. Additionally, the transactor implementer can supply *proxy models* on the software side which provide untimed "sockets" to the transactors.

Using the models is like using any other stand-alone IP models and the details of bridging not only two different abstraction levels, but possibly two different verification platforms (such as SystemC and an emulator), is completely hidden within the implementations of the models which need to be distributed with the appropriate object code, netlists, RTL code, configuration files, and documentation.

### 4.3.3 SCE-MI infrastructure implementor

The SCE-MI infrastructure implementer is interested in furnishing a working implementation of an SCE-MI that runs on some verification platform, including both the *software side* and the *hardware side* components of the SCE-MI. For such a release to be complaint, it needs to conform to all the requirements set forth in this specification.

## 4.4 Bridging levels of modeling abstraction

The central goal of this specification is to provide an interface designed to bridge two modeling environments, each of which supports a different level of modeling abstraction.

### 4.4.1 Untimed software level modeling abstraction

Imagine a testbench consisting of several, possibly independent models that stimulate and respond to a DUT at different interface points (as depicted in Figure 4.3). This configuration can be used to test a processor DUT which has some communications interfaces that can include an Ethernet adapter, a PCI interface, and a USB interface. The testbench can consist of several models that independently interact with these interfaces, playing their protocols and exchanging packets with them. These packets can be recoded as *messages* with the intent of verifying the processor DUT's ability to deal with them. Initially, the system shown in Figure 4.3 might be implemented fully at the *untimed* level of abstraction by using the SystemC software modeling environment.

Suppose the ultimate desire here is to create a cycle-accurate RTL model of a design and eventually synthesize this model to gates that can be verified on a high speed emulation platform. Afterwards, however, they might also be tested with the unaltered, untimed testbench models. To do all of this requires a way of somehow bridging the untimed level of abstraction to the *bus-cycle accurate* (*BCA)* level.

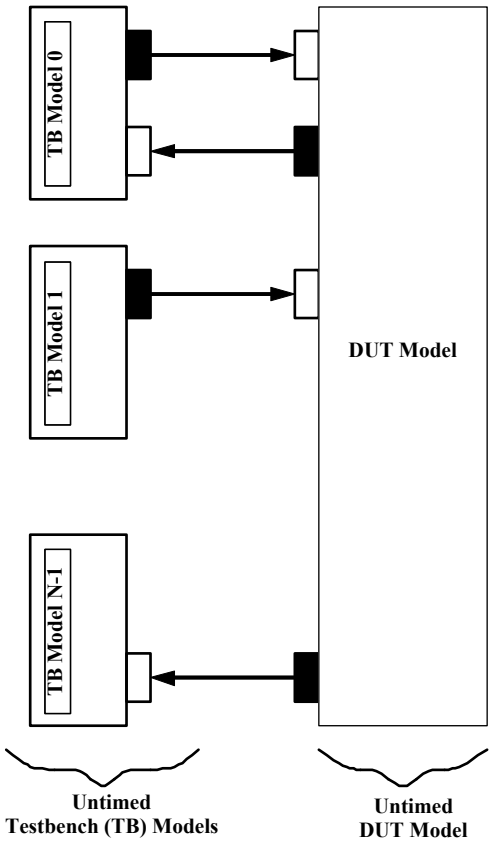



Figure 4.3 Untimed software testbench and DUT models

### 4.4.2 Cycle-accurate hardware level modeling abstraction

Take the purely untimed system shown in Figure 4.3, "pry apart" the direct coupling between the testbench models and the untimed DUT model, and insert an *abstraction bridge* from the still untimed system testbench model to what is now a emulator resident, RTL-represented DUT. This bridge consists of a set of *DUT proxy* models, SCE-MI *message input and output port proxies*, a set of *message channels* which are transaction conduits between the software simulator and the emulator, *message input and output ports*, and a set of user implemented *transactors*. Figure 4.4 depicts this new configuration.

The SCE-MI infrastructure performs the task of serving as a transport layer that guarantees delivery of *messages* between the *message port proxy* and *message port* ends of each channel. Messages arriving on input channels are presented to the transactors through *message input ports*. Similarly, messages arriving on output channels are dispatched to the *DUT proxy* software models via *message output port proxies* which present them to the rest of the testbench as if they had come directly from the original untimed DUT model (shown in Figure 4.3). In fact, the testbench models do not know the messages have actually come from and gone to a totally different abstraction level.

The DUT input proxies accept untimed messages from various C models and send them to the message input port proxies for transport to the hardware side. The DUT output proxies establish callbacks or provide functions that monitor the message output port proxies for arrival of messages from the hardware side. In other words, the SCE-MI infrastructure *dispatches* these messages to the specific DUT proxy models to which they are addressed. Taking this discussion back to the context of users of the interface described in Figure 4.3, the *end-user* only has to know how to interface the DUT proxy models on the software side of Figure 4.4 with the transactor models on the hardware side; whereas, the *transactor implementer* authors the proxy and transactor models using the SCE-MI message port (and clock control components between them in the macro-based use model), and provides those models to the end-user.

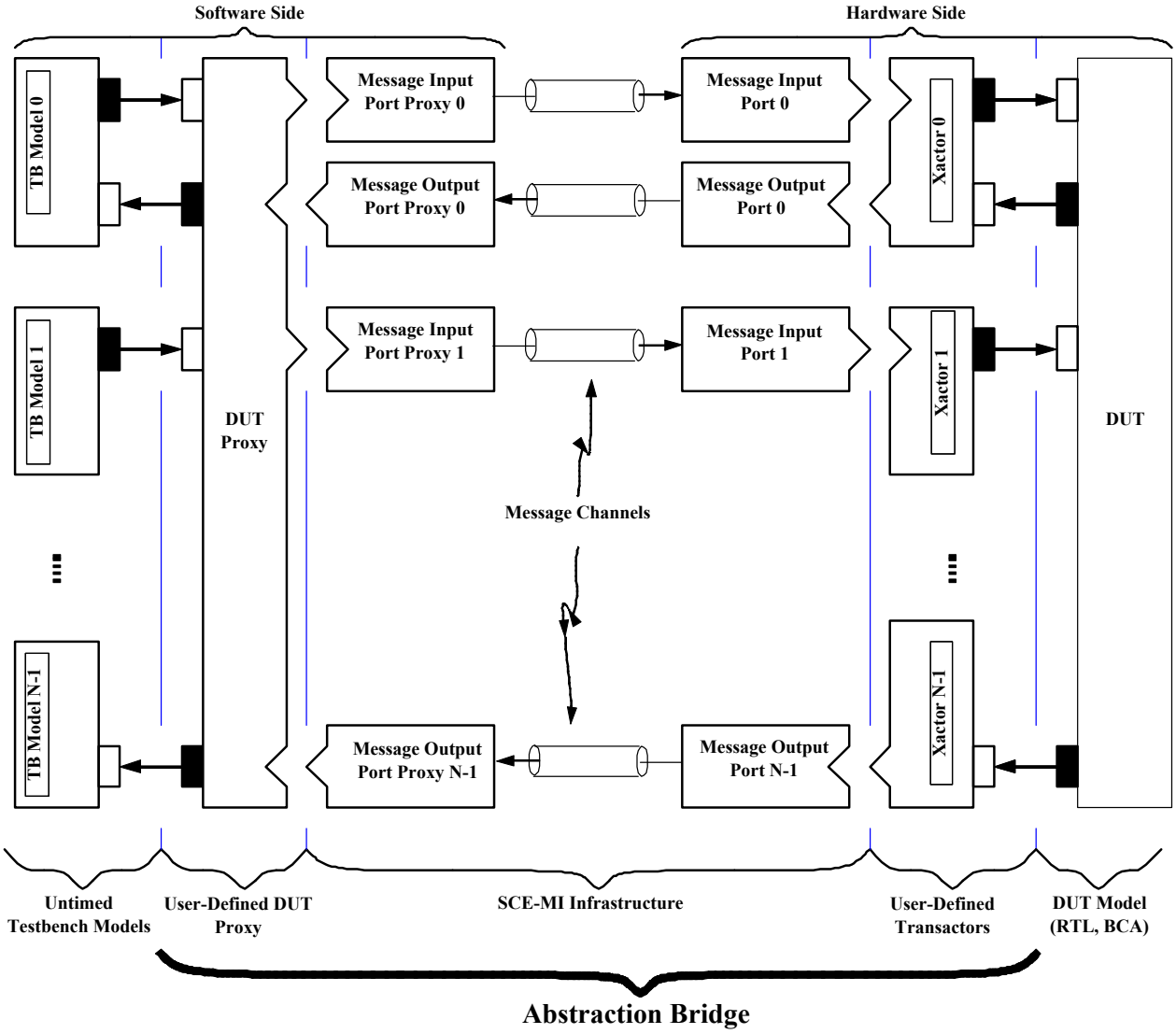



Figure 4.4 Multi-channel abstraction bridge architecture

### 4.4.3 Messages and transactions

In a purely untimed modeling environment, messages are not associated with specific clocks or events. Rather, they can be considered arbitrary data types ranging in abstraction from a simple bit, Boolean, or integer, on up to something as complex as a C++ class or even some aggregate of objects. It is in this form that messages can be transported either *by value* or *by reference* over abstract ports between fully untimed software models of the sort described in Figure 4.4 (and, in substantially more detail, in bibliography [B2]).

However, before messages can be transported over an SCE-MI message channel, they need to be serialized into a large bit vector by the DUT proxy model. Conversely, after a message arrives on a message output channel and is dispatched to a DUT output proxy model, it can be *de-serialized* back into an abstract C++ data type. At this point, it is ready for presentation at the output ports of the DUT proxy to the connected software testbench models.

Meanwhile, on the hardware side, a message arriving on the message input channel can trigger dozens to hundreds of clocks of event activity. The transactor decomposes the message data content to sequences of clocked events that are presented to the DUT hardware model inputs. Conversely, for output messages, the

transactor can accept hundreds to thousands of clocked events originating from the DUT hardware model and then assemble them into serialized bit streams which are sent back to the software side for de-serialization back into abstract data types.

For the most part, the term *message* can be used interchangeably with *transaction*. However, in some contexts, *transaction* can be thought of as including infrastructure overhead content, in addition to user payload data (and handled at a lower layer of the interface), whereas the term *message* denotes only user payload data.

### 4.4.4 Controlled and uncontrolled time

One of the implications of converting between message bit streams and clocked events in the macro-based use model is the transactor might need to "freeze" controlled time while performing these operations so the *controlled clock* that feeds the DUT is stopped long enough for the operations to occur. In the other use modes, time on the HW side is frozen implicitly when the SW side is called.

Visualizing the transactor operations strictly in terms of controlled clock cycles, they appear between edges of the controlled clock, as shown in the *controlled time view* within Figure 4.5. But if they are shown for all cycles of the *uncontrolled clock*, the waveforms would appear more like the *uncontrolled time view* shown in Figure 4.5. In this view, the controlled clock is suspended or disabled and the DUT is "frozen in controlled time."

Now, suppose a system has multiple controlled clocks (of possibly differing frequencies) and multiple transactors controlling them. Any one of these transactors has the option of stopping any clock. If this happens, all controlled clocks in the system stop in unison. Furthermore, all other transactors, which did not themselves stop the clock, shall still sense the clocks were globally stopped and continue to function correctly even though they themselves had no need to stop the clock. In this case, they might typically idle for the number of `uclock`s during which the `cclock`s are stopped, as illustrated in Figure 4.5.

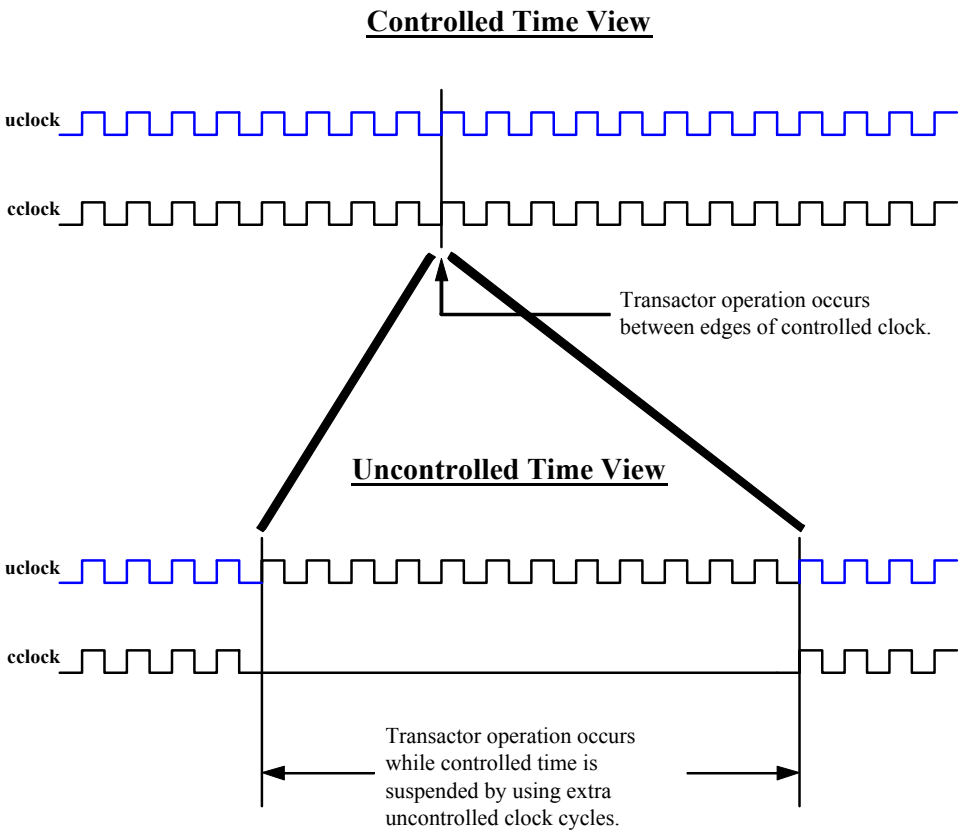



Figure 4.5 Controlled and uncontrolled time views

In the SCE-MI macro-based interface, the semantics of clock control can be described as follows.

Any transactor can instruct the SCE-MI infrastructure to stop the controlled clock and thus cause controlled time to freeze.

- All transactors are told by the SCE-MI infrastructure when the controlled clock is stopped.
- Any transactor shall function correctly if controlled time is stopped due to operations of another transactor, even if the transactor in question does not itself need to stop the clock.
- A transactor might need to stop the controlled clock when performing operations that involve decomposition or composition of transactions arriving from or going to a message channel.
- The DUT is always clocked by one or more controlled clocks which are controlled by one or more transactors.
- A transactor shall sample DUT outputs on valid controlled clock edges. The transactor can use a clock control macro to know when edges occur.
- All transactors are clocked by a free running uncontrolled clock provided by the SCE-MI hardware side infrastructure.

## 4.5 Work flow

There are four major aspects of work flow involved in constructing system verification with the SCE-MI environment:

- software model compilation
- infrastructure linkage
- hardware model elaboration

- software model construction and binding

### 4.5.1 Software model compilation

The models to be run on the workstation are compiled using a common C/C++ compiler or they can be obtained from other sources, such as third-party vendors in the form of IP, ISS simulators, etc. The compiled models are linked with the software side of the SCE-MI infrastructure to form an executable program.

### 4.5.2 Infrastructure linkage

*Infrastructure linkage* is the process used by in the macro-based use model that reads a user description of the hardware, namely the source or *bridge* netlist which describes the interconnect between the transactors, the DUT, and the SCE-MI interface components, and compiles that netlist into a form suitable for emulation. Part of this compile process can involve adding additional structure to the bridge netlist that properly interfaces the user-supplied netlist to the SCE-MI infrastructure implementation components. Put more simply, the infrastructure linker is responsible for providing the core of the SCE-MI interface macros on the hardware side.

As part of this process, the infrastructure linker also looks at the parameters specified on the instantiated interface macros in the user-supplied bridge netlist and uses them to properly establish the dimensions of the interface, including the:

- number of transactors
- number of input and output channels
- width of each channel
- number of clocks
- clock ratios
- clock duty cycles

Once the final netlist is created, the infrastructure linker can then compile it for the emulation platform and convert it to a form suitable to run on the emulator.

The *Infrastructure linkage* process is optional as when only the function-based and pipes-based use models are used as this step is provided natively by the interfaces for these two use models.

### 4.5.3 Hardware model elaboration

The compiled netlist is downloaded to the emulator, elaborated, and prepared for binding to the software.

### 4.5.4 Software model construction and binding

The software executable compiled and linked in the software compilation phase is now executed, which constructs all the software models in the workstation process image space. Once construction takes place, the software models bind themselves to the message port proxies using special calls provided in the API. Parameters passed to these calls establish a means by which specific message port proxies can *rendezvous* with its associated message port macro in the hardware. Once this binding occurs, the co-modeling session can proceed.

## 4.6 Macro-based SCE-MI interface components

The SCE-MI run-time environment consists of a set of interface components on both the *hardware side* and the *software side* of the interface, each of which provides a distinct level of functionality. Each side is introduced in this section and detailed later in this document (see Chapter 5).

### 4.6.1 Hardware side interface components

The interface components presented by the SCE-MI *hardware side* consist of a small set of macros which provide connection points between the transactors and the SCE-MI infrastructure. These compactly defined and simple-to-use macros fully present all necessary aspects of the interface to the transactors and the DUT. These macros are simply represented as empty Verilog or VHDL models with clearly defined port and parameter interfaces. This is analogous to a software API specification that defines function prototypes of the API calls without showing their implementations.

Briefly stated, the four macros present the following interfaces to the transactors and DUT:

- message input port interface
- message output port interface
- controlled clock and controlled reset generator interface
- uncontrolled clock, uncontrolled reset, and clock control logic interface

### 4.6.2 Software side interface components

The interface presented by SCE-MI infrastructure to the software side consists of a set of C++ objects and methods which provide the following functionality:

- version discovery
- parameter access
- initialization and shutdown
- message input and output port proxy binding and callback registration
- rendezvous operations with the hardware side
- infrastructure service loop polling function
- message input send function
- message output receive callback dispatching
- message input-ready callback dispatching
- error handling

In addition to the C++ object oriented interface, a set of C API functions is also provided for the benefit of pure C applications.

## 4.7 Function-based Interface

### 4.7.1 Overview

This section describes some of the attributes of the function-based interface based on SystemVerilog DPI. These attributes are listed follows:

- The DPI is API-less
- Define a function in one language, call it from the other - universal programming concept, easy to learn
- The function call is the transaction
- Function calls provide mid-level of abstraction - not too high, not too low
- SystemVerilog DPI is already a standard

These attributes are discussed in more detail in the following sections.

### 4.7.2 The DPI is API-less

The SystemVerilog DPI was designed to provide an easy to use inter-language communication mechanism based on simple function calls. The idea is, rather than providing an API, simply allow the user to create his or her own API by defining functions in one language and calling them from the other.

### 4.7.3 Define a function in one language, call it from the other

Functions are defined and called in their native languages. This requires very little training for a user to understand. The “golden principle” of DPI is, on each side, the calls look and behave the same as native function calls for that language.

The following figures depict this simple principle both for the C-to-HDL and HDL-to-C directions:

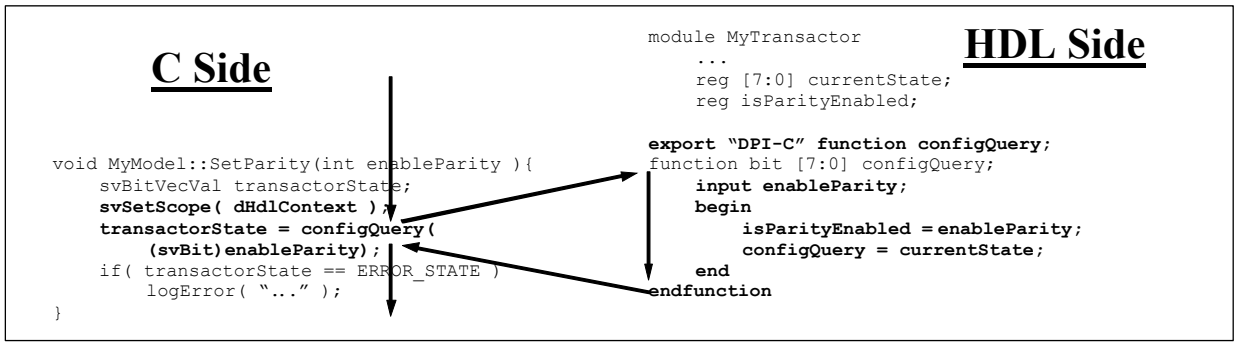



Figure 4.6 Define a function in HDL, call it from C

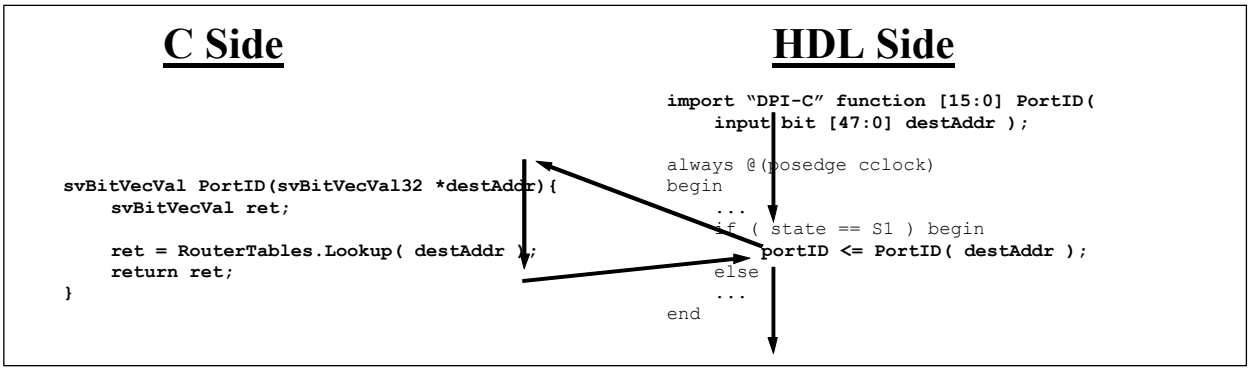



Figure 4.7 Define a function in C, call it from HDL

The DPI SystemVerilog layer is described in detail in the SystemVerilog LRM IEEE 1800 (See reference [3]).

The DPI SystemVerilog layer is designed to allow imported and exported function calls to be used with identical semantics to plain SystemVerilog functions. This means that argument passing and calling conventions remain identical.

In addition, all scoping considerations remain identical. For example the calling scope of a call to any SystemVerilog function call is the scope where the function is defined and not the caller site. In the case of an imported function, special function declaration syntax serves as a place holder for where the function would actually be defined if it were a plain SystemVerilog function. That placeholder represents a declaration of the actual function definition itself which is on the C side. As with plain SystemVerilog functions, the calling scope of this function is considered to be the scope of this import declaration rather than the caller site. *This becomes important when understanding calling scope for purposes of context handling as described in section 4.7.8.*

Here is an example of an imported function declaration in SystemVerilog:

```
// Declare an imported context sensitive C function with cname "MyCFunc"
import "DPI-C" context MyCFunc = function integer MapID(int portID);
```

This declaration is telling the SystemVerilog side that, "there's a C function called `MyCFunc()` that can be called directly from SystemVerilog as the aliased SystemVerilog name `MapID()`".

When the Verilog code makes a call to `MapID()`, this results in the C function `MyCFunc()` being called. This is very useful when resolving incompatibilities in legal names between the C language and the SystemVerilog language. For example a SystemVerilog name could be an escaped identifier that is illegal in C. This can be easily fixed by choosing a legal C name and using aliasing in the import declaration.

For exported functions, the entire function body is defined in some module scope in SystemVerilog. Special additional declaration syntax is used to declare that function is allowed to be called from the C side, for example,

```
export "DPI-C" SetParityGetConfig = function configQuery;
function int configQuery;
    input bit enableParity;
    begin
         isParityEnabled = enableParity;
         configQuery = currentState;
    end
endfunction
```

In this example the variables `isParityEnabled` and `currentState` are defined in the same module scope as the function `configQuery()` and can thus be accessed freely by the function itself.

Like imported functions, C-name aliasing works for exported functions as well. In this case, when the C side calls the function `SetParityGetConfig()` the HDL function `configQuery()` will actually get called.

### 4.7.4 The function call is the transaction

- The function call itself is the transaction and the function call arguments (input plus output) comprise the transaction's named data members - this avoids having to use slices and bit fields of a single big vector
- Function calls can have individually named input args, or output args, or both, or neither
- In SystemVerilog a wide range of data types can be used for function arguments but for SCE-MI 2 it is restricted to a useful subset consisting of bit vectors and integers

### 4.7.5 Function calls provide mid level of abstraction - not to high, not to low

Function calls provide a good "lowest common denominator" mid level abstraction for transporting simple transactions across language domains.

Low enough abstraction for:

- synthesizeability
- use with legacy ANSI C.

High enough abstraction for:

- building user defined simple transactor applications
- building simulation-orineted, reusable verification IP
- providing a good base upon which users can build higher abstraction interfaces (such as TLM, SystemVerilog mailboxes)
- optimal implementation for targeted verification engine (simulation or acceleration)
- providing a deterministic programming interface
- avoiding the need to be aware of uncontrolled time and clock control in HDL.

### 4.7.6 SystemVerilog DPI is already a standard

SCE-MI 2 is leveraging the fact that the SystemVerilog DPI:

- Has been a standard since 2007. It went through a thorough development process and has been proven in several arenas. It has also had significant industry exposure (see bibliography [B4], [B5] and [B6]).
- Clearly defined syntax of function declarations in SystemVerilog.
- Clearly defined argument data type mappings between SystemVerilog and C (see section 5.6.1.2.1).
- Clearly and rigidly defined semantics of calling functions in terms of argument passing conventions, time consumption of the function (0-time vs. time consuming - see section 5.6.2.2), and other details.

- Is explicitly designed to be binary compatible across implementation for any given C host platform and compiler tool (such as GNU gcc-3.2).

### 4.7.7 DPI Datatypes

The philosophy that was used in the development of the DPI was to make the type mappings between C and SystemVerilog as common sense and simple as possible and to minimize the requirements for special helper functions that are used to convert from one type to the other. In other words, define a type in SystemVerilog, define the same type in C the way your common sense would tell you to, and the two will match.

Basic C scalar types, structures, and unpacked arrays of such types, will map directly to equivalent SystemVerilog types almost literally. There are some caveats to this however:

- SystemVerilog integer types are specified to be of fixed size regardless of the inherent data width of a given machine architecture. For example the SystemVerilog types byte, shortint, int, and longint specifically have widths of 8, 16, 32, and 64 bits respectively.
- Unfortunately, by contrast in ANSI C, integer types do not have widths that are as cast in stone as the corresponding types in SystemVerilog (see Wikipedia reference for ANSI C data types at:
    - http:// en.wikipedia.org/wiki/C_variable_types_and_declarations and
    - http://www.opengroup.org/public/ tech/aspen/lp64_wp.htm for a further explanation of this paradox). What this means is that even though there is a fixed correspondence between fixed sized SystemVerilog integer types and non- fixed sized ANSI C integer types as shown in the table above, it will be up to the user to understand which bits of data passed between SystemVerilog and C are significant and where padding/masking is implied/required. But despite this caveat, the use of scalar types to pass small data values by value back and forth between the language domains is extremely useful and should be supported to the extent possible in the SCE-MI 2 standard (see proposed type support for SCE-MI 2 below).
- While byte is always unsigned in SystemVerilog, shortint, int, and longint are always signed. The four types correspond to the C types char, short int, int, long long signed types respectively. The correspondence of byte to C char is a bit of an oddity that may have been overlooked by the SystemVerilog C language interfaces committee. It would really make more sense that this is unsigned char.

Additional complexities arise with bit vector (packed array) types and open arrays. For these, great care was taken to make their mappings as easy to use and intuitive as possible.

### 4.7.8 Context Handling

Context handing in DPI is the term used to refer to the mapping of an imported function call to an instance of user C data (such as an object pointer) that was previously associated with the SystemVerilog caller module instance.

This is useful for maintaining an association between, for example, a pointer to a SystemC proxy module and the instance of the SystemVerilog transactor associated with it. Because an imported function call is just a C function, by definition, it has no context as would say a method or member function of a C++ class. Context handling in SystemVerilog DPI is very similar to context handling for *receive callbacks* in the SCE-MI macro-based interface (see 5.4.7.1). In the case of SCE-MI macro-based interface, the Context data member of the `SceMiMessageOutPortBinding` struct is used to pass a user model context to the receive callback function that can be associated with an instance of an output message port, as shown in Figure 4.8.

```
// Define the function and model class on the C++ side:
class MyCModel {
    private:
        int locallyMaped (int portID); // Does something interesting…

        sc_event notifyPortIdRequest;
        int portID;

    pPublic:
        // Constructor
        MyCModel (const char * instancePath) {
             SceMiMessageOutPortBinding outBinding =
                    = { this, myCFunc, NULL }

             SceMiMessageOutPortProxy outPort = outPort->BindMessageOutPort (
                    instancePath, "SceMiMessageOutPort", outBinding );
        }

        friend int myCFunc ( int portID );
};

// Implementations of receive callback function SCE-MI
void MyCFunc ( void *context, const SceMiMessageData *data ) {

    MyCModel* me = (MyCModel*)context;

    me->portID = data->Get(0);
    me->notifyPortIdRequest.notify();
}
```

Figure 4.8 Context handling in SCE-MI Macro-based interface

In SCE-MI 2 function-based interface, context binding is similarly established at initialization time by storing a context pointer with a SystemVerilog module instance scope and later retrieving it via `svGetScope()` and `svGetUserData()`.

Figure 4.9 shows an example of context handing in SCE-MI 2 function-based interface:

```
SV Side:
        // Declare an imported context sensitive C function with name "MyCFunc"
        Import "DPI-C" context MyCFunc = function integer MapID ( int portID );

C Side:
        // Define the function and model class on the C++ side:
        class MyCModel {
          private:
              int locallyMapped ( int portID ); // Does something interesting…

          public:
          // Constructor
          MyCModel ( const char* instancePath ) {
              svScope  scope = svGetScopeFromName ( instancePath );
              // Associate "this" with the corresponding SystemVerilog scope
              // for fast retrieval during runtime.
              svPutUserData ( svScope, (void*) MyCFunc, this );
          }

          Friend int MyCFunc ( int portID );
        };

        // Implementation of imported context function callable in SV
        Int MyCFunc ( int portID ) {
          // Retrieve SV instance scope (i.e. this function's context).
          svScope = svGetScope();

          // Retrieve and make use of user data stored in SV scope.
          MyCModel* me = (MyCModel*)svPutUserData ( svScope, (void*) MyCFunc );
          Return me->locallyMapped ( portID );
        }
```

Figure 4.9 Context handling in SCE-MI 2 function-based interface

In this example notice that because functions can have both input and output arguments, the return argument can be sent directly out of the function return argument. In the SCE-MI macro-based interface, the receive callback must notify another thread to send the mapped `portID`.

# 4.8 Pipe-based Interface

## 4.8.1 Overview

As currently defined, the DPI standard handles strict reactive semantics for function calls. There are no extensions for variable length messaging and streaming data.

The SCE-MI 2 supports constructs called *transaction pipes* which can be implemented as *built- in* library functions. Transaction pipes can potentially be implemented with reference source code that uses basic DPI functions, or can be implemented in an optimized implementation specific manner.

A transaction pipe is a construct that is accessed via function calls that provides a means for streaming transactions to and from the HDL side.

Two operation modes are defined for transaction pipes that enable different data visibility semantics. They are called deferred visibility and immediate visibility modes.

Generally speaking, in *deferred visibility* mode, there is a precisely defined lag between when elements are written to pipe by the producer side and when they are actually visible and available for consumption by the consumer side. In this mode a pipe may absorb one or more elements when non-blocking send calls are made but the consumer will not see these elements until the pipe either fills or is flushed.

Whereas, in *immediate visibility* mode, any elements written by the producer side are immediately visible by the consumer side that next time it gains execution control for any reason.

Transaction pipes are as easy to use as simple function calls, yet have semantics that can be thought of as a hybrid between UNIX sockets, UNIX file streams and UNIX named pipes.

- Like UNIX sockets, transaction pipes provide a facility for sending one-way message passing through simple function calls. Transaction pipes are composed of send and receive calls that look very much like write and read calls to UNIX sockets (but are much easier to create and bind endpoints).
- Like UNIX file streams, items written to the pipe can be buffered by the infrastructure which allows for more optimal streaming throughput. Pipes leverage the fact that in some cases round trip latency issues can be avoided by using pipelining, and therefore more effective throughput of streamed transactions can be realized.
- Like UNIX file streams, transaction pipes can be flushed. Flushing a transaction pipe has the effect of guaranteeing to the writer of the transaction that the reader of the transaction at the other end has consumed it. This is useful for providing synchronization points in streams.
- Like UNIX named pipes, each transaction pipe is uniquely identified with a name. In the case of transaction pipes, that name is the hierarchical path to the interface instance of the HDL endpoint of the pipe.

Transaction pipes are unidirectional meaning that in any given pipe, the transactions only flow in one direction. The data sent by the producer is guaranteed to be received by the consumer in the same order when the consumer asks for the data (by calling a function). However, the data is not guaranteed to be available to the consumer immediately after it was sent depending on how buffering is used. That is, if the pipe has some amount of buffering, that could continue to be filled by a producer thread as long as there is room. The consumer would not see it until control is yielded to the consumer. This could happen if either the pipe filled while being written to, thus suspending the producer, or via a flush operation initiated by the producer. See 5.8.4.3 for more information on flush operations.

Transaction pipes that pass one-way transactions from the C side to the HDL side are called *input pipes*. Pipes that pass transactions from the HDL side to the C side are called *output pipes*.

Unlike normal DPI calls, in which one end calls and the other end is called, models on both ends of a transaction pipe call into the pipe, with one end calling the send function and the other calling the receive function.

### 4.8.2 Streaming Pipes vs. TLM FIFOs

A blocking interface is well suited to true streaming applications and follows the easy use model of UNIX streams as discussed previously.

It is useful to compare and contrast the semantics of streaming pipes to those of FIFOs - particularly the FIFOs that follow the semantics of TLM FIFOs described in the OSCI-TLM standard. A possible reason we often stumble when discussing issues like user vs. implementation specified buffer depth, its effect on determinism, etc. is because people are thinking of a FIFO model rather than a pipe model.

Both pipes and FIFOs are deterministic and have similar functions in term of providing buffered data throughput capability. But they have different basic semantics.

Here is a small listing that tries to compare and contrast the semantics of FIFOs vs. pipes:

FIFOs

- Follow classical OSCI-TLM like FIFO model
- User specified fixed sized buffer depth
- Automatic synchronization
- Support blocking and non-blocking put/get operations
- "Under the hood" optimizations possible - batching
- No notion of a flush

Pipes

- Follows Unix stream model (future/past/present semantics)
- Implementation specified buffer depth
- User controlled synchronization
- Makes concurrency optimization more straightforward
- Support only blocking operations (for determinism)
- "Under the hood" optimizations possible - batching, concurrency
- More naturally supports data shaping, vlm, eom, flushing

One could argue that we may wish to entertain the notion of a "SCE-MI_FIFO" reference library to augment the "SCE-MI_PIPE" reference library currently proposed and thus provide two alternative DPI extension libraries that are part of the SCE-MI 2 proposal that address different sets of user needs.

But it is useful to make the clear distinction between FIFOs and pipes and, for now, at least converge on the semantics of proposed pipes and making sure they address the original requirements of variable length messaging.

Pipes are intended for streaming, batching, variable length messages, and potentially can be used even for more exotic purposes if the modeling subset allows it. Given that pipes can be implemented at the application layer, the choice between using pipes and DPI is one of convenience in many cases. However, since an implementation can choose to provide an optimized version of the pipes, this would be a factor as well in the choice to use them.

In order to facilitate this FIFO model, the following chapter proposes TLM compatible, thread-neutral transaction FIFO interface.

### 4.8.3 Reference vs. optimized implementations of transaction pipes

The HDL-side API can be implemented as a built-in library, but it must allow the user to use the API with a syntax that is exactly compatible with the SystemVerilog interface declarations as described above.

On the C-side, the transaction pipes API might be used to build a higher level C++ object oriented interfaces to pipes that may provide a more user friendly object oriented interface to basic pipes.

The C-side transaction pipes API could also conceivably be used to build alternative native HVL object-oriented interfaces such as OSCI-TLM interfaces.

While not required, it is possible to implement pipes as a reference model of library functions of source code built over basic DPI function calls. As such they can be made to run on any DPI compliant software simulator.

It is an absolute requirement however that DPI based implementations and built-in implementations of pipes must have identical deterministic behavior and must strictly adhere to the semantics defined in this specification.

#### 4.8.3.1 Implementation of pipes in multi-threaded C environments

Pipes blocking access functions does not have a thread-neutral API that can be used to aid in adapting the implementations of user friendly (but thread-aware) blocking pipe functions to arbitrary threading systems on the C side.

To satisfy this requirement the pipes interface was designed to address the following needs:

- A user-friendly, but thread-aware pipe interface (which the blocking pipe functions already provide).
- A lower level implementation-friendly, but thread-neutral pipe interface - essentially implementation API and callback functions to facilitate easy creation of adapters that allow implementation of the user-friendly API in selected C threading environments.

Transaction pipes provide a solution to the second requirement. It provides:

- Easy-to-use blocking pipe access API at the user level.
- Thread neutral API and callback functions that implementations can choose to use to create adapter layers that implement pipes over a selected threading system.

- Easy to demonstrate reference implementation of the blocking pipe calls that uses the pipe API and callback functions in their implementation. The example below shows a working reference model of such an implementation for the HDL side.

In summary, the non-blocking calling and callback functions for pipes described in section 5.8.2 provide thread-neutral functions that can be used by any implementation to implement the thread-aware blocking pipe access calls.

### 4.8.4 Deadlock Avoidance

SCE-MI pipe implementations are in no way expected to guard against application induced deadlocks.

Note: An example of an application induced deadlock is the case where an HDL-side process is blocking while waiting for data on an empty input pipe, a yield to the C-side producer thread occurs, but the C-side never feeds more data into the pipe. In this case, this specific HDL-side process would never advance (deadlock). Another example is in the output direction where an HDL-side process is blocking while trying to feed data to a full output pipe, a yield to the C-side consumer thread occurs, but the C-side never drains data from the pipe. In this case, this specific HDL-side process would never advance (deadlock). In both of these cases it is up to the application to be properly designed to avoid such deadlocks.

### 4.8.5 Input Pipe

Figure 4.10 shows an example of the use of an *input pipe* on both the C and HDL sides:

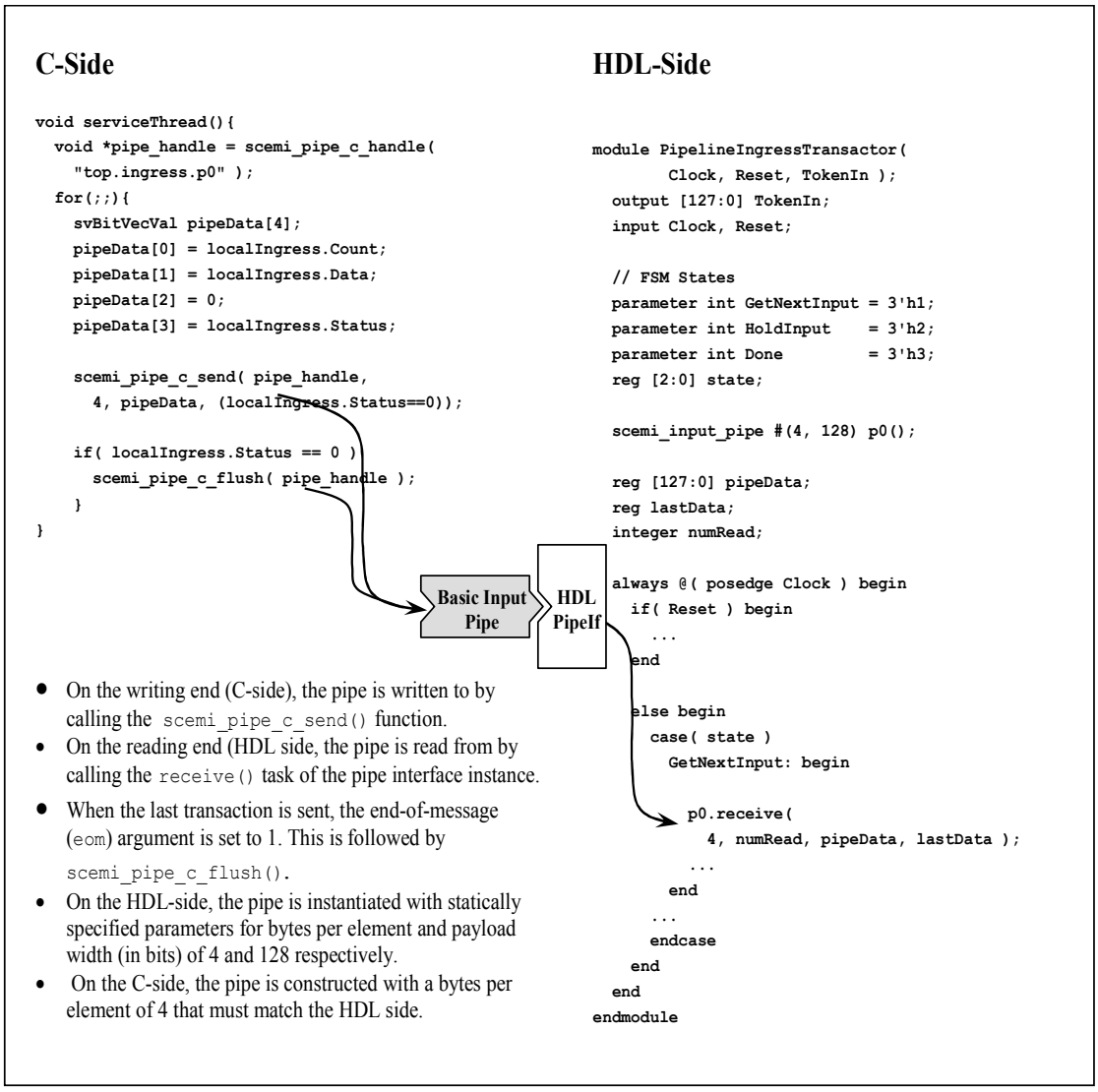



Figure 4.10 Example of Input Pipe

### 4.8.6 Output Pipe

Figure 4.11 shows an example of the use of an *output pipe* on both the C and HDL sides:

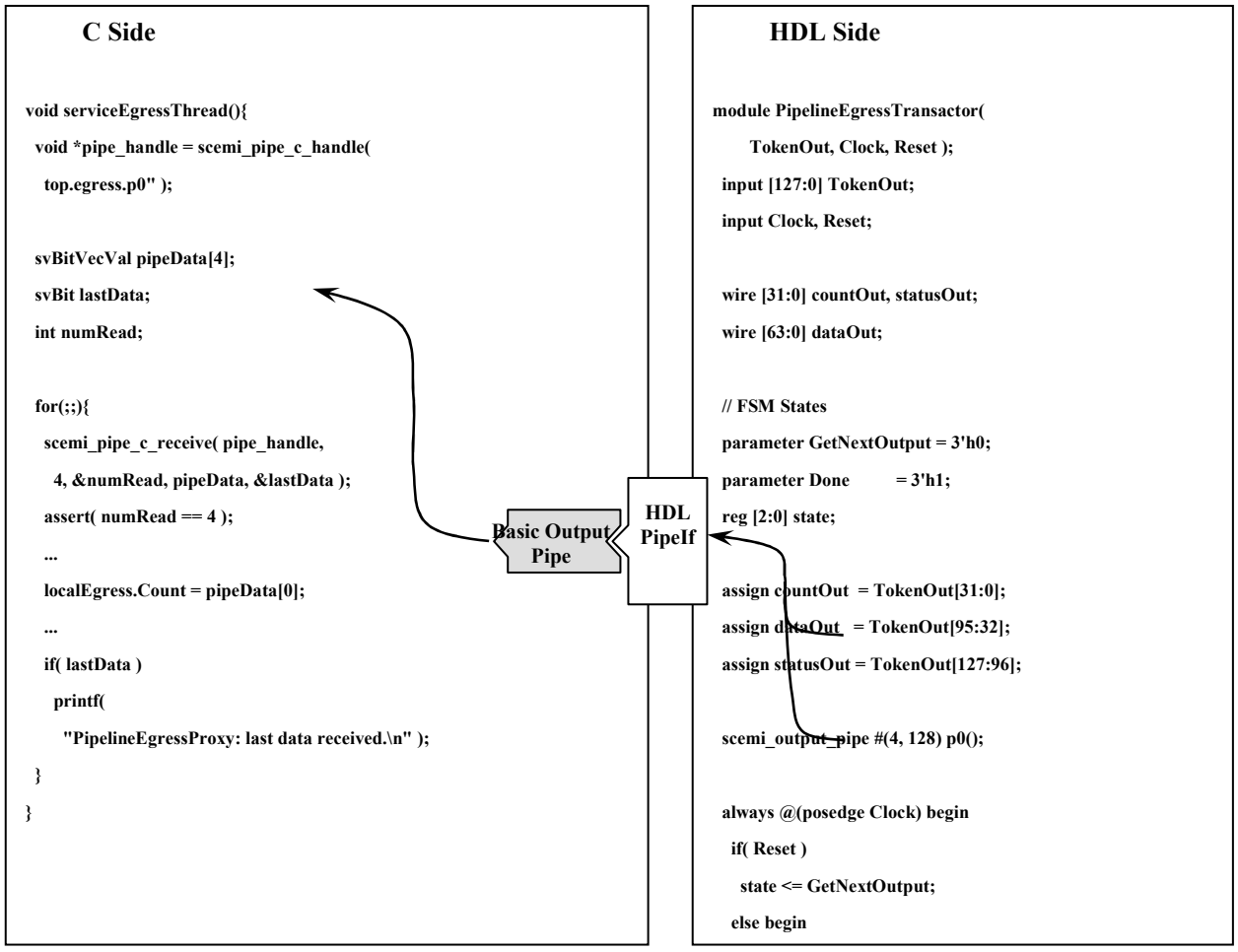



Figure 4.11 Example of output pipe

### 4.8.7 Implementation defined buffer depth for pipes, user defined buffer depth for FIFOs

For a typical streaming use model, a user may instantiate a pipe with the `BYTES_PER_ELEMENT` and/or `PAYLOAD_MAX_ELEMENTS` specified but the `BUFFER_MAX_ELEMENTS` left alone, for example,

```
scemi_input_pipe #(
    .BYTES_PER_ELEMENT(32),
    .PAYLOAD_MAX_ELEMENTS(16)) p0( ... );
```

By not specifying depth, pipes used in streaming applications can benefit from pipe depths that are optimal for a given implementation. This will allow streaming of transactions in an optimal manner for each implementation. This use model may typically choose to use a flush/eom mechanism with the pipe as well to define proper synchronization points between producer and consumer.

For a typical FIFO oriented use model (such as TLM FIFOs), a user will explicitly want to specify the pipe to be a specific depth which will facilitate consistent behavior in terms of how long threads continue to write to or read from pipes before yielding.

Such a use model may typically choose not to use a flush mechanism.

### 4.8.8 Variable length messaging

In addition to providing a means of highly optimizing streaming performance, transaction pipes can be a natural mechanism to implement *variable length messaging*.

Consider the case of the transmission of an Ethernet frame transaction. As per the IEEE 802.3 Ethernet standard, a frame can be anywhere up to 1500 bytes (although there is some disagreement if this is data payload size or total frame size). However, in some applications, typical frames may be far smaller. This is a classic example of where a variable length transaction would be useful as it saves the overhead of transmitting a fixed width 1500 byte transaction every time regardless of actual length.

Using pipes one could implement this example as follows. Let's assume for the sake of simplicity that we are transmitting frames from the C side to the HDL side:

- The HDL side declares an input pipe with `BYTES_PER_ELEMENT` statically specified as 1 (i.e. the default value) in its transactor module scope and makes calls to it with a `num_elements = 1`.
- Using the data shaping capability, each time the C side calls the send function it sends an array of bytes with `num_elements` set to whatever the desired number of bytes is which can vary from call to call (hence variably sized messages)
- On each send call, the C side sets `eom` to 1 since it is sending all the bytes at once
- Because the receive side is only reading a byte at a time, it will not see the `eom` indication until the last byte is received.

Because pipes can, at the option of the implementer, be optimized for streaming, one can imagine that if there are several such interfaces generating traffic simultaneously (say with a multi-port Ethernet packet router) the benefit from concurrency of execution (between the multiple threads on the workstation and the emulator) within the transmission of each frame could be appreciable.

One can also envision another scenario where a sequence of several sequential frames could be sent before an actual *flush* is performed. This would support streaming of multiple sequential variable length frames before synchronization is required.

One can also consider a pure streaming data thread to be one long variable length message (or sequence of them) that lasts the entire simulation, essentially requiring no synchronizations in the interim, such as feeding the entire contents of a file as traffic for an interface with a flush only occurring at the very end.

#### 4.8.8.1 Variable length messaging features of transaction pipes

Three areas have been identified that are desirable to support with transaction pipes:

- Data shaping
- End-of-message <eom> marking mechanism
- Support for multiple pipe transactions in 0-time

##### 4.8.8.1.1 Data shaping

Data shaping is a concept that addresses the need for random access to variable length messages. Data shaping simply allows a transaction pipe to have a different width at one end than the other.

For example suppose a frame of 100 elements of data is desired to be sent over an input pipe 1 element at a time but the consumer of the frame wants random access to the entire variable length message of 100 elements. The consumer could read the entire 100 elements in one call. The producing end could write 1 element per call.

In this case transmission of the elements would be buffered but time would be stopped on the reading end until all 100 elements are received since, the read is blocking. Once received, any or all elements could be accessed as desired.

In this case the send end of the pipe is narrower than the receive end. One can refer to such a pipe as a *nozzle*.

Conversely suppose the producer wished to send the frame of 100 elements of data all at once but the consumer only wanted to read 1 element at a time. The producer could send all 100 elements in a single call. The producer end of the pipe could receive only one element with each read call.

In this case, transmission of the elements would be buffered but time could advance on the reading end between each element read since each is a separate call that can be separated by `@(posedge clock)` statements for example.

In this case the send end of the pipe is wider than the receive end. One can refer to such a pipe as a *funnel*.

#### 4.8.8.1.2 End-of-message <eom> marking mechanism

Using the `eom` the user can mark the end of a message or "last data".

This flag can be queried at the receive end to know if it is the end of the message. The infrastructure does nothing with this flag (unless autoflush is enabled – see section 5.8.4.3.3), it is simply passed as received. However, if data shaping is involved, the infrastructure does not pass the `eom` flag until the last element is read by the consumer, regardless of the shape of the data.

So for example, in the case of a funnel, if the sender sends 100 elements all at once and sets the `eom` flag to 1 and the receiver only reads one element at a time, it will not see the `eom` set to 1 until the last element.

Conversely, in the case of a nozzle, if the sender sends 1 element at a time and only sets the `eom` flag to 1 on the last one, and the receiver reads 100 elements at a time, the receiver will see the `eom` flag set to 1 on the first read of the message.

Special considerations must be made if a producer endpoint of a nozzle does a data send operation with a smaller `num_elements` than that requested by the subsequent data receive operation at the consumer endpoint of that nozzle. If an `eom` is specified on that send operation, in order to satisfy its request the consumer will see a return of `num_elements_valid` that is smaller than its requested `num_elements`. This is because, in order to satisfy the producer's `eom` condition, the consumer's blocking receive call must have satisfactorily returned from its read operation even if that read operation was asking for a larger number of elements than had been sent as of the time of the `eom`.

So, referring back to the nozzle example above where the consumer reads 100 elements, if the producer only sends 75 elements before setting `eom`, the request to read 100 elements will return with the `eom` bit set but with a `num_elements_valid` of only 75.

#### 4.8.8.1.3 Support for multiple pipe transactions in 0-time

Operation of pipes is identical whether successive access operations (sends or receives) are done in 0-time or over user clock time, i.e. 1 access per clock. It is strictly a function of modeling subset as to whether 0-time operations are supported or not. But the pipe interface itself does nothing to preclude transmission of multiple transactions in 0-time without requiring the need for user awareness of uncontrolled time. This is true whether the transactions are variable or fixed length messages transmitted through a pipe or whether they are just simple DPI.

## 4.9 Backward Compatibility and Coexistence of Function and Pipes-based Applications with Macro-based Applications

The SCE-MI 2 standard enables new use models that allows higher modeling abstraction and improved modeling ease-of-use over the original macro-based standard. This improvement is embodied mainly in the DPI specification and the capabilities of transaction pipes.

At the same time however, one requirement of the SCE-MI 2 standard is backward compatibility with and coexistence with macro-based applications defined by the SCE-MI 1 standard. The main idea is that pure DPI applications can run in either a simulator that natively supports the SystemVerilog DPI or in a simulator or emulator platform that supports SCE-MI 2 standard (which implies that it also supports SCE-MI 1).

The following sections provide more detail on how the two interfaces can co-exist.

### 4.9.1 What does not change ?

Aside from guaranteeing compatibility with legacy macro-based models, the SCE-MI 2 interface specifically does not change the following:

- The SCE-MI Initialization/Shutdown API
- SceMiClockPort Support for Clock Definitions

### 4.9.2 Error Handling, Initialization, and Shutdown API

SCE-MI 2 function and pipe-based applications can continue to use the macro-based initialization and shutdown API functions without changes:

- SceMi::RegisterErrorHandler()
- SceMi::RegisterInfoHandler()
- SceMi::Version()
- class SceMiParameters
- SceMi::Init()
- SceMi::Shutdown()

The following rules dictate the use of these macro-based calls:

- They are only required for applications that use macro models.
- They are optional for applications that use purely function or pipe models (see definitions of Macro-based vs. Function-based models in section 4.9.4).

Applications with only function or pipe models that choose not to use the error handling, initialization, and shutdown functions above will run on any simulator that supports DPI but does not necessarily support the SCE-MI macro-based initialization and shutdown API standard functions.

### 4.9.3 Requirements and Limitations for Mixing Macro-based Models with Function- and Pipe-based Models

This section describes the formal requirement for preventing mixes of macro constructs with function and pipe constructs in the same transactor and C models.

Macro models can co-exist in an application with function and pipe models but conceptually the following requirements must generally be followed:

- Macro models would be ported as a whole and would not be allowed to intermix function (DPI) and pipe constructs with macro constructs.
- function- and pipe-based models would not be allowed to use macro constructs within the calling hierarchy. In other words, mixing of uncontrolled time interactions with controlled time interactions within the same model would not be allowed (see section 5.6.2.2).
- Legacy macro-based models and function- and pipe-based models would be allowed to co-exist in a single simulated environment.
- These models can share clocks (SceMiClockPorts), but only macro-based models are allowed to use SceMiClockControls)
- On the C side imported DPI functions cannot be called from macro callbacks.
- On the C side macro message input port ::Send() functions cannot be called from DPI imported functions.
- In multi-threaded C environments, calls to ::ServiceLoop() would be restricted to one thread that could be embedded in an implementation's infrastructure so as to hide this detail from users.
- Macro-based callbacks and function-based imported functions alike would be serviced by this same thread.
- For single threaded HVL, use of ::ServiceLoop() would not change.

The following sections present a more formal specification of how the above constraints for model and construct mixing are enforced.

### 4.9.4 Definition of Macro-based vs. Function and pipe-based Models

For purposes of describing requirements of model mixing, the following definitions are given.

The uses of the term model here are somewhat arbitrary but convenient. A model is some level of hierarchy and all its descendants.

A Macro-based HDL model is defined as a hierarchy with the following properties:

- At least one macro-based message port or clock control macro (but not clock port macro) is instantiated at the highest level of the hierarchy within the model.
- More macro-based message ports or clock controls may be instantiated at lower sub-hierarchies of the model.

A Function or pipe-based HDL model is defined as a hierarchy with the following properties:

- At least one DPI function call or pipe-based function call is declared at the highest level of the hierarchy within the model.
- More DPI function calls or pipe-based function calls may be declared at lower sub-hierarchies of the model.

### 4.9.5 Requirements for a Function or pipe-based model

- On the HDL side, no macro-based models as defined above can contain any function- or pipe-based call declarations or calls anywhere in their hierarchy.
- On the HDL side, no function- or pipe-based models as defined above can contain any macro-based message port or clock control macros anywhere in their hierarchy.
- On the C side, no macro-based callback functions can make direct calls to exported DPI function of pipe-based function calls .
- On the C side, no imported DPI function call or pipe-based call can make calls to the macro-based service loop or to send messages on any of the macro-based input ports.

The above requirements force macros and their proxies to only exist in disjoint hierarchies from DPI and pipe-based functions.

### 4.9.6 Subset of DPI for SCE-MI 2

SCE-MI uses a subset of DPI that is restricted in such a way as to provide a nice balance between usability, ease of adoption and implementation. The subset includes:

- Data types used with DPI functions are limited as detailed in section 5.6.1.2.1
- Certain restrictions on calling imported functions from exports and vice versa (see 5.6.2.3 for more details)

### 4.9.7 Use of SCE-MI DPI subset with Verilog and VHDL

The SCE-MI standard does not support using the SCE-MI 2 DPI subset for Verilog 2001 and VHDL 1993. Verilog and VHDL users who prefer not using SystemVerilog can use the SCE-MI macro-based interface defined in section 4. SCE-MI 2 also supports mixed usage of Verilog and VHDL SCE-MI macro-based transactors with SCE-MI function and pipe-based transactors following the use model guidelines descried in the Mixed Usage section 4.9.

### 4.9.8 Support for multiple messages in 0-time

DPI places no restrictions on the number of imported function calls made in the same block of code without intervening time advancement.

One important point to make about the SCE-MI 2 function-based approach is that it does not preclude the ability to support transmission of multiple messages in 0-time either by calling the same function or by calling multiple functions in the same timestep.

This interfacing feature is fundamentally missing from SCE-MI macro-based interface where macros supporting controlled time interfacing are fed with user clocks. The only way of accomplishing this is to use some sort of over-clocking scheme in which the message clock (still a controlled clock) has a frequency that is some multiple of the main clock being used in the transactor.

For example, if I am using a message macro that is clocked by transactor_clock and I wish to send 3 messages between posedges of transactor_clock, I must define essentially a message_clock that is at least 3 times the frequency of transactor_clock. Short of this over-clocking there is no other way to fundamentally accomplish transmission of multiple messages between clocks.

With the SCE-MI 2 function-based approach, multiple messaging is possible. Take the following code example:

```
always @( posedge transactor_clock ) begin
    if( reset == 1 ) begin
        // Do the reset thing ...
    else switch ( fsm_state ) begin
         case `FSM_STATE_1: begin
             ...
             c_function1( data1, data2 );
             c_function1( data2, data3 );
             c_function2( data3, data4 );
         end
         ...
    end
    ...
end
```

In this case, there are two consecutive calls to `c_function_1()`. The first takes `data1` as the input and returns `data2` as the output. The second takes `data2` as the input and returns `data3` as the output. The third call is actually a call to a different function (which could be to different SCE-MI 1 message ports underneath).

## 4.10 Scope of Calling DPI exported functions

The Accellera SystemVerilog working group defines the following:

- The SystemVerilog context of DPI export tasks and functions must be known when they are called, including when they are called by imports. When an import invokes the `svSetScope` utility prior to calling the export, it sets the context explicitly. Otherwise, the context will be the context of the instantiated scope where the import declaration is located. Because imports with diverse instantiated scopes can export the same task or function, multiple instances of such an export can exist after elaboration. Prior to any invocations of `svSetScope`, these export instances would have different contexts, which would reflect their imported caller's instantiated scope.
- The concept of 'call chains' is useful for understanding how context works as control weaves in and out of SystemVerilog and another language through a DPI interface. A DPI import call chain is a series of task or a function invocation that begins with a call from SystemVerilog into a task or function that is defined in a DPI-supported language and is declared in a DPI import declaration. The import call chain consists of successive calls to routines in the imported language. One of those routines becomes the last routine in the import call chain when it calls a SystemVerilog exported task or function. The import call chain can also end by simply unwinding without calling any SystemVerilog export. Once entered, an exported SystemVerilog task or function can transfer control to new import chains by invoking imports and, when exiting, can return control to its caller in the original import call chain.
- In addition, Accellera SystemVerilog working group states that the behavior of DPI utility functions that manipulate context is undefined when they are invoked by any function or task that is not part of a DPI context call chain (see 26.4.3).

In this section we describe the following main use cases where DPI exported functions and DPI utility functions can be called from an application linked with the C side which is considered by the SystemVerilog LRM being "outside a context DPI imported function call chain". Each use case will describe the assumption and the constraints.

This section only applies to DPI exported functions and does not apply to DPI exported tasks.

### 4.10.1 The calling application is linked with the simulation kernel:

This use case applies to standard languages on the C-side that are linked with a simulation kernel running on the HDL-side. The term 'linked with' implies that the language is either simulated directly by the simulator or is handled by the simulation kernel as a direct extension to the simulator running the HDL-side. Examples for

such languages are SystemC, SystemVerilog and Specman *e* that are 'tightly integrated or running natively on the SystemVerilog simulator.

In this case, assume you call a DPI imported function which triggers a thread in the calling application that calls an exported function via the SCE-MI 2 C side. According to the Accellera SystemVerilog working group, such a call is considered outside a context DPI imported function call chain and thus its result is undefined.

However SCE-MI 2 allows such calls to be made if the calling application is linked with the simulation kernel as long as it meets the following requirements:

DPI exported functions can be invoked by C code called from an application linked with a simulation kernel, and outside a context DPI imported function call chain as long as the calling application is triggered (or notified) from a context DPI imported function call chain and executed in zero simulation time or delta simulation time from when the imported DPI function was invoked.

Note: Any calls to DPI exported functions during any other time not covered by the above may result in undefined behavior. These include calling DPI exported functions during HDL side compilation, by C code called by PLI, VPI, VHPI callbacks or from a SystemVerilog system task. It also includes any calls from C code executing concurrently with the SystemVerilog code.

The key constraints when calling exported functions from an application linked with the simulation kernel are:

a) The context of the DPI exported function must be known (or obtainable) before its being called.

b) Only exported functions (that do not consume time) can be called. Calling DPI exported tasks will result in undefined behavior.

c) There is no control of any event ordering meaning no control how events get processed on both the calling application and the HDL side.

d) An imported DPI function call cannot return arguments that are dependent on the exported function calls.

### 4.10.2 Calling application is not linked with the simulation kernel:

In this case, the simulation is not aware of the calling application running on the SW side and therefore the simulation kernel doesn't suspend its execution to let the calling application external to the simulation kernel to run and furthermore to call the DPI exported function. In other words, if the DPI imported function returned, the simulator will proceed and none of the external threads of the calling application will ever wake up.

However assuming that the C code is running under the control of a foreign threading package, then the imported C function can suspend itself allowing other threads of the application to run and call DPI exported functions, and then resume before returning. In this case, the call to a DPI exported function is considered as being made from a Context DPI imported function call chain given that SystemVerilog simulation kernel is not aware of any context switching that is taking place. Furthermore, it really doesn't matter if the external calling application is a simulator that consumes time, and calls the simulator after waiting on time. Until the imported C function called from by the HDL side returns, the simulator kernel on the HDL side is not aware that the imported function was suspended and that an exported function is being called from another thread. Therefore, the imported DPI function call can return arguments that are dependent on the exported function calls and event ordering is defined, meaning that the exported function returns before the DPI imported function returns.

The key constraints when calling exported functions from an application not linked with the simulation kernel are:

- The context of the DPI exported function must be known (or obtainable) before its being called.
- Only exported functions (that do not consume time) can be called. Calling DPI exported tasks will result in undefined behavior.

### 4.10.3 DPI function calls are deterministic

In either of the configurations mentioned in sections 4.10.1 and 4.10.2 it is the case that fundamentally determinism must be guaranteed by the implementation of SCE-MI 2 on a hardware engine, just as it is implicitly guaranteed in software simulator implementations of DPI.

This means that, for a given design, assuming there are no race conditions in that design, that not only must it be guaranteed that simulation results are identical from run to run or even from compile to compile where no design changes occur, but that those results are identical to results of running the same design on a software simulator, in terms of the timing of when SCE-MI compliant DPI calls are made.

In other words, all SCE-MI compliant imported and exported DPI calls must occur in the same time slots during the simulation of a given design whether that design is simulated on a DPI compliant software simulator or a hardware simulator.

# 5. Formal specification

This chapter defines the API calls and macros that make up the entire SCE-MI

## 5.1 General

This section contains items that relate to all aspects of the specification.

### 5.1.1 Reserved Namespaces

Prefixes beginning with the three letter sequence s, c, e, or the four letter sequence s, c, e, _ (underscore), in any case combination, are reserved for use by this standards group.

Prefixes beginning with the five-letter sequence s, c, e, m, i, or the six-letter sequence s, c, e, _ (underscore), m, i, in any case combination, are reserved for use by SCE-MI and SCE-MI related specifications.

### 5.1.2 Header Files

The ANSI-C and C++ API's shall be declared in a header file with the name

```
scemi.h
```

Note: the name is all lowercase, and the same for both API's. Examples of the header files are given in Appendix E and F. Where any discrepancy exists between this specification and the included header file, the specification should be the one that is used.

### 5.1.3 Const Argument Types

All input arguments whose types are pointers with 'const' qualifier should be strictly honored as read-only arguments. Attempts to cast away 'constness' and alter any of the data denoted or pointed to by any of these arguments is prohibited and may lead to unpredictable results.

### 5.1.4 Argument Lifetimes

The lifetime of any input pointer argument passed from the SCE-MI infrastructure into a SCE-MI callback function (such as input ready callback or receive callback) shall be assumed by the application to be limited to the duration of the callback. Once the callback returns, the application cannot assume that such pointer arguments remain valid. So, for example it would lead to undefined behavior for an application receive callback to cache the `SceMiMessageData *` pointer and refer to it at some point in time after the callback returns.

Conversely, the lifetime of any input pointer argument passed from an application into a SCE-MI API call shall be assumed by the SCE-MI infrastructure to be limited to the duration of the API call. Once the API call returns, the infrastructure cannot assume that such pointer arguments remain valid.

### 5.1.5 SCE-MI Compliance

SCE-MI defines three interfaces, namely: macro-based, function-based and pipes-based interfaces. SCE-MI implementation providers must qualify their level of compliance if their implementation does not support all three SCE-MI interfaces.

A SCE-MI implementation that is only compliant with SCE-MI {Macro-based and/or Function-based and/or Pipes-based} interface must be stated as "compliant with SCE-MI {Macro-based and/or Function-based and/or Pipes-based} interface(s) only".

## 5.2 Macro-Based Hardware side interface macros

This section contains the macros that need to be implemented on the hardware side of the interface.

### 5.2.1 Dual-ready protocol

The message port macros on the hardware side use a general PCI-like dual-ready protocol, which is explained in this section. Briefly, the dual-ready handshake works as follows.

The transmitter asserts `TransmitReady` on any clock cycle when it has data and de-asserts when it does not.

The receiver asserts `ReceiveReady` on any cycle when it is ready for data and de-asserts when it is not.

In any clock cycle in which `TransmitReady` and `ReceiveReady` are both asserted, data "moves", meaning it is taken by the receiver.

Note:

1) After a ready request (TransmitReady or ReceiveReady) has been asserted, it cannot be removed until a data transfer has taken place.
2) After TransmitReady has been asserted, the data must be held constant otherwise the result is undefined.

The waveforms in Figure 5.1 depict several dual-ready handshake scenarios.

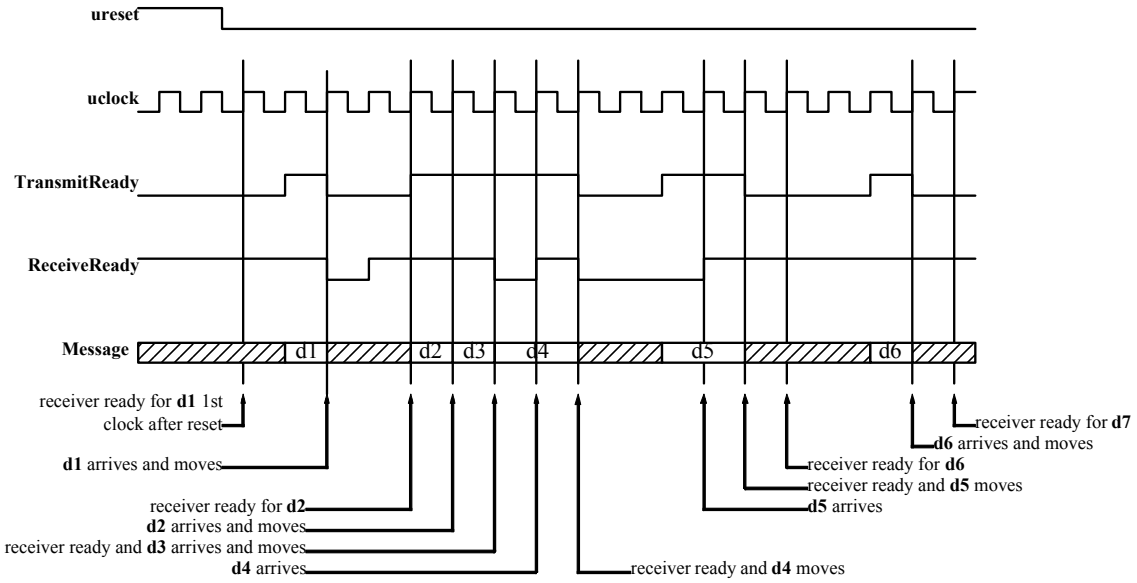


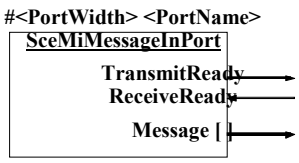

Figure 5.1 Dual-ready handshake protocol

The dual-ready protocol has the following two advantages.

a) Signals are level-based; therefore, they are easily sampled by posedge clocked logic.
b) If both TransmitReady and ReceiveReady stay asserted, sequences of data can still move every clock cycle; therefore, the same performance can be realized as, for example, a toggle-based protocol.

### 5.2.2 SceMiMessageInPort macro

The `SceMiMessageInPort` macro presents messages arriving from the software side of a channel to the transactor. The macro consists of two handshake signals which play a dual-ready protocol and a data bus that presents the message itself. Figure 5.2 shows the symbol for the `SceMiMessageInPort` macro, as well as Verilog and VHDL source code for the empty macro wrappers.




**Verilog Macro Wrapper:**

```
module SceMiMessageInPort(
  //inputs                        outputs
    ReceiveReady,                 TransmitReady,
                                  Message );
  //------------------------      ------------------------------
    parameter PortWidth = 1;
    input ReceiveReady;           output TransmitReady;
                                  output [PortWidth-1:0] Message;
endmodule
```

**VHDL Macro Wrapper:**

```
entity SceMiMessageInPort is
    generic( PortWidth: natural ;)
    port(
       ReceiveReady: in std_logic; TransmitReady: out std_logic;
       Message: out std_logic_vector( PortWidth-1 downto 0 ) );
end;

architecture EmptyMacro of SceMiMessageInPort is begin end;
```

Figure 5.2 SceMiMessageInPort macro

#### 5.2.2.1 Parameters and signals

**PortWidth**

The message width in bits is derived from the setting of this parameter.

**PortName**

The port's name is derived from its instance label.

**TransmitReady**

A value of one (1) on this signal sampled on any `posedge` of the `uclock` indicates the channel has message data ready for the transactor to take. If `ReceiveReady` is not asserted, the `TransmitReady` remains asserted until and during the first clock in which `ReceiveReady` finally becomes asserted. During this clock, data moves and if no more messages have arrived from the software side, the TransmitReady is de-asserted.

**ReceiveReady**

A value of one (1) on this signal indicates the transactor is ready to accept data from the software. By asserting this signal, the hardware indicates to the software that it has a location into which it can put any data that might arrive on the message input port. When a new message arrives, as indicated by the `TransmitReady` and `ReceiveReady` both being true, that location is consumed (see Figure 5.1). When this happens, a notification is sent to the software side that a new empty location is available and this triggers an input-ready callback to occur on the software side. (5.2.2.2 explains in detail when input-ready propagation notifications are done with respect to the timing of the `TransmitReady` and `ReceiveReady` handshakes.)

Transactors do not need to utilize `ReceiveReady` and the input-ready callback. If this is the case, the `ReceiveReady` input needs to be permanently asserted (i.e., "tied high") and, on the software side, no input-ready callback is registered. In this case, `TransmitReady` merely acts as a strobe for each arriving message. The transactor needs to be designed to take any arriving data immediately, as it is not guaranteed to be held for subsequent `uclock` cycles.

**Message**

This vector signal constitutes the payload data of the message.

#### 5.2.2.2 **Input-ready propagation**

The SCE-MI provides a functionality called input-ready propagation. This allows a transactor to communicate (to the software) it is ready to accept new input on a particular channel. When the transactor asserts the `ReceiveReady` input, the `IsReady` callback on that port is called during the next call to the `::ServiceLoop()`.

If the software client code registers an input-ready callback when it first binds to a message input port proxy (see 5.4.3.5), the hardware side of the infrastructure shall notify the software side each time it is ready for more input. Each time it is so notified, the port proxy on the software side makes a call to the user registered input-ready callback. This mechanism is called input-ready propagation.

Input-ready propagation shall happen:

1) On the first rising edge of `uclock` after reset at which `ReceiveReady` is asserted, and
2) On the first rising edge of `uclock` after a message transferred at which `ReceiveReady` is asserted,

when an `IsReady()` callback is registered. Case 1 covers the input-ready propagation for d1 in Figure 5.3. Case 2 covers the others (d2, d3, and d4).

The prototype for the input-ready callback is:

```
void (*IsReady)(void *context);
```

When this function is called, a software model can assume that a message can be sent to the message input port proxy for transmission to the message input port on the hardware side. The context argument can be a pointer to any user-defined object, presumably the software model that bound the proxy.

The application needs to follow the protocol that if the transactor is not ready to receive input, the software model shall not do a send. The software model knows not to send if it has not received an input-ready callback. The SCE-MI infrastructure does not enforce this.

Note: An application can service as many output callbacks as is desired while pending an input callback. In other words, the software model can have an outer loop which checks the status of an application-defined `OKToSend` flag on each iteration and skips the send if the flag is false.
So, suppose an application has an outer loop that repeatedly calls `::ServiceLoop()` and checks for arriving output messages and input-ready notifications. Each callback function sets a flag in the context that the outer loop uses to know if an output message has arrived and needs processing, or an input port needs more input. It is possible that, before an input-ready callback gets called, the outer loop called `::ServiceLoop()` 50 times and each call results in an output message callback and the subsequent processing of that output message. Finally, on the 51'st time `::ServiceLoop()` is called, the input-ready callback is called, which sets the `OKToSend` flag in its context, and then the outer loop detects the new flag status and initiates a send on that input channel.
The handshake waveforms in Figure 5.3 are intended purely to illustrate the semantics of the dual-ready protocol. There can be a couple of reasons why these waveforms might not be realistic in an actual implementation of a `SceMiMessageInPort` macro. The waveforms shown in Figure 5.3 show what typically occurs when input-ready callbacks are enabled. It shows four possible scenarios where an input-ready notification occurs.

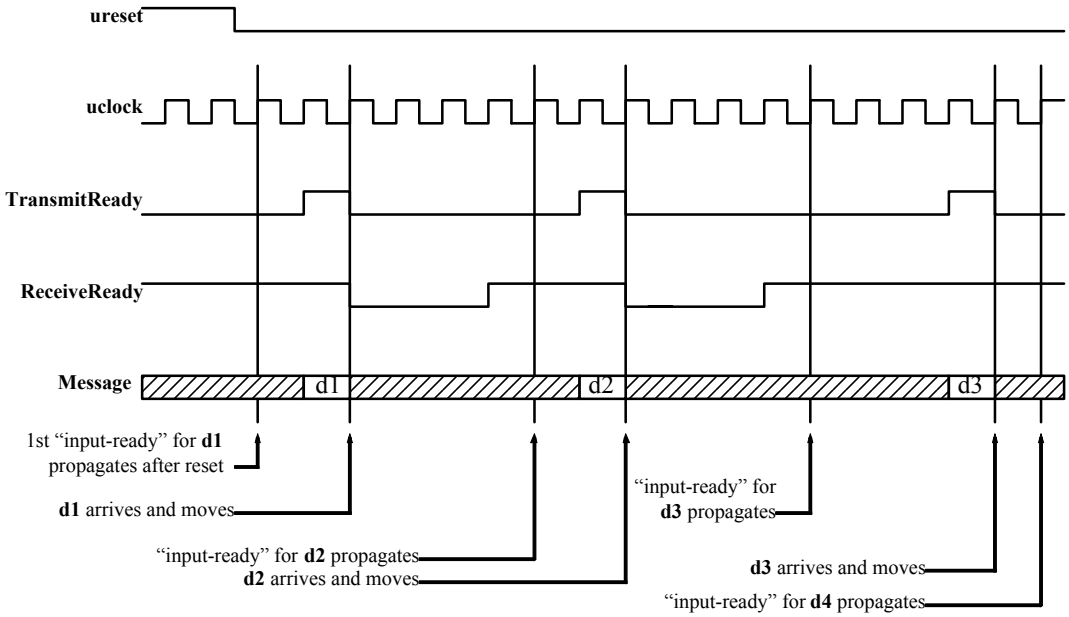



Figure 5.3 SceMiMessageInPort handshake waveforms with input-ready propagation

In the depicted scenarios, an input-ready notification is propagated to the software if:

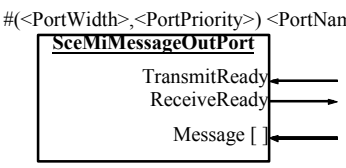
the `ReceiveReady` from a transactor is asserted in the first clock following a reset or

the `ReceiveReady` from a transactor transitions from a 0 to a 1 or

the `ReceiveReady` from a transactor remains asserted in a clock following one where a transfer occurred due to assertions on both `TransmitReady` and `ReceiveReady`.

### 5.2.3 SceMiMessageOutPort macro

The `SceMiMessageOutPort` macro sends messages to the software side from a transactor. Like the `SceMiMessageInPort` macro, it also uses a dual-ready handshake, except in this case, the transmitter is the transactor and the receiver is the SCE-MI interface. A transactor can have any number of `SceMiMessageOutPort` macro instances. Figure 5.4 shows the symbol for the `SceMiMessageOutPort` macro, as well as Verilog and VHDL source code for the empty macro wrappers.




Verilog Macro Wrapper:

```
module SceMiMessageOutPort(
  //inputs                        outputs
    TransmitReady,                ReceiveReady,
    Message );
  //------------------------      ------------------------------
    parameter PortWidth = 1;
    parameter PortPriority;  // Parameter no longer used
    input TransmitReady;          output ReceiveReady;
    input [PortWidth-1:0] Message;
endmodule
```

VHDL Macro Wrapper:

```
entity SceMiMessageOutPort is
    generic( PortWidth: natural; PortPriority: natural := 10 )
    port(
       TransmitReady: in std_logic; ReceiveReady: out std_logi
       Message: in std_logic_vector( PortWidth-1 downto 0 ) );
end;

architecture EmptyMacro of SceMiMessageOutPort is begin end;
```

Figure 5.4 SceMiMessageOutPort macro

#### 5.2.3.1 **Parameters**

**PortWidth**

The message width in bits is derived from the setting of this parameter.

**PortPriority**

The parameter is no longer in use.

**PortName**

The port's name is derived from its instance label.

#### 5.2.3.2 **Signals**

**TransmitReady**

A value of one (1) on this signal indicates the transactor has message data ready for the output channel to take. If `ReceiveReady` is not asserted, the `TransmitReady` shall remain asserted until and during the first clock in which `ReceiveReady` finally becomes asserted. During this clock, data moves and if the transactor has no more messages for transmission, it de-asserts the TransmitReady.

**ReceiveReady**

A value of one (1) on this signal sampled on any `uclock posedge` indicates the output channel is ready to accept data from the transactor. By asserting this signal, the SCE-MI hardware side indicates to the transactor the output channel has a location where it can put any data that is destined for the software side of the channel. In any cycle during which both the `TransmitReady` and `ReceiveReady` are asserted, the transactor can assume the data moved. If, in the subsequent cycle, the `ReceiveReady` remains asserted, this means a new empty location is available which the transactor can load any time by asserting `TransmitReady` again. Meanwhile, the last message data, upon arrival to the software side, triggers a receive callback on its message output port proxy (see 5.4.7.1).

**Message**

This vector signal constitutes the payload data of the message originating from the transactor, to be sent to the software side of the channel.

#### 5.2.3.3 Message Ordering

The idea of ordering message delivery to software arises from the fact that there is a global time order defined in the hardware domain by the order of `cclock` edges. The delivery of messages from hardware to software respects this ordering. In particular, the delivery of messages from hardware to software is ordered using the following rules:

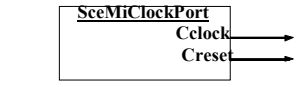
a) Messages from a single message out port are delivered to software in the same time order in which they are delivered to the port.

b) Messages from different ports which complete the dual-ready protocol on different `cclocks` are delivered to software in the time order in which the receive ready signals are asserted. In the case that two message ports accomplish the dual-ready protocol and have data move in the same `cclock` cycle, the order of delivery of the messages to the software is undefined.

### 5.2.4 SceMiClockPort macro

The `SceMiClockPort` macro supplies a controlled clock to the DUT. The `SceMiClockPort` macro is parameterized so each instance of a `SceMiClockPort` fully specifies a controlled clock of a given frequency, phase shift, and duty cycle. The `SceMiClockPort` macro also supplies a controlled reset whose duration is the specified number of cycles of the `cclock`.

Figure 5.5 shows the symbol for the `SceMiClockPort` macro, as well as Verilog and VHDL source code for the empty macro wrappers.




**Verilog Macro Wrapper:**

```
module SceMiClockPort(
  //inputs                        outputs
                                  Cclock, Creset );
  //------------------------      ------------------------------
    parameter ClockNum=1;
    parameter RatioNumerator=1, RatioDenominator=1;
    parameter DutyHi=0,    DutyLo=100,     Phase=0;
    parameter ResetCycles=8;
endmodule
```

**VHDL Macro Wrapper:**

```
entity SceMiClockPort is
    generic( ClockNum:  natural := 1;
        RatioNumerator: natural := 1;  RatioDenominator: natural := 1;
        DutyHi:         natural := 0; DutyLo:           natural := 100;
        Phase:          natural := 0; ResetCycles:      natural := 8 );
    port( Cclock: out std_logic; Creset: out std_logic );
end;

architecture EmptyMacro of SceMiClockPort is begin end;
```

Figure 5.5 SceMiClockPort macro

All of the clock parameters have default values. In simpler systems where only one controlled clock is needed, exactly one instance of a `SceMiClockPort` can be instantiated at the top level with no parameters specified. This results in a single controlled clock with a ratio of 1/1, a don't care duty cycle (see 5.2.4.3), and a phase shift of 0. Ideally, this clock's frequency matches that of the `uclock` during cycles in which it is enabled.

The SCE-MI infrastructure always implicitly creates a controlled clock with a 1/1 ratio, which is the highest frequency controlled clock in the system. Whether or not it is visible to the user's design depends on whether a `SceMiClockPort` with a 1/1 ratio and a don't care duty cycle is explicitly declared (instantiated).

In more complex systems that require multiple clocks, a `SceMiClockPort` instance needs to be created for each required clock. The clock ratio in the instantiation parameters always specifies the frequency of the clock as a ratio relative to the fastest controlled clock in the system (whose ratio is always 1/1).

For example, if a `cclock` is defined with a ratio of 4/1 this is interpreted as, "for every 4 edges of the 1/1 `cclock` there is only 1 edge of this `cclock`". This defines a "divide-by-four" clock.

It is sometimes necessary to establish a timebase which is associated with the fastest clock (the 1/1 clock) in the system. An implementation should provide a mechanism by which this can be done.

#### 5.2.4.1 Parameters and signals

**ClockNum=1**

This parameter assigns a unique number to a clock which is used to differentiate it from other `SceMiClockPort` instances. It shall be an error (by the infrastructure linker) if more than one `SceMiClockPort` instances share the same `ClockNum`. The default `ClockNum` is 1.

**RatioNumerator=1, RatioDenominator=1**

These parameters constitute the numerator and denominator, respectively, of this clock's ratio. The numerator always designates the number of cycles of the fastest controlled clock that occur during the number of cycles of "this" clock specified in the denominator. For example, `RatioNumerator=5` and `RatioDenominator=2` specifies a 5/2 clock, which means for every five cycles of the 1/1 clock that occur, only two cycles of this clock occur. The default clock ratio is 1/1. For more information refer to section 5.2.4.

**DutyHi=0, DutyLo=100, Phase=0**

The duty cycle is expressed with arbitrary integers which are normalized to their sum, such that the sum of `DutyHi` and `DutyLo` represent the number of units for a whole cycle of the clock. For example, when `DutyHi=75` and `DutyLo=25`, the high time of the clock is 75 out of 100 units or 75% of the period. Similarly, the low time would be 25% of the period. The phase shift is expressed in the same units; if `Phase=30`, the clock is shifted by 30% of its period before the first low to high transition occurs.

The default duty cycle shown in the macro wrappers within Figure 5.6 is a don't care duty cycle of 0/100 (see 5.2.4.3).

**ResetCycles=8**

This parameter specifies how many cycles of this controlled clock shall occur before the controlled reset transitions from its initial value of 1 back to 0.

**ClockName**

The clock port's name is derived from its instance label.

**Cclock**

This is the controlled clock signal the SCE-MI infrastructure supplies to the DUT. This clock's characteristics are derived from the parameters specified on instantiation of this macro.

**Creset**

This is the controlled reset signal the SCE-MI infrastructure supplies to the DUT.

#### 5.2.4.2 Deriving clock ratios from frequencies

Another way to specify clock ratios is enter them directly as frequencies, all normalized to the clock with the highest frequency. To specify ratios this way requires the following.

Make each ratio numerator equal to the highest frequency.

Use consistent units for all ratios.

Omit those units and simply state them as integers.

For example, suppose a system has 100Mhz, 25Mhz, and 10Mhz, 7.5 Mhz, and 32kHz clocks. To specify the ratios, the frequencies can be directly entered as integers, using kHz as the unit (but omitting it!):

```
100000 / 100000 - the fastest clock
100000 / 25000
100000 / 10000

100000 / 7500
100000 / 32
```

Users who like to think in frequencies rather than ratios can use this simple technique.

Note: An implementor of the SCE-MI macro-based interface may wish to provide a tool to assist in deriving clock ratios from frequencies. Such a tool could allow a user to enter clock specifications in terms of frequencies and then generate a set of equivalent ratios. In addition, this tool could be used to post process waveforms (such as .vcd files) generated by the simulation so the defined clocks appear in the waveform display to be the exact same frequencies given by the user.

#### 5.2.4.3 Don't care duty cycle

The default duty cycle shown within the macro wrappers in Figure 5.6 is a don't care duty cycle. Users can specify they only care about `posedges` of the `cclock` and do not care where the `negedge` falls. This is known as a `posedge` active don't care duty cycle. In such a case, the `DutyHi` is given as a 0. The `DutyLo` can be given as an arbitrary number of units, such that the Phase offset can still be expressed as a percentage of the whole period (i.e., `DutyHi+DutyLo`).

For example, this combination:

```
DutyHi=0, DutyLo=100, Phase=30
```

means the following:

a) I don't care about the duty cycle. Specifically, I don't care where the `negedge` of the clock falls.

b) If the total period is expressed as 100 units (0+100), the phase should be shifted by 30 of those units. This represents a phase shift of 30%.

Another example:

```
DutyHi=3, DutyLo=1, Phase=2
```

means:

a) I care about both intervals of the duty cycle. The duty cycle is 75%/25%.

b) The phase shift is 50% of period (expressed as 3+1 units).

It is also possible to have a `negedge` active don't care duty cycle. In this case, the `DutyLo` parameter is given as a 0 and the `DutyHi` is given as a positive number (> 0).

For example:

```
DutyHi=1, DutyLo=0, Phase=0
```

means:

a) I don't care about the duty cycle. Specifically, I don't care where the posedge of a clock falls.

b) The phase shift is 0.

In any clock specification, it shall be an error if `Phase >= DutyHi + DutyLo`.

Note: The intent of the don't care duty cycle is to relax the requirement that each edge of a controlled clock must coincide with a rising edge of `uclock`. A controlled clock with a `posedge` active don't care duty cycle, i.e., with `DutyHi` given as 0, is not required to have its falling edge coincide with a rising edge of `uclock`. Similarly, a controlled clock with a `negedge` active don't care duty cycle, i.e., with `DutyLo` given as 0, is not required to have its rising edge coincide with a rising edge of `uclock`. Hence, the don't care duty cycle enables controlled clocks to be the same frequency of the `uclock`. Conversely, the maximum possible frequency of a non-don't care duty cycle controlled clock is 1/2 the frequency of the `uclock`. Since the implicit 1/1 controlled clock is specified to have `posedge` active don't care duty cycle, it may be as fast as `uclock`.

#### 5.2.4.4 Controlled reset semantics

The `Creset` output of the `SceMiClockPort` macro shall obey the following semantics:

`Creset` will start low (de-asserted) and transition to high one or more `uclock` cycles later. It then remains high (asserted) for at least the minimum duration specified by the `ResetCycles` parameter adorning the `SceMiClockPort` macro. This duration is expressed as a number of edges of associated `Cclock`. Following the reset duration, the `Creset` then goes low (de-asserted) and remains low for the remaining duration of the simulation. Some applications require 2-edged resets at the beginning of a simulation.

For multiple `cclocks`, the reset duration shall have a minimum length so it is guaranteed to span the `ResetCycles` parameter of any clock. In other words, the minimum controlled reset duration for all clocks shall be:

```
max( ResetCycles for cclock1, ResetCycles for cclock2, ...)
```

Some implementations can use a reset duration that is larger than the quantity shown above to achieve proper alignment of multiple `cclocks` on the edges of the controlled reset, as described in 5.2.4.5.

During the assertion of `Creset`, `Cclock` edges shall be forced, regardless of the state of the `ReadyForCclock` inputs to the `SceMiClockControl` macros. Once the reset duration completes, the `Cclock` will be controlled by the `ReadyForCclock` inputs.

Note: The operation of controlled reset just described provides the default controlled reset behavior generated by the `SceMiClockPort` macro. If more sophisticated reset handling is required, use a specially written reset transactor in lieu of the simpler controlled resets that come from the `SceMiClockPort` instances. For example, if a software controlled reset is required, an application needs to create a reset transactor which responds to a special software originated reset command that arrives on its message input port.

#### 5.2.4.5 Multiple `cclock` alignment

In general, all `cclocks` need to align on the first rising `uclock` edge following the trailing edge of the `creset`. This `uclock` edge is referred to as the point of alignment. For `cclocks` with phases of 0, this means rising edges of these clocks shall coincide with the point of alignment. For `cclocks` with phases > 0, those edges occur at some time after the point of alignment. Every `cclock` edge must occur on a `uclock` edge.

Figure 5.6 shows an assortment of `cclocks` with the `uclock` and `creset`. It also shows how those `cclocks` behave at the point of alignment.

In Figure 5.6, `cclock1`, `cclock2`, and `cclock3` have phases of 0 and, therefore, have rising edges at the point of alignment. `cclock4` has the same duty cycle as `cclock2`, but a phase shift of 50%. Therefore, its rising edge occurs two `uclocks` (1/2 cycle) after the point of alignment. Its starting value at the point of alignment is still 0.

`cclock5` has the same duty cycle as `cclock3`, but a phase of 50%. Again, its rising edge occurs 1/2 cycle after the point of alignment. But notice its starting value at the point of alignment is 0. This can be alternatively thought of as an inverted phase. Anytime the phase is greater than the high duty cycle interval, the starting value at the point of alignment is a 0. In the case where the phase equals the high duty cycle, a falling edge occurs at the point of alignment.

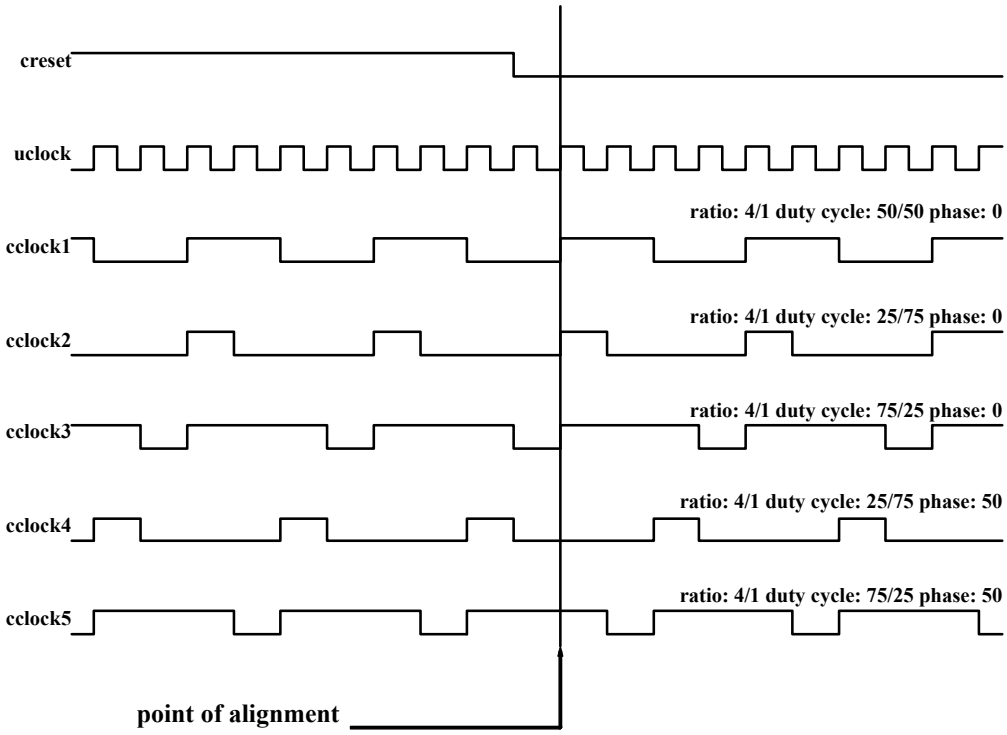
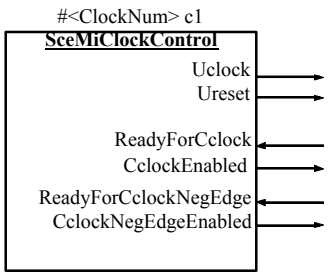



Figure 5.6 Multi-clock alignment

### 5.2.5 SceMiClockControl macro

For every `SceMiClockPort` macro instance there must be at least one counterpart `SceMiClockControl` macro instance presumably encapsulated in a transactor. The `SceMiClockControl` macro is the means by which a transactor can control a DUT's clock and by which the SCE-MI infrastructure can indicate to a transactor on which `uclock` cycles that controlled clock have edges.

Figure 5.7 shows the symbol for the `SceMiClockControl` macro as well as Verilog and VHDL source code for the empty macro wrappers.




Verilog Macro Wrapper:

```
module SceMiClockControl(
  //inputs                          outputs
                                    Uclock, Ureset;
    ReadyForCclock,                 CclockEnabled,
    ReadyForCclockNegEdge,          CclockNegEdgeEnabled );
  //-------------------------      ------------------------------
    parameter ClockNum = 1;
                                    output Uclock, Ureset;
    input ReadyForCclock;           output CclockEnabled;
    input ReadyForCclockNegEdge, output CclockNegEdgeEnabled;
endmodule
```

VHDL Macro Wrapper:

```
entity SceMiClockControl is
    generic( ClockNum: natural := 1 );
    port(
                                Uclock, Ureset: out std_logic;
        ReadyForCclock: in std_logic;
                                CclockEnabled: out std_logic;
        ReadyForCclockNegEdge: in std_logic;
                                 CclockNegEdgeEnabled: out std_lo
        );
end;

architecture EmptyMacro of SceMiClockControl is begin end;
```

Figure 5.7 SceMiClockControl macro

For each `SceMiClockPort` defined in the system, typically one corresponding `SceMiClockControl` macro is instantiated in one or more transactors. If no clock controls are associated with a given controlled clock, it is assumed there is an implicit clock control which is always enabling that clock so the controlled clock simply runs free. In addition to providing uncontrolled clocks and resets, this macro also provides handshakes that provide explicit control of both edges of the generated `cclock`.

#### 5.2.5.1 Parameters

**ClockNum=1**

This is the only parameter given to the `SceMiClockControl` macro. This parameter is used to associate a `SceMiClockControl` instance with its counterpart `SceMiClockPort` instance defined at the top level. The default `ClockNum` is 1.

There shall be exactly one instance of `SceMiClockPort` associated with each instance of `SceMiClockControl` in the system. But there can be one or more instances of `SceMiClockControl` for each instance of `SceMiClockPort`. A `SceMiClockControl` instance identifies its associated `SceMiClockPort` by properly specifying a `ClockNum` parameter matching that of its associated `SceMiClockPort`.

#### 5.2.5.2 Signals

**Uclock**

This is the uncontrolled clock signal generated by the SCE-MI infrastructure.

**Ureset**

This is the uncontrolled reset generated by the SCE-MI infrastructure. This signal is high at the beginning of simulated time and transitions to a low an arbitrary (implementation-dependent) number of `uclocks` later. It can be used to reset the transactor.

The uncontrolled reset shall have a duration spanning that of the longest controlled reset (`Creset` output from each `SceMiClockPort`; see 5.2.4.4) as measured in `uclocks`. This guarantees all DUTs and transactors properly wake up in an initialized state the first `uclock` following expiration of the last controlled reset.

**ReadyForCclock**

This input to the macro indicates to the SCE-MI infrastructure that a transactor is willing to allow its associated DUT clock to advance. One of the most useful applications of this feature is to perform complex algorithmic operations on the data content of a transaction before presenting it to the DUT.

If this input to one of the `SceMiClockControl` instances associated with a given controlled clock is de-asserted, the next `posedge` of that `cclock` will be disabled. In reacting to a `ReadyForCclock` of a slower clock, the infrastructure must not prematurely disable active edges of other faster clocks that occur prior to the last possible `uclock` preceding the edge to be disabled. In other words, that edge is disabled just in time so as to allow faster clock activity to proceed until the last moment possible. Once the edge is finally disabled, all active edges of all controlled clocks are also disabled. This is referred to as just in time clock control semantics.

Note: It may sometimes be desired for a transactor to stop all clocks in the system immediately. This is referred to as emergency brake clock control semantics. This can simply be done by instantiating a `SceMiClockControl` associated with the fastest clock in the system and applying normal clock control to it. See Section 5.2.4 for more information.

**CclockEnabled**

This macro output signals the transactor, that on the next `posedge` of `uclock`, there is a `posedge` of the controlled clock. The transactor can thus sample this signal to know if a DUT clock `posedge` occurs. It can also use this signal as a qualifier that says it is okay to sample DUT output data. Transactors shall only sample DUT outputs on valid controlled clock edges. The SCE-MI infrastructure looks at the `ReadyForCclock` inputs from all the transactors and asserts `CclockEnabled` only if they are all asserted. This means any transactor can stop all the clocks in the system by simply de-asserting `ReadyForCclock`.

For a `negedge` active don't care duty cycle (see 5.2.4.3), since the user does not care about the `posedge`, the `CclockEnabled` shall always be 0.

**ReadyForCclockNegEdge**

Similarly, for `negedge` control, if this input to one of the `SceMiClockControl` instances that are associated with a given controlled clock is de-asserted, the next `negedge` of that clock will be disabled. In reacting to a `ReadyForCclockNegEdge` of a slower clock, the infrastructure must not prematurely disable active edges of other faster clocks that occur prior to the last possible `uclock` preceding the edge to be disabled. In other words, that edge is disabled just in time so as to allow faster clock activity to proceed until the last moment possible. Once the edge is finally disabled, all active edges of all controlled clocks are also disabled. This is referred to as just in time clock control semantics.

Note: Support for explicit `negedge` control is needed for transactors that use the `negedge` of a controlled clock as an active edge. Transactors that do not care about controlling `negedges` (such as the one shown in Figure A.1) need to tie this signal high.

**CclockNegEdgeEnabled**

This signal works like `CclockEnabled`, except it indicates if the `negedge` of a controlled clock occurs on the next `posedge` of the `uclock`. This can be useful for transactors that control double pumped DUTs. Transactors that do not care about `negedge` control can ignore this signal.

For a `posedge` active don't care duty cycle (see 5.2.4.3), since the user does not care about the `posedge`, the `CclockNegEdgeEnabled` shall always be 0.

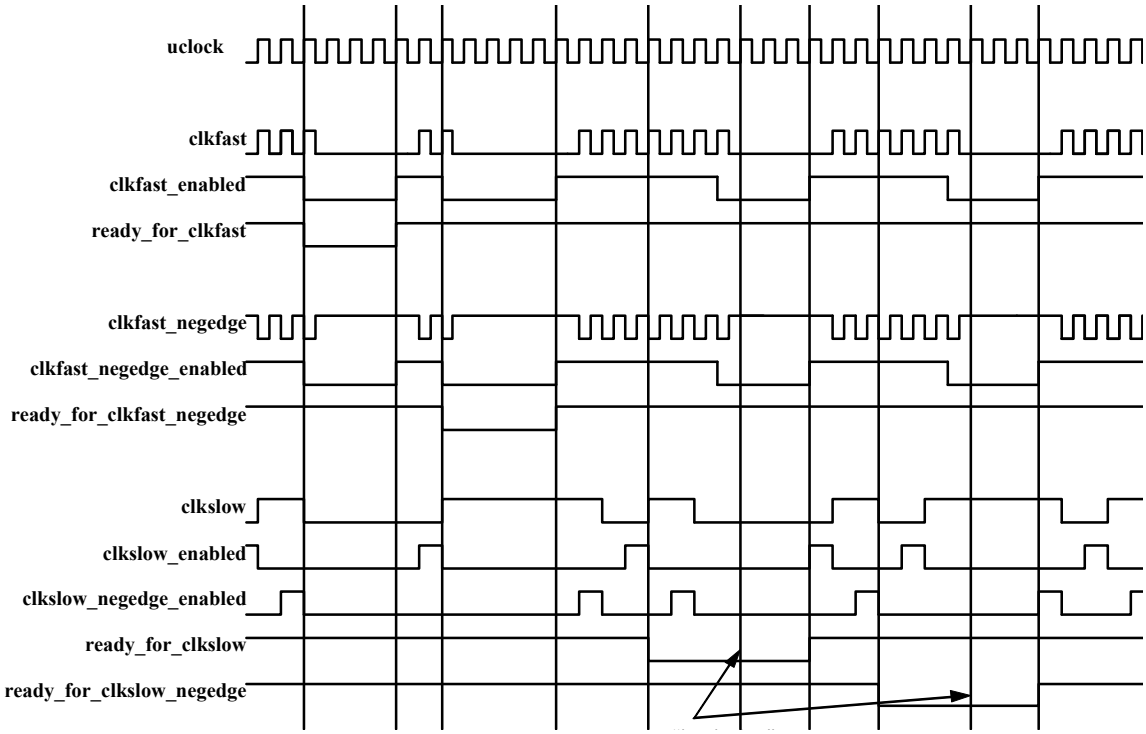



Figure 5.8 Example of Clock Control Semantics

#### 5.2.5.3 Example of Clock Control Semantics

Figure 5.8 shows an example of clock control for two fast clocks (`clkfast`, `clkfast_negedge`) that use don't care duty cycle semantics and one slow clock (`clkslow`) that uses a 50/50 duty cycle. `clkfast` uses `posedge` active don't care duty cycle and `clkfast_negedge` uses `negedge` active don't care duty cycle.

The effect of the 4 respective clock control signals `ready_for_clkfast`, `ready_for_clkfast_negedge`, `ready_for_clkslow`, and `ready_for_clkslow_negedge` can be seen.

De-assertion of `ready_for_clkfast` prevents subsequent `posedges` of `clkfast`, `negedges` of `clkfast_negedge`, and all edges of `clkslow` from occurring on subsequent `posedges` of `uclock`. Once re-asserted, all these edges are allowed to occur on the subsequent `uclock` `posedges` where relevant.

De-assertion of `ready_for_clkfast_negedge` prevents subsequent `negedges` of `clkfast_negedge`, `posedges` of `clkfast`, and all edges of `clkslow` from occurring on subsequent `posedges` of `uclock`. Once re-asserted, all these edges are allowed to occur on the subsequent `uclock` `posedges` where relevant.

De-assertion of `ready_for_clkslow` prevents subsequent `posedges` of `clkslow`. But notice that this happens just in time for the next scheduled `posedge` `clkslow`. Prior to this, edges of faster clocks or the `negedge` of the same clock are allowed to occur. Once the edge is finally disabled, all edges of other clocks are disabled as well. Once re-asserted, all these edges are allowed to occur on the subsequent `uclock` `posedges` where relevant.

De-assertion of `ready_for_clkslow_negedge` prevents subsequent `negedges` of `clkslow`. But notice that this happens just in time for the next scheduled negedge `clkslow`. Prior to this, edges of faster clocks or the posedge of the same clock are allowed to occur. Once the edge is finally disabled, all edges of other clocks are disabled as well. Once re-asserted, all these edges are allowed to occur on the subsequent `uclock` posedges where relevant.

Note: that all of the clock enabled signals, `clkfast_enabled`, `clkfast_negedge_enabled`, `clkslow_enabled`, and `clkslow_negedge_enabled` are shown to transition on `uclock` posedges. The implementation can also choose to

transition them on negedges. The only hard requirement is that their values can be sampled on the `uclock` posedge at which the associated controlled clock edge will occur.

### 5.2.6 SCE-MI 2 support for clock definitions

The SceMiClockPort continues to be supported in SCE-MI 2 and can be used to provide clocks to SCE-MI function and pipe-based models.

Although clock port macros continue to be supported, SCE-MI 2 makes no requirement that clocks must be specified using only clock ports. Alternative clock specifications are allowed such as simple behavioral clock generation blocks that are traditionally used with HDL languages. The SCE-MI 2 standard does not preclude use of such specifications in place of clock port macros.

Additionally, although there are no changes to clock ports for definitions of clocks, it is recognized that with the SCE-MI 2 function and pipes-based approach no clock control is needed as there is no explicit notion of uncontrolled time in SCE-MI 2 models.

Use of the SceMiClockControl macro is only needed for clock control in legacy macro-based transactor models.

## 5.3 Macro-Based Infrastructure linkage

This section is strictly the concern of the infrastructure implementer class of user, as defined in 4.3.3. End-users and transactor implementers can assume the operations described herein are automatically handled by the infrastructure linker.

As described in section 4.5.2, infrastructure linkage is the process which analyzes the user's bridge netlist on the hardware side and compiles it into a form suitable to run on the emulator. This may involve expanding the interface macros into infrastructure components that are added to the existing structure, as well as to generate parameter information which is used to bind the hardware side to the software side. In order to determine this information, the infrastructure linker analyzes the netlist and searches for instances of the SCE-MI hardware side macros, reads the parameter values from those instances, and generates a parameter file that can be read during software side initialization to properly bind message port proxies to the hardware side.

Typically, the infrastructure linker provides options in the form of switches and/or an input configuration file which allows a user to pass along or override implementation-specific options. A well crafted infrastructure linker, however, needs to maximize ease-of-use by transparently providing the end-user with a suitable set of default values for implementation-specific parameters, so that most, if not all, of these parameters need not be overridden.

### 5.3.1 Parameters

The following set of parameters define the minimum set that is needed for all implementations of the SCE-MI standard. Specific implementations might require additional parameters.

**Number** of transactors

The number of transactors shall be derived by counting the number of modules in the user's design that qualify as transactors. Any one of 3 conditions can qualify a module as a transactor:

1. The module has a `SceMiClockControl` macro instantiated immediately inside it, or,

2. The module has the following parameter defined within its scope:

**Verilog:**

```
parameter SceMiIsTransactor = 1;
```

**VHDL:**

```
generic( SceMiIsTransactor: boolean := true );
```

or,

3. The module has at least one SceMi message port instantiated immediately inside it and neither that module nor any of its enclosing parent modules has otherwise been defined as a transactor.

Nested transactors are allowed. A message port's owning transactor is defined to be the lowest module in that port's enclosing hierarchical scope that qualifies as a transactor based on the definition above.

**Transactor name**

The transactor name shall be derived from the hierarchical path name to an instance of a module that qualifies as a transactor (as per the above definition). Naturally, if there are multiple instances of a given type of transactor, they shall be uniquely distinguished by their instance path names. The syntax used to express the path name shall be that of the bridge netlist's HDL language.

**Number of message input or output channels**

The infrastructure linker derives the number of message input and output ports by counting instances of the `SceMiMessageInPort` and `SceMiMessageOutPort` macros.

**Port name**

The name of each port shall be derived from the relative instance path name to that port, relative to its containing transactor module. For example, if the full path name to a message input port macro instance is (using Verilog notation) `Bridge.u1.tx1.ip1` and the transactor name is `Bridge.u1.tx1`, then the port name is `ip1`. If an output port is instantiated one level down from the input port and its full path is `Bridge.u1.tx1.m1.op1`, then its port name is `m1.op1`, since it is instantiated a level down relative to the transactor root level.

The full pathname to a port can be derived by concatenating the transactor name to the port name (with a hierarchical separator inserted between).

**Message input or output port width**

The width of a port in bits shall be derived from the `PortWidth` parameter defined in the message port macro. This width defaults to 1, but is almost always overridden to a significantly larger value at the point of instantiation.

**Number of controlled clocks**

This number shall be derived by counting all instances of the `SceMiClockPort` macro.

**Controlled clock name**

The name of a controlled clock is derived from the instance label (not path name) of its `SceMiClockPort` instance, necessarily instantiated at the top level of the user's bridge netlist and unique among all instances of `SceMiClockPort`.

**Controlled clock ratio**

The clock ratio is determined from the `RatioNumerator` and `RatioDenominator` parameters of the `SceMiClockPort` macro. The `RatioNumerator` designates the number of cycles of the 1/1 controlled clock that occur during the number of cycles of "this" clock specified in `RatioDenominator`. See 5.2.4 for more details about the clock ratio.

**Controlled clock duty cycle and phase**

The duty cycle is determined from the `DutyHi`, `DutyLo`, and Phase parameters of the `SceMiClockPort` macro. The duty cycle is expressed as a pair of arbitrary integers: `DutyHi` and `DutyLo` interpreted as follows: if the sum of `DutyHi` and `DutyLo` represents the number of units in a period of the clock, then `DutyHi` represents the number of units of high time and `DutyLo` represents the number of units of low time. Similarly, Phase represents the number of units the clock is phase shifted relative to the reference 1/1 `cclock`. A user can also specify a don't care duty cycle. See 5.2.4 for more details about the duty cycle and phase.

**Controlled reset cycles**

The duration of a controlled reset expressed in terms of `cclock` cycles is determined from the `ResetCycles` parameter of the `ClockPort` macro.

**Parameter file**

The infrastructure linker needs to automatically generate a parameter file after analyzing the user-supplied netlist and determining all the parameters identified in 5.3.1. The parameter file can be read by the software side of the SCE-MI infrastructure to facilitate binding operations that occur after software model construction. Because it is automatically generated, the content and syntax of the parameter file is left to specific implementers of the SCE-MI. The content itself is not intended to be portable.

However, on the software side, the infrastructure implementer needs to provide a parameter access API that conforms to the specification in 5.4.4. This access block shall support access to a specifically named set of parameters required by the SCE-MI, as well as an optional, implementation specified set of named parameters.

All SCE-MI required parameters are read-only, because their values are automatically determined by the infrastructure linker by analyzing the user-supplied netlist. Implementation-specific parameters can be read-only or read-write as the implementation requires.

# 5.4 Macro-Based Software side interface - C++ API

To gain access to the hardware side of the SCE-MI, the software side shall first initialize the SCE-MI software side infrastructure and then bind to port proxies representing each message port defined on the hardware side. Part of initializing the SCE-MI involves instructing the SCE-MI to load the parameter file generated by the infrastructure linker. The SCE-MI software side can use this parameter file information to establish rendezvous with the hardware side in response to port binding calls from the user's software models. Port binding rendezvous is achieved primarily name association involving transactor names and port names.

Note: Clock names and properties identified in the parameter file are of little significance during the binding process although this information is procedurally available to applications that might need it through the parameter file API (see 5.4.4).

Access to the software side of the interface is facilitated by a number of C++ classes:

```
class SceMiEC
class SceMi
class SceMiMessageInPortProxy
class SceMiMessageOutPortProxy
class SceMiParameters
class SceMiMessageData
```

## 5.4.1 Primitive data types

In addition to C data types, such as `integer`, `unsigned`, and `const char *`, many of the arguments to the methods in the API require unsigned data types of specific width. To support these, SCE-MI implementations need to provide two primitive unsigned integral types: one of exactly 32 bits and the other exactly 64 bits in width. The following example implementation works on most current 32-bit compilers.

Example:

```
typedef unsigned int SceMiU32; //unsigned 32-bit integral type
typedef unsigned long long SceMiU64; //unsigned 64-bit integral type
```

## 5.4.2 Miscellaneous interface issues

In addition to the basic setup, teardown, and message-passing functionality, the SCE-MI provides error handling, warning handling, and memory allocation functionality. These verbatim API declarations are described here.

**Class SceMiEC - error handling**

Most of the calls in the interface take an `SceMiEC * ec` as the last argument. Because the usage of this argument is consistent for all methods, error handling semantics are explained in this section rather than documenting error handling for each method in the API.

Error handling in SCE-MI is flexible enough to either use a traditional style of error handling where an error status is returned and checked with each call or a callback based scheme where a registered error handler is called when an error occurs.

```
enum SceMiErrorType {
 SceMiOK,
 SceMiError
};

struct SceMiEC {
 const char *Culprit;
 const char *Message;
 SceMiErrorType Type;
 int Id;
};

typedef void (*SceMiErrorHandler)(void *context, SceMiEC *ec);

static void
SceMi::RegisterErrorHandler(
 SceMiErrorHandler errorHandler,
 void *context );
```

This method registers an optional error handler with the SCE-MI that is called when an error occurs.

When any SCE-MI operation encounters an error, the following procedure is used:

If the `SceMiEC *` pointer passed into the function was non-NULL, the values of the `SceMiEC` structure are filled out by the errant call with appropriate information describing the error and control is returned to the caller. This can be thought of as a traditional approach to error handling, such as done in C applications. It is up to the application code to check the error status after each call to the API and take appropriate abortive action if an error is detected.

Else if the `SceMiEC *` pointer passed to the function is NULL (or nothing is passed since the default is NULL in each API function) and an error handler was registered, that error handler is called from within the errant API call. The error handler is passed an internally allocated `SceMiEC` structure filled out with the error information. In this error handler callback approach, the user-defined code within the handler can initiate abort operations. If it is a C++ application, a catch and throw mechanism can be deployed. A C application can simply call the `abort()` or `exit()` function after printing out or logging the error information.

Else if the `SceMiEC` * pointer passed to the function is NULL and no error handler is registered, an `SceMiEC` structure is constructed and passed to a default error handler. The default error handler attempts to print a message to the console and to a log file and then calls `abort()`.

This error handling facility only supports irrecoverable errors. This means if an error is returned through the `SceMiEC` object, either via a handler or a return object, there is no point in continuing with the co-modeling session. Any calls that support returning a recoverable error status need to return that status using a separate, dedicated return argument.

Also, the Message text filled out in the error structure is meant to fully describe the nature of the error and can be logged or displayed to the console verbatim by the application error handling code. The Culprit text is the name of the errant API function and can optionally be added to the message that is displayed or logged.

Because every API call returns a success or fatal error status and the detailed nature of errors is fully described within the returned error message, the `SceMiErrorType` enum has only two values pertaining to success: (`SceMiOK`) or failure (`SceMiError`). The `SceMiEC::Type` returned from API functions to the caller can be either of these two values, depending on whether the call was a success or a failure. However the `SceMiEC::Type` passed into an error handler shall, by definition, always have the value `SceMiError`;

otherwise the error handler would not have been called. In addition, the optional Id field can be used to further classify different major error types or tag each distinct error message with a unique integer identifier.

#### 5.4.2.1 Class SceMiIC - informational status and warning handling (info handling)

The SCE-MI also provides a means of conveying warnings and informational status messages to the application. Like error handling, info handling is done with callback functions and a special structure that is used to convey the warning information.

```
enum SceMiInfoType {
 SceMiInfo,
 SceMiWarning,
 SceMiNonFatalError
};

struct SceMiIC {
 const char *Originator;
 const char *Message;
 SceMiInfoType Type;
 int Id;
};

typedef void (*SceMiInfoHandler)(void *context, SceMiIC *ic);

static void
SceMi::RegisterInfoHandler(
 SceMiInfoHandler infoHandler,
 void *context );
```

This method registers an optional info handler with the SCE-MI that is called when a warning or informational status message occurs. This method must only be used for message reporting or logging purposes and must not abort the simulation (unless there is an application error). Only `SceMiEC` error handlers are reserved for that purpose.

When any SCE-MI operation encounters a warning or wishes to issue an informational message, the following procedure is used:

If an info handler was registered, it is called from within the API call that wants to issue the warning. The info handler is passed an internally allocated `SceMiIC` structure filled out with the warning information. In this info handler callback approach, the user-defined code within the handler can convey the warning to the user in a manner that is appropriate for that application. For example, it can be displayed to the console, logged to a file, or both.

Else if no info handler is registered, a `SceMiIC` structure is constructed and passed to a default, implementation-defined error handler. The default error handler can attempt to print a message to the console and/or to a log file in an implementation-specific format.

The Message text filled out in the error structure is meant to fully describe the nature of the info message and can be logged or displayed to the console verbatim by the application's warning and info handling code. The Originator text is the name of the API function that detected the message and can optionally be added to the message that is displayed or logged. The `SceMiInfoType` is an extra piece of information which indicates if the message is a warning or just some informational status.

An additional category, called `SceMiNonFatalError`, can be used to log all error conditions leading up to a fatal error. The final fatal error message shall always be logged using a `SceMiEC` structure and `SceMiErrorHandler` function so an abort sequence is properly handled (see 5.4.2.1). In addition, the info message can optionally be tagged with a unique identifying integer specified in the Id field.

#### 5.4.2.2 Memory allocation semantics

The following rules apply to SCE-MI memory allocation semantics.

Anything constructed by the user is the user's responsibility to delete.

Anything constructed by the API is the API's responsibility to delete.

Thus any object, such as `SceMiMessageData`, that is created by the application using that object's constructor, shall be deleted by the application when it is no longer in use. Some objects, such as `SceMiMessage[In/Out]PortProxy` objects, are constructed by the API and then handed over to the application as pointers. Those objects shall not be deleted by the application. Rather, they are deleted when the entire interface is shut down during the call to `SceMi::ShutDown()`.

Similarly, non-NULL `SceMiEC` structures that are passed to functions are assumed to be allocated and deleted by the application. If a NULL `SceMiEC` pointer is passed to a function and an error occurs, the API allocates the structure to pass to the error handler and, therefore, is responsible for freeing it.

### 5.4.3 Class SceMi - SCE-MI software side interface

This is the singleton object that represents the software side of the SCE-MI infrastructure itself. Global interface operations are performed using methods of this class.

#### 5.4.3.1 Version discovery

```
static int
SceMi::Version(
     const char *versionString );
```

This method allows an application to make queries about the version prior to initializing the SCE-MI that gives it its best chance of specifying a version to which it is compatible. A series of calls can be made to this function until a compatible version is found. With each call, the application can pass version numbers corresponding to those it knows and the SCE-MI can respond with a version handle that is compatible with the queried version. This handle can then be passed onto the initialization call described in 5.4.3.2.

If the given version string is not compatible with the version of the SCE-MI used as the interface, a -1 is returned. At this point, the application has the option of aborting with a fatal error or attempting other versions it might also know how to use.

This process is sometimes referred to as mutual discovery.

**versionString**

This argument is of the form "`<majorNum>.<minorNum>.<PatchNum>`" and can be obtained by the application code from the header file of a particular SCE-MI installation.

The following macros are defined

```
#define SCEMI_MAJOR_VERSION 2
#define SCEMI_MINOR_VERSION 1
#define SCEMI_PATCH_VERSION 0

#define SCEMI_VERSION_STRING "2.1.0"
```

Note: the version mapping shown above is for example purposes only and should always be set to match the actual version of the document that the implementation adheres to.

#### 5.4.3.2 Initialization

```
static SceMi *
SceMi::Init(
 int version,
 SceMiParameters *parameters,
 SceMiEC *ec=NULL );
```

This call is the constructor of the SCE-MI interface. It gives access to all the other global methods of the interface.

The return argument is a pointer to an object of class `SceMi` on which all other methods can be called.

**version**

This input argument is the version number returned by the `::Version()` method described in 5.4.3.1. An error results if the version number is not compatible with the SCE-MI infrastructure being accessed.

**parameters**

This input argument is a pointer to the parameter block object (`class SceMiParameters`) initialized from the parameter file generated by the infrastructure linker. See 5.4.4 for a description of how this object is obtained.

#### 5.4.3.3 SceMi Object Pointer Access

```
static SceMi *
SceMi::Pointer(
 SceMiEC *ec=NULL );
```

This accessor returns a pointer to the `SceMi` object constructed in a previous call to `SceMi::Init`. The return argument is a pointer to an object of class `SceMi` on which all other methods can be called.

If the `SceMi::Init` method has not yet been called, `SceMi::Pointer` will return NULL.

#### 5.4.3.4 Shutdown

```
static void
SceMi::Shutdown(
 SceMi *sceMi,
 SceMiEC *ec=NULL );
```

This is the destructor of the SCE-MI infrastructure object which shall be called when connection to the interface needs to be terminated. This call is the means by which graceful decoupling of the hardware side and the software side is achieved. Termination (`Close()`) callbacks registered by the application are also called during the shutdown process.

#### 5.4.3.5 Message input port proxy binding

```
SceMiMessageInPortProxy *
SceMi::BindMessageInPort(
 const char *transactorName,
 const char *portName,
 const SceMiMessageInPortBinding *binding = NULL,
 SceMiEC *ec=NULL );
```

This call searches the list of input ports learned from the parameter file, which is generated during infrastructure linkage, for one whose names match the `transactorName` and `portName` arguments. If one is found, an object of class `SceMiMessageInPortProxy` is constructed to serve as the proxy interface to that port and the pointer to the constructed object is returned to the caller to serve all future accesses to that port. It shall be an error if no match is found.

The implementation shall copy the contents of the object pointed to by the binding argument, to an internal implementation specific location.

Note: The application is free to de-allocate and/or modify the binding object at any time after calling message input port proxy binding. Since the binding object is copied, the binding itself will not change as a result of this.

**transactorName, portName**

These arguments uniquely identify a specific message input port in a specific transactor on the hardware side to which the caller wishes to bind. These names need to be the path names (described in 5.3.1) expressed in the hardware side bridge's netlist HDL language syntax.

**binding**

The binding argument is a pointer to an object, defined as follows:

```
struct SceMiMessageInPortBinding {
    void *Context;
    void (*IsReady)(void *context);
    void (*Close)(void *context);
};
```

whose data members are used for the following:

**Context**

The application is free to use this pointer for any purposes it wishes. Neither class `SceMi` nor class `SceMiMessageInPortProxy` interpret this pointer, other than to store it and pass it when calling either the `IsReady()` or `Close()` callbacks.

**IsReady()**

This is the function pointer for the callback used whenever an input-ready notification has been received from the hardware side. This call signals that it is okay to send a new message to the input port. If this pointer is given as a NULL, the SCE-MI assumes this port does not need to deploy input-ready notification on this particular channel. See 5.2.2.2 for a detailed description of the input-ready callback.

**Close()**

This is a termination callback function pointer. It is called during destruction of the SCE-MI. This pointer can also be optionally specified as NULL.

If the binding argument is given as a NULL, the SCE-MI assumes that each of the Context, `IsReady()`, and `Close()` data members all have NULL values.

Note: This call

```
inProxy = scemi->BindMessageInPort("Transactor","Port");
```

is equivalent to this code

```
SceMiMessageInPortBinding inBinding;

inBinding.Context = NULL;
inBinding.IsReady = NULL;
inBinding.Close = NULL;

inProxy = scemi->BindMessageInPort("Transactor", "Port",&inBinding);
```

#### 5.4.3.6 Message output port proxy binding

```
SceMiMessageOutPortProxy *
SceMi::BindMessageOutPort(
 const char *transactorName,
 const char *portName,
 const SceMiMessageOutPortBinding *binding,
 SceMiEC *ec=NULL );
```

This call searches the list of output ports learned from the parameter file, which was generated during infrastructure linkage, for one whose names match the `transactorName` and `portName` argument. If one is found, an object of class `SceMiMessageOutPortProxy` is constructed to serve as the proxy interface to that port and the handle to the constructed object is returned to the caller to serve all future accesses to that port. It shall be an error if no match is found.

The implementation shall copy the contents of the object pointed to by the binding argument to an internal, implementation specific location.

Note: The application is free to de-allocate and/or modify the binding object at any time after calling message output port proxy binding. Since the binding object is copied, the binding itself will not change as a result of this.

**transactorName, portName**

These arguments uniquely identify a specific message output port in a specific transactor on the hardware side to which the caller wishes to bind. These names must be the path names (described in 5.3.1) expressed in the hardware side bridge's netlist HDL language syntax.

**binding**

The binding argument is a pointer to an object, defined as follows:

```
struct SceMiMessageOutPortBinding {
    void *Context;
    void (*Receive)(
        void *context,
        const SceMiMessageData *data);
    void (*Close)(void *context);
    };
```

whose data members are used for the following:

**Context**

The application is free to use this pointer for any purposes it wishes. Neither class `SceMi` nor class `SceMiMessageOutPortProxy` interpret this pointer other than to store it and pass it when calling either the `IsReady()` or `Close()` callbacks.

**Receive()**

This is the function pointer for the receive callback used whenever an output message arrives on the port. If this function pointer is set to NULL, it indicates that any messages from the output port should be ignored. See 5.4.7.1 for more information about how receive callbacks process output messages.

**Close()**

This is a termination callback function pointer. It is called during destruction of the SCE-MI. This pointer can also be optionally specified as NULL.

### 5.4.3.7 Service loop

```
typedef int (*SceMiServiceLoopHandler)( void *context, bool pending );

int
SceMi::ServiceLoop(
 SceMiServiceLoopHandler g=NULL,
 void *context=NULL,
 SceMiEC *ec=NULL );
```

This is the main workhorse method that yields CPU processing time to the SCE-MI. In both single-threaded and multi-threaded environments, calls to this method allow the SCE-MI to service all its port proxies, check for arriving messages or messages which are pending to be sent, and dispatch any input-ready or receive callbacks that might be needed. The underlying transport mechanism that supports the port proxies needs to respond in a relatively timely manner to messages queued on the input or output port proxies. Since these messages cannot be handled until a call to `::ServiceLoop()` is made, applications need to call this function frequently.

The return argument is the number of service requests that arrived from the HDL side and were processed since the last call to `::ServiceLoop()`.

The `::ServiceLoop()` first checks for any pending input messages to be sent and sends them.

**g()**

If g is NULL, `::ServiceLoop()` checks for pending service requests and dispatches them, returning immediately afterwards. If `g()` is non-NULL, `::ServiceLoop()` enters a loop of checking for pending service requests, dispatching them, and calling `g()` for each service request. A service request is defined to be one of the following:

An arriving message in a SCE-MI message output port that will result in a receive callback being called.

An input ready notification that will result in an input ready callback being called.

When g() returns 0, control returns from the loop. When g() is called, it is passed a pending flag of 1 or 0 indicating whether or not there is at least one service request pending.

**context**

The context argument to `::ServiceLoop` is passed as the context argument to `g()`.

The following pseudo code illustrates implementation of the `::ServiceLoop()` according to the semantics described above:

```
int SceMi::ServiceLoop(
 SceMiServiceLoopHandler g, void* context, SceMiEC* ec)
{
 bool exit_service_loop = false;
 int service_request_count = 0;
 while( input messages pending ) Send them to HDL side.
 while( exit_service_loop == false ) {
       if( input ready notifications pending ){
              Dispatch input ready callback;
              service_request_count++;
              if( g != NULL && g(context, 1) == 0 )
                     exit_service_loop = true;
       }
       else if( output messages pending ){
              Dispatch message to appropriate receive callback.
              service_request_count++;
              if (g != NULL && !g(context, 1))
                     exit_service_loop = true;
       }
       // if( g is not specified ) We kick out of the loop.
       // else we stay in as long as g returns non-zero.
       else if (g == NULL || g(context, 0) == 0)
              exit_service_loop = true;
 }
 return service_request_count;
}
```

#### 5.4.3.7.1 Example of using the g() function to return on each call to ::ServiceLoop()

There are several different ways to use the `g()` function.

Some applications do force a return from the `::ServiceLoop()` call after processing each message. The `::ServiceLoop()` call always guarantees a separate call is made to the g() function for each message processed. In fact, it is possible to force `::ServiceLoop()` to return back to the application once per message by having the `g()` function return a 0.

So even if all `g()` does is return 0, as follows,

```
int g( void */*context*/, bool /*pending*/ ){ return 0; }
```

the application forces a return from `::ServiceLoop()` for each processed message.

Note: In this case, the `::ServiceLoop()` does not block because it also returns even if no message was found (i.e., `pending == 0` ). Basically `::ServiceLoop()` returns no matter what in this case with zero or one message.

#### 5.4.3.7.2 Example of using the `g()` function to block `::ServiceLoop()` until exactly one message occurs

An application can use the `g()` function to put `::ServiceLoop()` into a blocking mode rather than its default polling mode. The `g()` function can be written to cause `::ServiceLoop()` to block until it gets one message, then return on the message it received. This is done by making use of the pending argument to the `g()` function. This argument simply indicates if there is a message to be processed or not, for example:

```
int g( void */*context*/, bool pending ){
 return pending == true ? 0 : 1 }
```

This blocks until a message occurs, then returns on processing the first message.

#### 5.4.3.7.3 Example of using the g() function to block ::ServiceLoop() until at least one message occurs

Alternatively, suppose the application wants `::ServiceLoop()` to block until at least one message occurs, then return only after all the currently pending messages have been processed.

To do this, the application can define a `haveProcessedAtLeast1Message` flag as follows:

```
int haveProcessedAtLeast1Message = 0;
```

Call `::ServiceLoop()` giving the `g()` function and this flag's address as the context:

```
...
haveProcessedAtLeast1Message = 0;
sceMi->ServiceLoop( g, &haveProcessedAtLeast1Message );
...
```

Now define the `g()` function as follows:

```
int g( void *context, bool pending ){
 int *haveProcessedAtLeast1Message = (int *)context;
 if( pending == 0 )

     // If no more messages, kick out of loop if at least
     // one previous message has been processed, otherwise
     // block until the first message arrives.
     return *haveProcessedAtLeast1Message ? 0 : 1;
 else {
     *haveProcessedAtLeast1Message = 1;
     return 1;
 }
}
```

In conclusion, depending on precisely what type of operation of `::ServiceLoop()` is desired, the `g()` function can be tailored accordingly.

## 5.4.4 Class SceMiParameters - parameter access

This class provides a generic API which can be used by application code to access the interface parameter set described in 5.3.1. It is basically initialized with the contents of the parameter file generated during infrastructure linkage. It provides accessors that facilitate the reading and possibly overriding of parameters and their values.

All SCE-MI required parameters are read-only, because their values are automatically determined by the infrastructure linker analyzing the user-supplied netlist. Implementation-specific parameters can be read-only or read- write as required by the implementation. All parameters in a `SceMiParameters` object shall be overridden before that object is passed to the `SceMi::Init()` call to construct the interface (see 5.4.3.2 Overriding parameters afterwards has no effect.

#### 5.4.4.1 Parameter set

While the format of the parameter file is implementation-specific, the set of parameters required by the SCE-API and the methods used to access them shall conform to the specifications described in this section. For purposes of access, the parameter set shall be organized as a database of attributed objects, where each object instance is decorated with a set of attributes expressed as name/value pairs. There can be zero or more instances of each object kind. The API shall provide a simple accessor to return the number of objects of a given kind, and read and write accessors (described in Table 5.1) to allow reading or overriding attribute values of specific objects.

The objects in the database are composed of the set of necessary interfacing components that interface the SCE- MI infrastructure to the application. For example, there is a distinct object instance for each message port

and a distinct object instance representing each defined clock in the system. Attributes of each of the objects then represent, collectively, the parameters that uniquely characterize the dimensions and constitution of the interface components needed for a particular application.

So, for example, a system that requires one input port, two output ports, and two distinct clocks is represented with five objects, parameterized such that each port object has name and width attributes, each clock object has ratio and duty cycle attributes, etc. These objects and their attributes precisely and fully describe the interfacing requirements between that application and the SCE-MI infrastructure.

Table 5.1 gives the minimal, predefined set of objects and attributes required by the SCE-MI. Additional objects and attributes can be added by implementations. For example, there can be a single, implementation-specific object representing the entire SCE-MI infrastructure facility itself. The attributes of this singleton object can be the set of implementation-specific parameters an implementer of the SCE-MI needs to allow the user to specify.

For more details on attribute meanings, see 5.3.1.

| **Object kind** | **Attribute name** | **Attribute value type** | **Meaning** |
|---|---|---|---|
| MessageInPort | TransactorName | String | Name of the transactor enclosing the message input port. |
| | PortName | String | Name of the message input port. |
| | PortWidth | Integer | Width of the message input port in bits. |
| MessageOutPort | TransactorName | String | Name of the transactor enclosing the message output port. |
| | PortName | String | Name of the message output port. |
| | PortWidth | Integer | Width of the message output port in bits. |
| Clock | ClockName | String | Name of the clock. |
| | RatioNumerator | Integer | Numerator (“fast” clock cycles) of clock ratio. |
| | RatioDenominator | Integer | Denominator (“this” clock cycles) of clock ratio. |
| | DutyHi | Integer | High cycle percentage of duty cycle. |
| | DutyLo | Integer | Low cycle percentage of duty cycle. |
| | Phase | Integer | Phase shift as percentage of duty cycle. |
| | ResetCycles | Integer | Number of controlled clock cycles of reset. |
| ClockBinding | TransactorName | String | Name of the transactor that contributes to the control of this clock. |
| | ClockName | String | Name of the clock that this transactor helps control. |

**Table 5.1: Minimum set of predefined objects and attributes, continued**

For simplicity, values can be signed integer or string values. More complex data types can be derived by the application code from string values. Each attribute definition of each object kind implies a specific value type.

5.4.4.2 Parameter set semantics

Although the accessors provided by the `SceMiParameters` class directly provide the information given in Table 1, other implied parameters can be easily derived by the application. Following are some of the implied parameters and how they are determined:

ClockBinding objects indicate the total number of transactor - clock control macro combinations. The number of distinct contributors to the control of a given clock, as well as the number of distinct transactors in the system, can be ascertained via the ClockBinding objects.

The number of transactors in the system is determined by counting the number of distinct TransactorName’s encountered in the ClockBinding objects.

The number of controlled clocks is determined by reading the number of Clock objects (using the `::NumberOfObjects()` accessor described below).

The number of input and output ports is determined by reading the number of MessageInPort and MessageOutPort objects, respectively.

In addition, the following semantics characterize the parameter set.

a) Transactor names are absolute hierarchical path names and shall conform to the bridge's netlist HDL language syntax.

b) Port names are relative hierarchical path names (relative to the enclosing transactor) and shall conform to the bridge's netlist HDL language syntax.

c) Clock names are identifiers, not path names, and shall conform to the bridge's netlist HDL language identifier naming syntax.

#### 5.4.4.3 Constructor

```
SceMiParameters::SceMiParameters(
 const char *paramsFile,
 SceMiEC *ec=NULL );
```

The constructor constructs an object containing all the default values of parameters and then overrides them with any settings it finds in the specified parameter file. All parameters, whether specified by the user or not shall have default values. Once constructed, parameters can be further overridden procedurally.

**paramsFile**

This is the name of the file generated by the infrastructure linker which contains all the parameters derived from the user's hardware side netlist. This name can be a full pathname to a file or a pathname relative to the local directory.

#### 5.4.4.4 Destructor

```
SceMiParameters::~SceMiParameters()
```

This is the destructor for the parameters object.

#### 5.4.4.5 Accessors

```
unsigned int
SceMiParameters::NumberOfObjects(
 const char *objectKind,
 SceMiEC *ec=NULL ) const;
```

This accessor returns the number of instances of objects of the specified `objectKind` name.

```
int
SceMiParameters::AttributeIntegerValue(
 const char *objectKind,
 unsigned int index,
 const char *attributeName,
 SceMiEC *ec=NULL ) const;
const char *
SceMiParameters::AttributeStringValue(
 const char *objectKind,
 unsigned int index,
 const char *attributeName,
 SceMiEC *ec=NULL ) const;
```

The implementation guarantees the pointer is valid until `Shutdown()` is called for read-only attributes. For non- read-only attributes, the implementation guarantees the pointer is valid until `Shutdown()` or `OverrideAttributeStringValue()` of the attribute whichever comes first.

Note: If the application needs the string value for an extended period of time, it may copy the string value to a privately managed memory area.

These two accessors read and return an integer or string attribute value.

```
void
SceMiParameters::OverrideAttributeIntegerValue(
 const char *objectKind,
 unsigned int index,
 const char *attributeName,
 int value,
 SceMiEC *ec=NULL );

void
SceMiParameters::OverrideAttributeStringValue(
 const char *objectKind,
 unsigned int index,

 const char *attributeName,
 const char *value,
 SceMiEC *ec=NULL );
```

These two accessors override an integer or string attribute value. It shall be an error to attempt to override any of the object attributes shown in Table 1, any implementation-specific attributes designated as read-only or any attribute that is not already in the parameter database.

The following argument descriptions generally apply to all the accessors shown above.

**objectKind**

Name of the kind of object for which an attribute value is being accessed. It shall be an error to pass an unrecognized `objectKind` name to any of the accessors.

**index**

Index of the instance of the object for which an attribute value is being accessed. It shall be an error if the index >= the number returned by the `::NumberOfObjects()` accessor.

**attributeName**

Name of the attribute whose value is being read or overwritten. It shall be an error if the `attributeName` does not identify one of the attributes allowed for the given `objectKind`.

**value**

Returned or passed in value of the attribute being read or overridden respectively. Two overloaded variants of each accessor are provided: one for string values and one for integer values.

### 5.4.5 Class SceMiMessageData - message data object

The class `SceMiMessageData` represents the vector of message data that can be transferred from a `SceMiMessageInPortProxy` on the software side to its associated `SceMiMessageInPort` on the hardware side or from a `SceMiMessageOutPort` on the hardware side to its associated `SceMiMessageOutPortProxy` on the software side. The message data payload is represented as a fixed-length array of SceMiU32 data words large enough to contain the bit vector being transferred to or from the hardware side message port. For example, if the message port had a width of 72 bits, Figure 5.9 shows how those bits are organized in the data array contained inside the `SceMiMessageData` object.

SceMiMessage[In/Out]Port.Message[] bits:

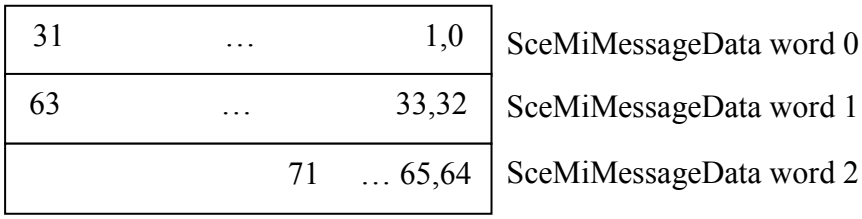



Figure 5.9 Organizing 72 bits in a data array

#### 5.4.5.1 Constructor

```
SceMiMessageData::SceMiMessageData(
 const SceMiMessageInPortProxy &messageInPortProxy,
 SceMiEC *ec=NULL );
```

This constructs a message data object whose size matches the width of the specified input port. The constructed message data object can only be used for sends on that port (or another of identical size) or an error will result.

**Destructor**

```
SceMiMessageData::~SceMiMessageData()
```

This destructs the object and frees the data array.

#### 5.4.5.2 Accessors

```
unsigned int
SceMiMessageData::WidthInBits() const;
This returns the width of the message in terms of number of bits.
unsigned int
SceMiMessageData::WidthInWords() const;
```

This returns the size of the data array in terms of number of `SceMiU32` words.

```
void
SceMiMessageData::Set( unsigned int i, SceMiU32 word, SceMiEC *ec = NULL );
```

This sets word element i of the array to word.

```
void
SceMiMessageData::SetBit( unsigned int i, int bit, SceMiEC *ec = NULL );
```

This sets bit element i of the message vector to 0 if bit == 0, otherwise to 1. It is an error if `i >= ::WidthInBits()`.

```
void
SceMiMessageData::SetBitRange(
    unsigned int i, unsigned int range, SceMiU32 bits, SceMiEC *ec = NULL );
```

This sets range bit elements whose LSB's start at bit element i of the message vector to the value of bits. It is an error if `i+range >= ::WidthInBits()`.

```
SceMiU32
SceMiMessageData::Get( unsigned int i, SceMiEC *ec = NULL ) const;
This returns the word at slot i in the array. It is an error if i >=
::WidthInWords().
int
SceMiMessageData::GetBit( unsigned int i, SceMiEC *ec = NULL ) const;
```

This returns the value of bit element i in the message vector. It is an error if `i >= ::WidthInBits()`.

```
SceMiU32
SceMiMessageData::GetBitRange( unsigned int i, unsigned int range, Sce¬MiEC *ec
```

```
= NULL ) const;
```

This returns the value of range bit elements whose LSB's start at `i` of the message vector. It is an error if `i+range >= ::WidthInBits()`.

```
SceMiU64
SceMiMessageData::CycleStamp() const;
```

The SCE-MI supports a feature called cycle stamping. Each output message sent to the software side is stamped with the number of cycles of the 1/1 controlled clock since the end of `creset` at the time the message is accepted by the infrastructure. The cycle stamp shall be 0 while `creset` is asserted and 1 at the point of alignment. This is shown diagrammatically in Figure 5.10. The cycle stamp provides a convenient way for applications to keep track of elapsed cycles in their respective transactors as the simulation proceeds. The returned value is an absolute, 64-bit unsigned quantity. For more information on the point of alignment, refer to 5.2.4.5.

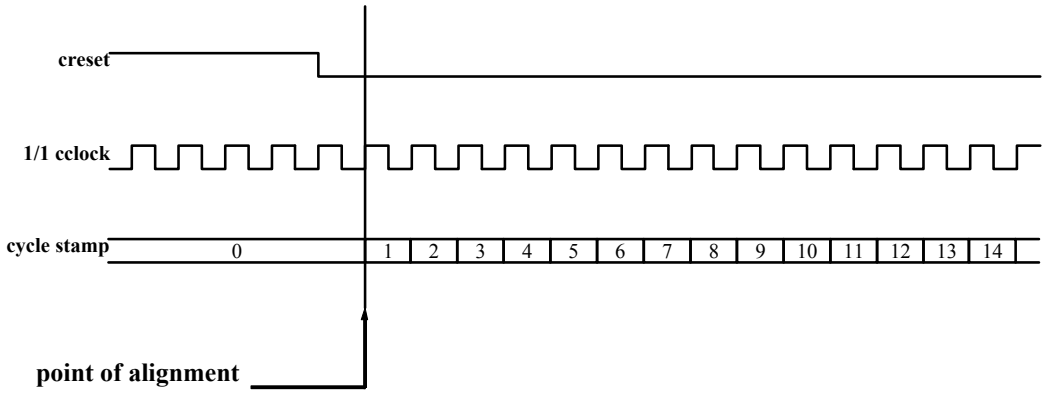



Figure 5.10 Cycle Stamps

Note: It is suggested that messages should not be sent during the reset period. If they are sent they will all have a cycle stamp of zero irrespective of the actual clock cycle that they occur on.

## 5.4.6 Class SceMiMessageInPortProxy

The class `SceMiMessageInPortProxy` presents to the application a proxy interface to a transactor message input port.

### 5.4.6.1 Sending input messages

```
void
SceMiMessageInPortProxy::Send(
 const SceMiMessageData &data,
 SceMiEC *ec=NULL );
```

This method sends a message to the message input channel. This message appears on the hardware side as a bit vector presented to the transactor via the `SceMiMessageInPort` macro (see 5.2.2), instance-bound to this proxy.

**data**

This is a message data object containing the message to be sent. This object may be arbitrarily modified after `Send()` and used for an arbitrary number of sends to the same and other message ports.

#### 5.4.6.2 Replacing port binding

```
void ReplaceBinding(
 const SceMiMessageInPortBinding* binding = NULL,
 SceMiEC* ec=NULL );
```

This method replaces the `SceMiMessageInPortBinding` object originally furnished to the `SceMi::BindMessageInPortProxy()` call that created this port proxy object (see 5.4.3.5). This can be useful for replacing contexts or input-ready callback functions some time after the input message port proxy has been established.

The implementation shall copy the contents of the object pointed to by the binding argument to an internal, implementation specific location.

Note: The application is free to deallocate and/or modify the binding object at any time after calling replace port binding. Since the binding object is copied, the binding itself will not change as a result of this.

**binding**

This is new callback and context information associated with this message input port proxy.

If the binding argument is given as a NULL, the SCE-MI assumes that each of the Context, `IsReady()`, and `Close()` data members have NULL values.

Note: The `ReplaceBinding()` call below

```
SceMiMessageInPortProxy *inProxy;

// ...
inProxy->ReplaceBinding();
```

is equivalent to this code

```
SceMiMessageInPortProxy *inProxy;

// ...

SceMiMessageInPortBinding inBinding;

inBinding.Context = NULL;
inBinding.IsReady = NULL;
inBinding.Close = NULL;

inProxy->ReplaceBinding(&inBinding);
```

#### 5.4.6.3 Accessors

```
const char *
SceMiMessageInPortProxy::TransactorName() const;
```

This method returns the name of the transactor connected to the port. This is the absolute hierarchical path name to the transactor instance expressed in the netlist's HDL language syntax.

```
const char *
SceMiMessageInPortProxy::PortName() const;
```

This method returns the port name. This is the path name to the `SceMiMessageInPort` macro instance relative to the containing transactor netlist's HDL language syntax.

```
unsigned
SceMiMessageInPortProxy::PortWidth() const;
```

This method returns the port width. This is the value of the `PortWidth` parameter that was passed to the associated `SceMiMessageInPort` instance on the hardware side.

#### 5.4.6.4 Destructor

There is no public destructor for this class. Destruction of all message input ports shall automatically occur when the `SceMi::ShutDown()` function is called.

### 5.4.7 Class SceMiMessageOutPortProxy

The class `MessageOutPortProxy` presents to the application a proxy interface to the transactor message output port.

#### 5.4.7.1 Receiving output messages

There are no methods on this object specifically for reading messages that arrive on the output port proxy. Instead, that operation is handled by the receive callbacks. Receive callbacks are registered with an output port proxy when it is first bound to the channel (see 5.4.3.6). The prototype for the receive callback is:

```
void (*Receive)( void *context, const SceMiMessageData *data );
```

When called, the receive callback is passed a pointer to a class `SceMiMessageData` object (see 5.4.5), which contains the content of the received message, and the context pointer. The context pointer is typically a pointer to the object representing the software model interfacing to the port proxy.

Use this callback to process the data quickly and return as soon as possible. The reference to the `SceMiMessageData` is of limited lifetime and ceases to exist once the callback returns and goes out of scope. Typically in a SystemC context, the callback does some minor manipulation to the context object, then immediately returns and lets a suspended thread resume and do the main processing of the received transaction.

No `SceMiEC *` error status object is passed to the call, because if an error occurs within the `SceMi::ServiceLoop()` function (from which the receive callback is normally called), the callback is never called and standard error handling procedures (see 5.4.2.1) are followed by the service loop function itself. If an error occurs inside the receive callback, by implication it is an application error, not an SCE-MI error, and thus is the application's responsibility to handle (perhaps setting a flag in the context object before returning from the callback).

It shall be an error if the class `SceMiMessageData` object passed to the receive callback is passed as the class `SceMiMessageData` argument of the `SceMiMessageInPortProxy::Send()` method. Modifying the class `SceMiMessageData` object by casting away const leads to undefined behavior. This is in addition to any compiler/run-time problems that may be generated by doing this.

#### 5.4.7.2 Replacing port binding

```
void ReplaceBinding(
 const SceMiMessageOutPortBinding* binding,
 SceMiEC* ec=NULL );
```

This method replaces the `SceMiMessageOutPortBinding` object originally furnished to the `SceMi::BindMessageOutPortProxy()` call that created this port proxy object (see 5.4.3.6). This can be useful for replacing contexts or receive callback functions some time after the output message port proxy has been established. Setting the receive callback to a NULL value indicates that any message from the output can be ignored.

The implementation shall copy the contents of the object pointed to by the binding argument to an internal, implementation specific location.

Note: The application is free to deallocate and/or modify the binding object at any time after calling replace port binding. Since the binding object is copied, the binding itself will not change as a result of this.

**binding**

This is new callback and context information associated with this message output port proxy.

#### 5.4.7.3 Accessors

```
const char *
SceMiMessageOutPortProxy::TransactorName() const;
```

This method returns the name of the transactor connected to the port. This is the absolute hierarchical path name to the transactor instance expressed in the netlist's HDL language syntax.

```
const char *
SceMiMessageOutPortProxy::PortName() const;
```

This method returns the port name. This is the path name to the `SceMiMessageOutPort` macro instance relative to the containing transactor expressed in the netlist's HDL language syntax.

```
unsigned
SceMiMessageOutPortProxy::PortWidth() const;
```

This method returns the port width. This is the value of the `PortWidth` parameter that was passed to the associated `SceMiMessageOutPort` instance on the hardware side.

#### 5.4.7.4 Destructor

There is no public destructor for this class. Destruction of all message output ports shall automatically occur when the `SceMi::ShutDown()` function is called.

## 5.5 Macro-Based Software side interface - C API

The SCI-MI software side also provides an ANSI standard C API. All of the following subsections parallel those described in the C++ API. The C API can be implemented as functions that wrap calls to methods described in the C++ API. The prototypes of those functions are shown in this section. For full documentation on a function, see its corresponding subsection in 5.4.

### 5.5.1 Primitive data types

The C API has its own header file with the following minimum content:

```
typedef unsigned SceMiU32;
typedef unsigned long long SceMiU64;
typedef void SceMi;
typedef void SceMiParameters;
typedef void SceMiMessageData;
typedef void SceMiMessageInPortProxy;
typedef void SceMiMessageOutPortProxy;

typedef int (*ServiceLoopHandler)( void *context, int pending );

typedef enum {
 SceMiOK,
 SceMiError,
} SceMiErrorType;
typedef struct {
 const char *Culprit;
 const char *Message;
 SceMiErrorType Type;
 int Id;
} SceMiEC;
typedef void (*SceMiErrorHandler)(void *context, SceMiEC *ec);

typedef enum {
 SceMiInfo,
 SceMiWarning
} SceMiInfoType;
typedef struct {
 const char *Culprit;
 const char *Message;
 SceMiInfoType Type;
 int Id;
} SceMiIC;
typedef void (*SceMiInfoHandler)(void *context, SceMiIC *ic);

typedef struct {
 void *Context;
 void (*IsReady)(void *context);
 void (*Close)(void *context);
} SceMiMessageInPortBinding;
typedef struct {
 void *Context;
 void (*Receive)(
     void *context,
     const SceMiMessageData *data );
 void (*Close)(void *context);
} SceMiMessageOutPortBinding;
```

An application shall include either the C API header or the C++ API header, but not both.

Note: Because ANSI C does not support default argument values, the last SceMiEC *ec argument to each function must be explicitly passed when called, even if only to pass a NULL.

### 5.5.2 Miscellaneous interface support issues

The C miscellaneous functions have semantics like the corresponding C++ methods (shown within 5.4).

```
SceMiEC - error handling
void
SceMiRegisterErrorHandler(
 SceMiErrorHandler errorHandler,
 void *context );
```

#### 5.5.2.1 SceMiIC - informational status and warning handling (info handling)

```
void
SceMiRegisterInfoHandler(
 SceMiInfoHandler infoHandler,
 void *context );
```

### 5.5.3 SceMi - SCE-MI software side interface

See also 5.4.3.

#### 5.5.3.1 Version discovery

```
int
SceMiVersion( const char *versionString );
1.1.4.18     Initialization
SceMi *
SceMiInit(
 int version,
 const SceMiParameters *parameterObjectHandle,
 SceMiEC *ec );
```

#### 5.5.3.2 SceMi Object Pointer Access

```
SceMi *
SceMiPointer(
 SceMiEC *ec );
```

#### 5.5.3.3 Shutdown

```
void
SceMiShutdown(
 SceMi *sceMiHandle,
 SceMiEC *ec );
```

#### 5.5.3.4 Message input port proxy binding

```
SceMiMessageInPortProxy *
SceMiBindMessageInPort(
 SceMi *sceMiHandle,
 const char *transactorName,
 const char *portName,
 const SceMiMessageInPortBinding *binding,
 SceMiEC *ec );
```

#### 5.5.3.5 Message output port proxy binding

```
SceMiMessageOutPortProxy *
SceMiBindMessageOutPort(
 SceMi *sceMiHandle,
 const char *transactorName,
 const char *portName,
 const SceMiMessageOutPortBinding *binding,
 SceMiEC *ec );
```

#### 5.5.3.6 Service loop

```
int
SceMiServiceLoop(
 SceMi *sceMiHandle,
 SceMiServiceLoopHandler g,
 void *context,
 SceMiEC *ec );
```

### 5.5.4 SceMiParameters - parameter access

See also 5.4.4.

#### 5.5.4.1 Constructor

```
SceMiParameters *
SceMiParametersNew(
 const char *paramsFile,
 SceMiEC *ec );
```

This function returns the handle to a parameters object.

#### 5.5.4.2 Destructor

```
void
SceMiParametersDelete(
 SceMiParameters *parametersHandle );
```

#### 5.5.4.3 Accessors

```
unsigned int
SceMiParametersNumberOfObjects(
 const SceMiParameters *parametersHandle,
 const char *objectKind,
 SceMiEC *ec );

int
SceMiParametersAttributeIntegerValue(
 const SceMiParameters *parametersHandle,
 const char *objectKind,
 unsigned int index,
 const char *attributeName,
 SceMiEC *ec );

const char *
SceMiParametersAttributeStringValue(
 const SceMiParameters *parametersHandle,
 const char *objectKind,
 unsigned int index,
 const char *attributeName,
 SceMiEC *ec );

void
SceMiParametersOverrideAttributeIntegerValue(
 SceMiParameters *parametersHandle,
 const char *objectKind,
 unsigned int index,
 const char *attributeName,
 int value,
 SceMiEC *ec );

void
SceMiParametersOverrideAttributeStringValue(
 SceMiParameters *parametersHandle,
 const char *objectKind,
 unsigned int index,
 const char *attributeName,
 const char *value,
 SceMiEC *ec );
```

### 5.5.5 SceMiMessageData - message data object

See also 5.4.5.

#### 5.5.5.1 Constructor

```
SceMiMessageData *
SceMiMessageDataNew(
 const SceMiMessageInPortProxy *messageInPortProxyHandle,
 SceMiEC *ec );
```

This function returns the handle to a message data object suitable for sending messages on the specified input port proxy.

#### 5.5.5.2 Destructor

```
void
SceMiMessageDataDelete(
 SceMiMessageData *messageDataHandle );
```

#### 5.5.5.3 Accessors

```
unsigned int
SceMiMessageDataWidthInBits(
 const SceMiMessageData *messageDataHandle );


unsigned int
SceMiMessageDataWidthInWords(
 const SceMiMessageData *messageDataHandle );

void
SceMiMessageDataSet(
 SceMiMessageData *messageDataHandle,
 unsigned int i,
 SceMiU32 word,
 SceMiEC *ec );

void
SceMiMessageDataSetBit(
 SceMiMessageData *messageDataHandle,
    unsigned int i,
    int bit,
    SceMiEC *ec );

void
SceMiMessageDataSetBitRange(
 SceMiMessageData *messageDataHandle,
    unsigned int i,
    unsigned int range,
    SceMiU32 bits,
    SceMiEC *ec );

SceMiU32
SceMiMessageDataGet(
 const SceMiMessageData *messageDataHandle,
 unsigned int i
 SceMiEC *ec );

int
SceMiMessageDataGetBit(
 const SceMiMessageData *messageDataHandle,
    unsigned int i,
    SceMiEC *ec );
SceMiU32
SceMiMessageDataGetBitRange(
    const SceMiMessageData *messageDataHandle,
    unsigned int i,
    unsigned int range,
    SceMiEC *ec );

SceMiU64
SceMiMessageDataCycleStamp(
 const SceMiMessageData *messageDataHandle );
```

### 5.5.6 SceMiMessageInPortProxy - message input port proxy

See also 5.4.6.

#### 5.5.6.1 Sending input messages

```
void
SceMiMessageInPortProxySend(
 SceMiMessageInPortProxy *messageInPortProxyHandle,
 const SceMiMessageData *messageDataHandle,
 SceMiEC *ec );
```

#### 5.5.6.2 Replacing port binding

```
void SceMiMessageInPortProxyReplaceBinding(
 SceMiMessageInPortProxy *messageInPortProxyHandle,
 const SceMiMessageInPortBinding* binding,
 SceMiEC* ec );
```

#### 5.5.6.3 Accessors

```
const char *
SceMiMessageInPortProxyTransactorName(
 const SceMiMessageInPortProxy *messageInPortProxyHandle );

const char *
SceMiMessageInPortProxyPortName(
 const SceMiMessageInPortProxy *messageInPortProxyHandle );

unsigned
SceMiMessageInPortProxyPortWidth(
 const SceMiMessageInPortProxy *messageInPortProxyHandle );
```

### 5.5.7 SceMiMessageOutPortProxy - message output port proxy

See also 5.4.7.

#### 5.5.7.1 Replacing port binding

```
void SceMiMessageOutPortProxyReplaceBinding(
 SceMiMessageOutPortProxy *messageOutPortProxyHandle,
 const SceMiMessageOutPortBinding* binding,
 SceMiEC* ec );
```

#### 5.5.7.2 Accessors

```
const char *
SceMiMessageOutPortProxyTransactorName(
 const SceMiMessageOutPortProxy *messageOutPortProxyHandle );

const char *
SceMiMessageOutPortProxyPortName(
 const SceMiMessageOutPortProxy *messageOutPortProxyHandle );

unsigned
SceMiMessageOutPortProxyPortWidth(
 const SceMiMessageInPortProxy *messageOutPortProxyHandle );
```

## 5.6 Function-based Interface

### 5.6.1 The DPI C-layer

This section will discuss the C side of DPI.

#### 5.6.1.1 Compliant subset of the SystemVerilog DPI C Layer

The SCE-MI 2 standard defines a subset of a DPI compliant SystemVerilog / C Layer and a subset of DPI data types supported by the SCE-MI 2 standard. That subset conforms to the DPI C Layer as described in Annex F of SystemVerilog LRM IEEE 1800-2005 (see reference [3]).

#### 5.6.1.2 Binding is automatic - based on static names

SCE-MI 2 function-based interface supports binding between where the DPI functions are defined and from where they are called based on static C symbol names. The user needs to define a function on one side and call it from the other side. It will be up to SCE-MI 2 implementation to make sure that wrappers with matching symbol names are provided where appropriate.

All C DPI symbol names conform to ANSI-C naming conventions and linkage. This provides a C symbol linkage mechanism that is adaptable to the HVL environment used on the software side.

##### 5.6.1.2.1 Supported types, static mapping

The SystemVerilog LRM IEEE 1800-2005 section F.6.4 defines the mapping between the basic SystemVerilog data types and the corresponding C types.

*DPI supports a variety of flexible data types ranging from simple scalar types such as integers to bit vectors to complex structures and dynamic arrays. Table F-1— Mapping data types from section F.6.4 in SystemVerilog LRM IEEE 1800-2005 (see reference [3]) defines the mapping between C data types and SystemVerilog DPI types.*

*In addition section 26.4.6 and Annex F sections F.6 and F.) lists the supported data types and their mappings between C data types and SystemVerilog DPI types.*

Table 5.3 lists the subset of those mappings between SystemVerilog and C supported for SCE-MI 2.

| **DPI formal argument types** | **Corresponding types mapped to C** |
|---|---|
| Scalar basic types:<br>byte<br>byte unsigned<br>shortint<br>shortint unsigned<br>int<br>int unsigned<br>longint<br>longint unsigned | Scalar basic types:<br>char<br>unsigned char<br>short int<br>unsigned short int<br>int<br>unsigned int<br>long long<br>unsigned long long |
| scalar values of type bit | unsigned char (with specifically defined values) |
| packed one-dimensional arrays of type bit and logic | canonical arrays of svBitVecVal<br>and svLogicVecVal |

**Table 5.3: Subset of DPI mapping supported in SCE-MI 2 function-based interface**

Note: Integer types, although supported, come with the caveat described above that for C their widths are not cast in stone but for SystemVerilog they are. As a result, the user will have to be aware of this when using these types in terms of knowing when padding is implied and when masking is required. That said, scalar data types that can be passed by value are extremely useful and are supported in SCE-MI 2. It shall of course be assumed that the fixed sizes of these types on the HDL side will be maintained and will always synthesize to the same number of bits.

##### 5.6.1.2.2 4-State logic types

SCE-MI 2 supports conveying both 2 state and 4 state logic types from the HDL side to the C side and vice versa. SCE-MI 2 implementations can handle 4 state logic types as follows:

- No coercion – the HDL side natively supports 4 state types

- No coercion – from HDL to C as the HDL will convey either 2 state types or 4 state types depending on whether the HDL side supports 2 stated or 4 state types.
- Coercion – from C to HDL if the HDL side only supports 2 state types. In this case X will be coerced to 1 and Z will be coerced to 0.

Note: Implementations can provide additional coercion options including warnings when coercion takes place.

Note: The above allows models using 4 state logic types to run on SCE-MI 2 compliant implementation w/o code modification. Support of 4 states types using coercion, while allowing 4 state types to run on 2 state HDL engines (such as 2 sates emulators) does not imply that models using 4 states types will provide results consistent with 4 state HDL engines (such as 4 state simulators) or even correct results. It is up to the modeler/user to decide whether to keep the modes unchanged or remodel the types to 2 state types.

## 5.6.2 The DPI SystemVerilog Layer

This section will discuss the SystemVerilog side of DPI.

### 5.6.2.1 Functions and tasks

The SystemVerilog DPI supports both functions and tasks. An imported or exported DPI function always executes in 0-time. An exported or imported DPI task, by contrast, can execute in 0-time or can consume time.

SCE-MI function-based interface supports exported or imported DPI functions and supports exported and imported DPI tasks. Unless explicitly mentioned in the SCE-MI spec, references to exported and imported DPI functions will also relate to exported and imported DPI tasks. Any subsets and restrictions defined for exported or imported DPI functions also apply to exported or imported DPI tasks.

SCE-MI only supports calling exported tasks from a context DPI imported task call chain. It does not allow calling it from outside a context DPI imported function or task call chain. Any use of imported and exported DPI task which is not allowed by the SystemVerilog LRM or is considered unpredictable or undefined, is not supported by SCE-MI.

Note: Section 4.10 extends the scope of calling DPI exported functions to applications linked with the C side considered by the SystemVerilog LRM “outside a context DPI imported function call chain”. This extension only applies to DPI exported functions and does not apply to DPI exported tasks.

### 5.6.2.2 Support for multiple messages in 0-time

DPI places no restrictions on the number of imported function calls made in the same block of code without intervening time advancement. Implementations must support the ability to transmit multiple messages in 0-time either by calling the same function or by calling multiple functions in the same time step.

### 5.6.2.3 Rules for DPI function call nesting

SCE-MI 2 compliant implementation must support two levels of nesting meaning that the HDL side can call an imported function that can call an exported function. Once the exported function returns, it can yield control back to the imported function. Supporting more than two levels of nesting is allowed by SystemVerilog DPI but considered undefined in SCE-MI 2 meaning it can result in undefined behavior.

Note: SCE-MI does not impose any restrictions on SCE-MI implementations supporting additional levels of nesting. An example for additional levels of nesting is when the exported function (called from an imported function) calls another imported function that calls another exported function establishing a call chain that is 4 levels deep.

### 5.6.2.4 DPI utility functions supported by SCE-MI 2

*DPI defines a small set of functions to help programmers work with DPI context tasks and functions. The term scope is used in the task or function names for consistency with other SystemVerilog terminology. The terms scope and context are equivalent for DPI tasks and functions.*

*There are functions that allow the user to retrieve and manipulate the current operational scope. There are also functions to associate an opaque user data pointer with an HDL scope. This pointer can then later be retrieved when an imported DPI function is called from that scope.*

SCE-MI 2 supports two types of DPI Utility functions, those that involve manipulation of scope (to be called scope-related DPI utility Functions) and additional helper functions that can be used for bit vector manipulation, version query, etc.

The scope-related DPI utility functions are:

```
svScope svGetScope(void)
svScope svSetScope(const svScope scope)
void svPutUserData(const svScope scope, void *userKey, void *userData)
void *svGetUserData(const svScope scope, void *userKey)
const char *svGetNameFromScope(const svScope scope)
svScope svGetScopeFromName(const char *scopeName)
int svGetCallerInfo(char **fileName, int *lineNumber)
```

The helper DPI utility functions are:

```
const char *svDpiVersion(void)
svBit svGetBitselBit(const svBitVecVal *s, int i)
void svPutBitselBit(svBitVecVal *d, int i, svBit s)
void svGetPartselBit(svBitVecVal *d, const svBitVecVal *s, int i, int w)
void svPutPartselBit(svBitVecVal *d, const svBitVecVal s, int i, int w)
```

Note: There is a restriction on when scope related functions can be called. They cannot be called at any time in the simulation prior to completion of design elaboration as it is possible that not all scopes are defined before this point. Helper utility functions can be called at any time.

## 5.7 Time Access

#### 5.7.1.1 Time access from the C side

To access current simulation time on the C side two calls from Verilog standard VPI interface API can be used to get current time and global precision. In any SCE-MI 2 implementation that already supports VPI, no additional work is needed on the part of the implementation to support time access. In SCE-MI 2 function and pipe feature sets, the two calls must be implemented *at least as described below at a minimum*, to provide time access capability.

The `vpi_get_time()` call can be used to obtain current time expressed in simulation units:

```
void vpi_get_time( vpiHandle obj, s_vpi_time *time_p );
```

Specifically for SCE-MI 2 compliance the `vpi_get_time()` call does not need to be implemented in its entirety. The only minimum requirement is that `vpi_get_time()` accepts a `NULL` value for the `obj` argument and a valid pointer to an `s_vpi_time` structure for the `time_p` argument.

The `vpi_get()` call can be used to obtain the global precision units in which current time is expressed:

```
int vpi_get( int prop, vpiHandle obj );
```

Specifically for SCE-MI 2 compliance the `vpi_get()` call does not need to be implemented in its entirety. The only minimum requirement is that `vpi_get()` accepts a value of `vpiTimePrecision` for the `prop` argument and a value of `NULL` for the `obj` argument.

Given the ability to obtain current time in simulation units and the precision of those simulation units, one can easily derive current time expressed in any units desired.

Here is an example of a small “reference code library" that can return current time in NS in any environment that supports the two VPI calls in the manner described above:

```
static uint64_t timescaleFactorForNs;
static bool useMultiplyForNs;

static uint64_t precisionConverterForNs[] = {
    1000000000LL, //  0    1  s
    100000000LL, // -1  100 ms
    10000000LL, // -2   10 ms
    1000000LL, // -3    1 ms
    100000LL, // -4  100 us
    10000LL, // -5   10 us
    1000LL, // -6    1 us
    100LL, // -7  100 ns
    10LL, // -8   10 ns
    1LL, // -9    1 ns
    10LL, // -10  100 ps
    100LL, // -11  10 ps
    1000LL, // -12  1 ps
    10000LL, // -13   100 fs
    100000LL, // -14   10 fs
    1000000LL  // -15   1 fs
};

//-----------------------------------------
// Call this at init time.

void initialize(){
    timescaleFactorForNs =
        precisionConverterForNs[ -vpi_get(vpiTimePrecision,NULL) ];
    useMultiplyForNs = vpi_get(vpiTimePrecision,NULL) >= -9 ? true : false;

}

//-----------------------------------------
// Call this whenever you want time in NS

uint64_t timeInNs() const {
    static s_vpi_time vtime = { vpiSimTime, 0, 0, 0.0 };

    vpi_get_time( NULL, &vtime );
    uint64_t vtime64 = (((uint64_t)vtime.high) << 32) | vtime.low;

    useMultiplyForNs == true ?
        vtime64 * timescaleFactorForNs :
        vtime64 / timescaleFactorForNs ;
}
```

In an emulation environment it will be up to the implementer's infrastructure to keep the C side's internal notion of time properly updated with the emulator's notion.

For streaming threads, the current time access would only be guaranteed at "synchronization points" defined by flushes of DPI pipes.

Note: Support for this is required by both the function call interface and the pipes-based interface.

## 5.8 Pipes-based Interface: Transaction Pipes

### 5.8.1 SCE-MI 2 Pipes Compliance

Implementation providers stating compliance with SCE-MI pipes-based interface must provide at least one implementation of C-side Pipes blocking semantics compliant with "Transaction Pipes API: Blocking, Thread-Aware Interface" specification as defined in section 5.8.4.

SCE-MI 2 C-side Pipes blocking semantics are intended to be implemented using a thread aware application to be determined by the EDA vendor or the end user. This implies that C-side Pipes blocking calls may be implemented by an EDA vendor using their threaded application of choice, or by an end user using their threaded application of choice. SCE-MI 2 defines the interface and the semantics of SCE-MI 2 C-side Transaction Pipes API: blocking, Thread-Aware interface in section 5.8.4.

SCE-MI 2 requires that EDA vendors implementing C-side Pipes blocking interface will allow end users implementing their C-side Pipes blocking interface to use their C-side Pipes blocking implementation together with the EDA vendor C-side Pipes non-blocking interface.

The above specification does not define which threaded applications EDA vendors should use for implementing SCE-MI 2 C-side Pipes blocking interface. This decision is left to the EDA vendor.

The above specification requires the EDA vendor to allow end users to use the EDA vendor provided Pipes C-side non-blocking interface for implementing their thread-aware blocking interface.

The above specification allows using both end-user C-side Pipes blocking interface and EDA vendor C-side Pipes non-blocking interfaces together if end users choose to do so.

The mechanism by which EDA vendor allows end users to choose between their own implementation of Pipes C-side blocking interface and EDA vendor provided C-side Pipes blocking interface is left to the EDA vendor.

### 5.8.2 Transaction Pipes

Transaction pipes are implemented using an API. On the C-side, the transaction pipes API consists of ANSI C functions. On the HDL side the API consists of functions and tasks defined in a SystemVerilog *interface*.

#### 5.8.2.1 C-Side Transaction Pipes API

The C-side transaction pipes API consists entirely of the following set of function declarations:

Configuration, query functions:

```
void *scemi_pipe_c_handle(       // return: pipe handle
    const char *endpoint_path ); // input: path to HDL endpoint instance

svBit scemi_pipe_set_eom_auto_flush(
 void *pipe_handle,  // input: pipe handle
 svBit enabled );    // input: 1=enable autoflush; 0=disable autoflush

typedef void (*scemi_pipe_notify_callback)(
    void *context );       // input: C model context

typedef void *scemi_pipe_notify_callback_handle;
    // Handle type denoting registered notify callback.

scemi_pipe_notify_callback_handle scemi_pipe_set_notify_callback(
    void *pipe_handle,      // input: pipe handle
    scemi_pipe_notify_callback notify_callback,
                            // input: notify callback function
    void *notify_context,   // input: notify context
    int callback_threshold); // input: threshold for notify callback function

void scemi_pipe_clear_notify_callback(
   scemi_pipe_notify_callback_handle notify_callback_handle );
        // input: notify callback handle

void *scemi_pipe_get_notify_context( //return: notify context object pointer
    scemi_pipe_notify_callback_handle notify_callback_handle);  // input:
notify handle

void scemi_pipe_put_user_data(
    void *pipe_handle,      // input: pipe handle
    void *user_key,         // input: user key
    void *user_data);       // input: user data

void *scemi_pipe_get_user_data(
    void *pipe_handle,      // input: pipe handle
    void *user_key);        // input: user key

int scemi_pipe_get_bytes_per_element( // return: bytes per element
    void *pipe_handle );              // input: pipe handle

svBit scemi_pipe_get_direction(//return: 1 for input pipe, 0 for output pipe
    void *pipe_handle );       // input: pipe handle

int scemi_pipe_get_depth( // return: current depth (in elements) of the pipe
    void *pipe_handle );  // input: pipe handle
```

Input pipe interface:

```
void scemi_pipe_c_send(
    void *pipe_handle,       // input: pipe handle
    int num_elements,        // input: #elements to be written
    const svBitVecVal *data, // input: data
    svbit eom );             // input: end-of-message marker flag

void scemi_pipe_c_flush(
    void *pipe_handle );     // input: pipe handle

int scemi_pipe_c_try_send(   // return: #requested elements
                             //         that are actually sent
    void *pipe_handle,       // input: pipe handle
    int byte_offset,         // input: byte offset into data, below
    int num_elements,        // input: #elements to be sent
    const svBitVecVal *data, // input: data
    svBit eom );             // input: end-of-message marker flag

int scemi_pipe_c_try_flush(  // return: indication of flush success
    void *pipe_handle );     // input: pipe handle

int scemi_pipe_c_can_send(   // return: #elements that can be sent
    void *pipe_handle ):     // input: pipe handle
```

Output pipe interface:

```
void scemi_pipe_c_receive(
    void *pipe_handle,       //  input: pipe handle
    int num_elements,        //  input: #elements to be read
    int *num_elements_valid, // output: #elements that are valid
    svBitVecVal *data,       // output: data
    svbit *eom );            // output: end-of-message marker flag

int scemi_pipe_c_try_receive(// return: #requested elements
                             //         that are actually received
    void *pipe_handle,       //  input: pipe handle
    int byte_offset,         //  input: byte offset into data, below
    int num_elements,        //  input: #elements to be read
    svBitVecVal *data,       // output: data
    svBit *eom );            // output: end-of-message marker flag

svBit scemi_pipe_c_in_flush_state( // return: whether pipe is in Flush state
    void *pipe_handle );           // input: pipe handle

int scemi_pipe_c_can_receive( // return: #elements that can be received
    void *pipe_handle );      // input: pipe handle
```

#### 5.8.2.2 HDL-Side API

The HDL-side API is fully defined by the following two SystemVerilog interface declarations.

Input pipe interface:

```
interface scemi_input_pipe();
    parameter BYTES_PER_ELEMENT = 1;
    parameter PAYLOAD_MAX_ELEMENTS = 1;
    parameter BUFFER_MAX_ELEMENTS = <implementation specified>;
    parameter VISIBILITY_MODE = 0;      // must be set to either 1 or 2
                                        // set to 1 for immediate visibility
                                        // set to 2 for deferred visibility
    parameter NOTIFICATION_THRESHOLD = BUFFER_MAX_ELEMENTS;
                                    // Can have a value = 1 or
                                    // BUFFER_MAX_ELEMENTS


    localparam PAYLOAD_MAX_BITS
        = PAYLOAD_MAX_ELEMENTS * BYTES_PER_ELEMENT * 8;

    task receive(
        input  int num_elements,       //  input: #elements to be read
        output int num_elements_valid, // output: #elements that are valid
        output bit [PAYLOAD_MAX_BITS-1:0] data, // output: data
        output bit eom );            // output: end-of-message marker flag
            <implementation goes here>
    endtask

    function int try_receive( // return: #requested elements
                              //         that are actually received
        input  int byte_offset,   //  input: byte_offset into data, below
        input  int num_elements,  //  input: #elements to be read
        output bit [PAYLOAD_MAX_BITS-1:0] data, // output: data
        output bit eom );         // output: end-of-message marker flag
            <implementation goes here>
    endfunction

    function int can_receive();  // return: #elements that can be received
            <implementation goes here>
    endfunction

    modport receive_if( import receive, try_receive, can_receive );
endinterface
```

Output pipe interface:

```
interface scemi_output_pipe();
   parameter BYTES_PER_ELEMENT = 1;
   parameter PAYLOAD_MAX_ELEMENTS = 1;
   parameter BUFFER_MAX_ELEMENTS = <implementation specified>;
   parameter VISIBILITY_MODE = 0;// must be set to either 1 or 2
                                 // set to 1 for immediate visibility
                                 // set to 2 for deferred visibility
   parameter NOTIFICATION_THRESHOLD = BUFFER_MAX_ELEMENTS;
                                 // Can have a value = 1 or
                                 // BUFFER_MAX_ELEMENTS


   localparam PAYLOAD_MAX_BITS = PAYLOAD_MAX_ELEMENTS * BYTES_PER_ELEMENT * 8;

   task send(
        input int num_elements,       // input: #elements to be written
        input bit [PAYLOAD_MAX_BITS-1:0] data, // input: data
        input bit eom );              // input: end-of-message marker flag
            <implementation goes here>
   endtask

   task flush;
            <implementation goes here>
   endtask

   function int try_send( // return: #requested elements
                          //         that are actually sent
      input int byte_offset,      // input: byte_offset into data, below
      input int num_elements,     // input: #elements to be sent
      input bit [PAYLOAD_MAX_BITS-1:0] data, // input: data
      input bit eom );            // input: end-of-message marker flag
            <implementation goes here>
   endfunction

   function int try_flush(); // return: 1 if pipe is successfully flushed
                             //                  i.e. an empty pipe
            <implementation goes here>
   endfunction

   function int can_send();   // return: #elements that can be sent
            <implementation goes here>
   endfunction

   modport send_if( import send, flush, try_send, can_send );
endinterface
```

### 5.8.3 Pipe handles

On the HDL side, a pipe interface endpoint is defined using the SystemVerilog *interface* construct.

Once the HDL side has instantiated a pipe interface, all pipe operations in the HDL code are done by calling functions and tasks defined within that interface.

The path to this endpoint interface instance uniquely identifies a specific pipe endpoint in an HDL hierarchy to which the C-side can bind. Using this path, the C application can derive a handle that is used in all operations involving the C-side endpoint of the pipe by calling the following function:

```
void *scemi_pipe_c_handle(       // return: pipe handle
    const char *endpoint_path ); // input: path to HDL endpoint instance
```

Note: The pipe handle can be derived once at initialization time and reused many times without having to set scope each time and requiring the internal implementation to do a lookup based on the scope and the pipe ID to retrieve the internal data structure associated with a pipe on each and every pipe operation.

Once a pipe handle is derived, it can be used as the handle argument for all the function calls described in the following sections to perform operations to the C-side endpoint of the designated pipe.

The arguments consist of:

- `endpoint_path` – the hierarchical path to the interface instance representing the opposite HDL endpoint of the pipe

## 5.8.4 Transaction Pipes API: Blocking, Thread-Aware Interface

### 5.8.4.1 Transaction input pipes – blocking operations

#### 5.8.4.1.1 Blocking input pipe access functions

The bold text in the *input pipe* SystemVerilog interface declaration below shows the *blocking receive* function:

```
interface scemi_input_pipe();
    ...

    task receive(
        input  int num_elements,       //  input: #elements to be read
        output int num_elements_valid, // output: #elements that are valid
        output bit [PAYLOAD_MAX_BITS-1:0] data, // output: data
        output bit eom );            // output: end-of-message marker flag
            <implementer supplied implementation goes here>
    endtask

    ...

endinterface
```

The infrastructure will supply the implementation of this task - essentially it is a *built-in* function and its declaration can be placed in an implementation provided file that defines the interface which can be compiled as a separate unit along with all of the user's other modules, packages and interfaces.

The arguments consist of:

- `num_elements` - number of elements to be read on this receive operation - can vary from call to call which again, facilitates data shaping
- `num_elements_valid` - number of read elements that are valid - in the case of data shaping or flushing this can be less than the requested number of bytes read if the `eom` and/or flush comes at some residual number of elements that does not fill out an entire request (see section 5.8.4.3.4).
- `data` - a user supplied target bit vector to which the requested `num_elements` will be deposited
- `eom` - a flag that can serve as an end-of-message marker on a variably sized message transmitted as a sequence of transactions
- The `data` and `eom` arguments always have an output direction when receiving from a pipe.

On the C side endpoint of an *input pipe*, the *blocking send* function provided by the infrastructure is declared as follows:

```
void scemi_pipe_c_send(
    void *pipe_handle,       // input: pipe handle
    int num_elements,        // input: #elements to be written
    const svBitVecVal *data, // input: data
    svbit eom );             // input: end-of-message marker flag
```

Note the following properties:

- `pipe_handle` - the handle identifying the specific pipe as derived from the unique path to the HDL endpoint of the pipe (see section 5.8.3).
- `num_elements` - number of elements to be sent on this send operation - can vary from call to call which again, facilitates data shaping

- `data` - a user supplied bit vector from which the requested `num_elements` will be obtained and sent to the pipe
- `eom` - a flag that can serve as an end-of-message marker on a variably sized message transmitted as a sequence of transactions
- The `data` and `eom` arguments always have an input direction when sending to a pipe.
- The `eom` flag is a user defined flag. Whatever value is passed to the send endpoint of the pipe will be received at the receive endpoint. This is useful for creating end-of-message markers in variable length messages or indicating flush points to the other end. In certain cases it can also be used to force flushes on a pipe (see description of autoflush in section 5.8.4.3.3).

On the C-side endpoint of an *input pipe*, the *flush* function provided by the infrastructure is declared as follows:

```
void scemi_pipe_c_flush(
    void *pipe_handle )  // input: pipe handle
```

Note the following properties:

- pipe_`handle` - the handle identifying the specific pipe as derived from the unique path to the HDL endpoint of the pipe (see section 5.8.3).

#### 5.8.4.2 Transaction output pipes – blocking operations

##### 5.8.4.2.1 Blocking output pipe access functions

The bold text in the *output pipe* SystemVerilog interface declaration below shows the *send and flush* functions which make up the API for the blocking operations of the HDL endpoint of an output pipe:

```
interface scemi_output_pipe();
    ...

    task send(
        input int num_elements,     // input: #elements to be written
        input bit [PAYLOAD_MAX_BITS-1:0] data, // input: data
        input bit eom );            // input: end-of-message marker flag
            < implementer supplied implementation goes here>
    endtask

    task flush;
            < implementer supplied implementation goes here>
    endtask

    ...
endinterface
```

The infrastructure will supply the implementation of these tasks - essentially they are *built-in* functions and their declarations can be placed in an implementation provided file that defines the interface which can be compiled as a separate unit along with all of the user's other modules, packages and interfaces.

The arguments for the `send()` task consists of:

- `num_elements` - number of elements to be sent on this send operation - can vary from call to call which again, facilitates data shaping
- `data` - a user supplied bit vector from which the requested `num_elements` will be obtained and sent to the pipe
- `eom` - a flag that can serve as an end-of-message marker on a variably sized message transmitted as a sequence of transactions
- The `data` and `eom` arguments always have an input direction when sending to a pipe.

On the C side endpoint of an *output pipe*, the *blocking receive* function provided by the infrastructure is declared as follows:

```
void scemi_pipe_c_receive(
    void *pipe_handle,       //  input: pipe handle
    int num_elements,        //  input: #elements to be read
    int *num_elements_valid, // output: #elements that are valid
    svBitVecVal *data,       // output: data
    svbit *eom );
```

Note the following properties:

- `pipe_handle` - the handle identifying the specific pipe as derived from the unique path to the HDL endpoint of the pipe (see section 5.8.3).
- `num_elements` - number of elements to be read on this receive operation - can vary from call to call which again, facilitates data shaping
- `num_elements_valid` - number of read elements that are valid - in the case of data shaping this can be less than the requested number of bytes read if the `eom` and/or flush comes at some residual number of elements that does not fill out an entire request (see section 5.8.4.3.4).
- `data` - a user supplied target bit vector to which the requested `num_elements` will be deposited
- `eom` - a flag that can serve as an end-of-message marker on a variably sized message transmitted as a sequence of transactions
- The `num_elements_valid`, `data` and `eom` arguments always have an output direction when receiving from a pipe.

#### 5.8.4.3 Flush Semantics

The SCE-MI transaction pipes API supports two types of flushing operations for pipes:

- explicit flushing
- implicit flushing

##### 5.8.4.3.1 Explicit flushing of pipes

When an *explicit flush* occurs on a pipe, it allows the producer of transactions previously sent on that pipe to suspend execution until all in-transit messages have been consumed by the consumer. The `scemi_pipe_c_flush()` call (see section 5.8.5.5.2) is used to flush transaction input pipes. The `flush()` task (see section 5.8.4.3.1) is used to flush transaction output pipes.

##### 5.8.4.3.2 Implicit flushing of pipes

Transaction pipes also support implicit flushing. If a pipe is enabled for implicit flushing, flushes will automatically occur on *end-of-message* (`eom`). This mode is called *autoflush*.

If a pipe has *autoflush* mode enabled, when a blocking send is performed on that pipe with the *end-of-message* (`eom`) flag set, the effect is as if an explicit blocking flush was combined with that send. The blocking send will not return until the consumer has fully received all messages in the pipe up to and including the `eom` tagged message being passed to it.

Producers of transactions to pipes can still call the explicit pipe flush functions at any time even on pipes that have autoflush mode enabled.

##### 5.8.4.3.3 Enabling autoflush

The following call lets an application indicate that *autoflush* mode is enabled or disabled for a pipe designated by a given handle. This configuration call is always initiated only from the C side for both input and output pipes.

For any given pipe on which this mode is enabled, a `scemi_pipe_c/hdl_send()` call with an `eom` value of 1 will have the same effect as if a `scemi_pipe_c/hdl_flush()` call was made following that data send call.

```
svBit scemi_pipe_set_eom_auto_flush(
 void *pipe_handle,  // input: pipe handle
 svBit enabled );    // input: enable/disable
```

Note the following properties:

- `pipe_handle` - the handle identifying the specific pipe as derived from the unique path to the HDL endpoint of the pipe (see section 5.8.3).
- `enabled` - flag that indicates enable (1) or disable (0) this mode.
- The call returns the previous mode setting
- The pipe will remain in its current mode until any subsequent call to this function that changes the mode.
- By default, pipes are not in autoflush mode.
- If a call is made to `scemi_pipe_set_eom_auto_flush()` to enable autoflush mode when the pipe is not empty, the new setting will go into effect only on the next pipe operation and will not cause a flush of any existing elements in transit through the pipe. For example, if a pipe with a capacity of 10 contains 5 elements and a `scemi_pipe_set eom_auto_flush()` call then enables autoflush mode, no flush is performed even if 1 or more of those 5 elements has `eom` set. However, if a call is subsequently made to send 1 more element with the `eom` bit set, all 6 elements are then flushed.
- The return value is the previous setting of eom autoflush mode (1 if enabled, 0 if not).

If autoflush mode is enabled and a call is made to one of the non-blocking pipe send functions (C-side `scemi_pipe_c_try_send()` or HDL-side `try_send()`) with the `eom` bit enabled, it will have the same effect on the internal state of the pipe as if the corresponding non-blocking flush function (C-side `scemi_pipe_c_try_flush()` or HDL-side `try_flush()` respectively), was called immediately after the send call.

#### 5.8.4.3.4 Using flushing with data shaping

When a flush (either implicit or explicit) occurs on a pipe used for data shaping, special considerations must be made if a producer endpoint of a pipe does data send operation with a smaller `num_elements` than that requested by the subsequent data receive operation at the consumer endpoint of that pipe. If the pipe is flushed on that send operation, in order to satisfy the flush the consumer will see a return of `num_elements_valid` that is smaller than its requested `num_elements`. This is because, in order to satisfy the producer's flush condition, the consumer's blocking receive call must have satisfactorily returned from its read operation even if that read operation was asking for a larger number of elements than had been sent as of the time of the flush.

A similar issue applies when specifying a pre-mature `eom` as explained in section 4.8.8.1. Using the *nozzle* example from that section, if a consumer requests 100 elements, but the producer only sends 75 elements before flushing (either implicitly using `eom` or explicitly), the request to read 100 elements will return with a `num_elements_valid` of only 75 thus leaving the pipe empty as required by the flush and/or `eom`.

### 5.8.4.4 Blocking pipes co-existance model

SCE-MI requires that implementer provided C-side pipes blocking interface shall not prevent user provided C-side pipes blocking interface from co-existing with the implementer C-side pipes non-blocking interface.

Note: SCE-MI C-side Pipes blocking semantics are intended to be implemented using a thread aware application to be determined by either the implementer or the end user. This implies that C-side Pipes blocking calls may be implemented in a tool using their threaded application of choice, and by an end user using their threaded application of choice. SCE-MI defines the interface and the semantics of SCE-MI C-side Transaction Pipes API: blocking, Thread-Aware interface in section 5.8.4.

SCE-MI requires that a tool implementing C-side Pipes blocking interface will allow end users implementing their C-side Pipes blocking interface to use their C-side Pipes blocking implementation together with the tool implementer C-side Pipes non-blocking interface.

Note: The above specification does not define which threaded applications implementers should use for implementing SCE-MI C-side Pipes blocking interface. This decision is left to the implementer. The above specification requires the implementer to allow end users to use the implementer provided Pipes C-side non-blocking interface for implementing their thread-aware blocking interface. The above specification also allows using both end-user SCE-MI C-side Pipes blocking interface and implementer C-side Pipes non-blocking interfaces together if end users choose to do so.

The mechanism by which implementers allows end users to choose between their own implementation of Pipes C-side blocking interface and implementer provided C-side Pipes blocking interface is left to the implementer.

### 5.8.5 Basic Transaction Pipes API: Non-Blocking, Thread-Neutral Interface

Everything described so far has pertained to the blocking pipes. Additionally it is desirable to support a non-blocking pipes interface that can be used in *thread-neutral* implementations.

The non-blocking pipe interface calls have the following semantics.

- Thread-neutral - no thread-awareness required in the implementation
- Is sufficiently compatible with the OSCI-TLM interface model that it can be directly used to implement OSCI- TLM compliant interfaces
- Support user configuration and query of buffer depth
- Provide primitive non-blocking operations which can be used to build higher level interfaces that have blocking operations implemented in selected threading systems

#### 5.8.5.1 Pipe Semantics

##### 5.8.5.1.1 Visibility Modes

Transaction pipes support two modes of operations:

- deferred visibility
- immediate visibility

In ***deferred visibility*** mode, there is a defined lag between when elements are written to a pipe by the producer and when they are actually visible and available for consumption by the consumer. In this mode, while the producer has execution control, it may place one or more elements in the pipe via send operations, but the consumer cannot see any of the elements until it has been notified that the pipe is either filled or flushed. Conversely, once a pipe is filled or flushed, the producer cannot place any new elements in the pipe until it has been notified that all the previously added elements were consumed by the consumer via receive operations.

In ***immediate visibility*** mode, any elements written by the producer are immediately visible and are available for consumption whenever the consumer gains execution control even if notification has not occurred. In this mode, while the producer has execution control, it may place one or more elements in the pipe via send operations. Even without the producer filling or flushing the pipe, the consumer can consume any of the elements in the pipe when it has execution control. Conversely, when execution control is switched from the consumer back to the producer, the producer may add new elements to the pipe even when previously added elements in the pipe have not been completely consumed by the consumer via receive operations.

To place a pipe in immediate visibility mode, the fourth `VISIBILITY_MODE` parameter of the pipe interface must be set to the value 1. To place a pipe in deferred visibility mode, the `VISIBILITY_MODE` parameter of the pipe interface must be set to the value 2. By default, this parameter is set to 0, denoting an illegal pipe interface with unspecified mode of operation.

##### 5.8.5.1.2 Notifications

As described in Section 5.8.2.1, the `scemi_pipe_set_notify_callback()` function is used to register user-defined notify callback functions for pipes. There are two types of programmable callback functions, a user defined "notify ok to send" function for an input pipe, and a user defined "notify ok to receive" function for an output pipe. The "notify operations" shown above correspond to calling these programmable functions from within the infrastructure to notify the application C-side that it has become *serviceable*. A producer becomes serviceable when it can place elements in the pipe via send operations; while a consumer becomes serviceable when it can consume elements from the pipe via receive operations.

For the deferred visibility mode, even if a consumer gains execution control, it will not have access to any elements in the pipe until it has been notified that the producer has filled or flushed the pipe. Similarly, even if a producer gains execution control, it will not have access to any empty space in the pipe until it has been notified that the consumer has emptied the pipe. Notification callback conditions must be met before exclusive access to pipe elements (or empty space) is allowed to switch between producer and consumer. For the immediate visibility mode, whenever a consumer gains execution control, it is always free to access any elements in the pipe and is not required to wait for notification. Similarly, whenever a producer gains execution control, it is always free to access any empty space in the pipe and is not required to wait for notification.

If any of the notify operations occur on the C-side of a pipe (i.e. notify of an input pipe's producer side or notify of an output pipe's consumer side), the registered notify callback functions of that pipe (if any) would be called immediately. In this case "immediately" means that the semantics of the notify callback has identical semantics to an HDL process calling an imported pure DPI function.

#### 5.8.5.1.3 General Concepts and Semantic Definitions for Pipes

- **Buffer Capacity (BUFFER_MAX_ELEMENTS):** Buffer capacity is defined as the maximum number of elements that can exist in the buffer before a blocking send will block or a non-blocking send will fail. Buffer capacity is the same as the static `BUFFER_MAX_ELEMENTS` parameter defined in the SystemVerilog pipe interface definition. The default value of `BUFFER_MAX_ELEMENTS` is implementation defined but can be overridden by the user. If specified it must be greater than PAYLOAD_MAX_ELEMENTS or an error will occur.
- **Notification Threshold:** The notification threshold defines the minimum number of elements that must be added to an empty pipe before a consumer is notified or removed from a full pipe before a producer is notified. The notification threshold can be specified via the static parameter, `NOTIFICATION_THRESHOLD`. It can be set to either 1 or `BUFFER_MAX_ELEMENTS`. All other values will result in an error. The default notification threshold for a pipe is `BUFFER_MAX_ELEMENTS`. When the notification threshold is set to `BUFFER_MAX_ELEMENTS`, notifications due to send and receive operations that occur when the pipe becomes full or empty, i.e. after adding `BUFFER_MAX_ELEMENTS` elements to an empty pipe or removing `BUFFER_MAX_ELEMENTS` elements from a full pipe. When the notification threshold is set to 1, notifications due to send and receive operations occur immediately, i.e. after adding at least one element to an empty pipe or removing at least one element from a full pipe. Notifications due to flush operations are not affected by the notification threshold setting. The state diagram in the next section will describe the precise semantics of when notifications occur with respect to threshold settings.
- **Producer, consumer:** The state diagram shown in the next section is generalized to refer to the *producer side* and the *consumer side* without referring specifically to input pipes or output pipes. In this way, the same diagram can generically and symmetrically describe either pipe direction. This implies the following:
  - For an input pipe, the producer side is the C-side and the consumer side is the HDL-side
  - For an output pipe, the producer side is the HDL-side and the consumer side is the C-side
- **Serviceable:** A pipe endpoint becomes serviceable if the pipe enters a state where that endpoint can successfully move elements to or from the pipe. Of course, execution control must first be acquired before elements can be added by a producer or removed by a consumer from the pipe.

  For a pipe in deferred visibility mode:
  - The producer side is serviceable only for the states within the left bounded box in the state diagram:
    *Empty/Buffering*, *Empty/Pending Receive*
  - The consumer side is serviceable only for the states within the right bounded box in the state diagram:
    *Full/Buffering*, *Full/Pending Send*, *Flush*

  For a pipe in immediate visibility mode:
  - The producer side is serviceable whenever there is room to add at least one more element to the pipe, except when the pipe is in the *Flush* state.
  - The consumer side is serviceable whenever there is at least one element in the pipe. The pipe can be in any state in the state diagram.

- **Input actions / output consequences:** Each pipe transition is caused by an input action and may involve an output consequence.

  Here are the possible input actions:

  - `prod.try_send()[adds]` – producer adds one or more elements to the pipe but does not fill it.
  - `prod.try_send()[fills]` – producer adds elements and fills the pipe but does not fail.
  - `prod.try_send()[fails]` – producer attempts to add more elements than can be placed in the pipe (i.e. exceeds buffer capacity) and thus fills the pipe before the request can be satisfied.
  - `prod.try_flush()` – producer attempts to flush the pipe.
  - `cons.try_receive()[removes]` – consumer removes one or more elements from the pipe but does not empty it.
  - `cons.try_receive()[empties]` – consumer removes elements and empties the pipe but does not fail.
  - `cons.try_receive()[fails]` – consumer attempts to remove more elements than can be provided by the pipe and thus empties the pipe before the request can be satisfied.

  The only possible output consequences of a pipe state transition are notify operations which can only occur in the specific pipe transitions that go from one side being notified to the other side being notified (i.e. transitions crossing the bounded box boundaries in the state diagrams):

  - `/notify prod` – the consumer side notifies the producer side.
  - `/notify cons` – the producer side notifies the consumer side.

  In the state diagram, an additional modifier indicates how an input action's behavior depends on the notification threshold. If a send operation is performed, the number of elements that *will be* in the pipe *after* the send operation is compared to the threshold. If a receive operation is performed, the number of empty element space that *will be* in the pipe *after* the receive operation is compared to the threshold.

  A `'>='` modifier denotes the state transition that will occur if the indicated action occurs when the number of elements or empty space in the pipe after a send or receive operation is greater than or equal to the notification threshold.

  A `'<'` modifier denotes the state transition that will occur if the indicated action occurs when the number of elements or empty space in the pipe after a send or receive operation is less than the notification threshold.

  For the `prod.try_send()[adds]` action described above, an annotation with a threshold modifier has the following modified meanings:

  - `prod.try_send()[adds>=]` – producer adds one or more elements to the pipe but does not fill it, *and the number of elements that will be in the pipe after the send operation is greater than or equal to the threshold*.
  - `prod.try_send()[adds<]` – producer adds one or more elements to the pipe but does not fill it, *and the number of elements that will be in the pipe after the send operation is less than the threshold*.

  For the `cons.try_receive()[remove]` action described above, an annotation with a threshold modifier has the following modified meanings:

- `cons.try_receive()[removes>=]` – consumer removes one or more elements from the pipe but does not empty it, *and the number of empty space that will be in the pipe after the receive operation is greater than or equal to the threshold*.
- `cons.try_receive()[removes<]` – consumer removes one or more elements from the pipe but does not empty it, *and the number of empty space that will be in the pipe after the receive operation is less than the threshold*.

If a pipe transition is annotated with a mode of operation, such as immediate visibility mode or deferred visibility mode, that transition applies only for that specified mode. If an action is shown on a transition with no threshold modifier, that transition applies regardless of the notification threshold setting.

Note: Notifications to the C-side will result in callback functions being called only when a non-NULL callback function pointer is currently registered.

- **Blocking vs. non-blocking operations:** All pipe operations in the state diagram are shown in terms of non-blocking operations. However, semantics of blocking operations can be inferred from this since it is possible to describe blocking operations in terms of non-blocking operations For example, the same condition that causes a non-blocking operation to fail will cause a blocking operation to block. A blocking operation can be thought of as a loop of a non-blocking operation and a wait on an implied event until the condition for unblocking is satisfied. Update of the event is implied to occur on the notify operation. Examples of C-side reference implementations of blocking operations built from non-blocking API calls are shown in section 5.8.5.3.
- **Push/pull operation:** This is used to describe behavior of a pipe when a producer is trying to *push* data through a pipe or a consumer is trying to *pull* data through a pipe. Push/pull operation has slightly different behaviors depending on the current number of elements in a pipe compared to its notification threshold setting.

  If a producer send operation is successful and it fills the pipe (i.e. the action of `prod.try_send()[fills]`), this will not trigger a notify to the consumer side unless there is a pending receive in effect (i.e. the pipe is in the *empty/pending receive* state). In this case the pending receive is attempting to "pull" data from the pipe and therefore wants to be notified as soon as the pipe becomes full.

  Similarly, if a consumer receive operation is successful and it empties the pipe (i.e. the action of `cons.try_receive()[empties]`), this will not trigger a notify to the producer side unless there is a pending send in effect (i.e. the pipe is in the *full/pending send* state). In this case the pending send is attempting to "push" data to the pipe and therefore wants to be notified as soon as the pipe becomes empty.

  Note the following additional property of push/pull operation:

  For deferred mode only, if a pipe is in the *empty/buffering* state and the consumer requests data, this will cause a transition to the *empty/pending receive* state since the request fails and there is now a pending receive. This is true regardless of the number of elements that may have been added to the pipe up to this point while it was in the *empty/buffering* state.

  However, there is a slight difference in operation between the following two scenarios while the pipe is in the *empty/buffering* state just prior to the point at which the consumer receive operation occurs:

  1. Pipe is in the *empty/buffering* state and the pipe is not full
  2. Pipe is in the *empty/buffering* state and the pipe is full

  In scenario #1, the transition to the *empty/pending receive* state still occurs and, as shown in the diagram, no notification occurs. From this point if the producer then precisely fills the pipe (i.e. does not fail) a notification will occur to the consumer and the pipe will transition to the *full/buffering* state.

In scenario #2, the transition to the *empty/pending receive* state still occurs and, as shown in the diagram, no notification occurs. However in this case the notification will occur only when the producer does yet another send, causing a fail. In this case the pipe would transition to the *full/pending* send state.

So, as can be seen from above description, slightly different notification behavior will occur depending on which of the above two scenarios is in effect at the time the consumer makes its receive request.

Coherency of pipe states is expected to be maintained by the implementation identically on both sides of the pipe. That is, the producer side's view of the pipe state must be identical to the consumer side's view at all times. Notify operations only occur on transitions to different states and never to the same state. Therefore maintaining this coherency across the two sides is practical and efficient. It also guarantees minimal notifies across the C-side/HDL-side boundary and prevents an "oscillation" of repeated "try" operations such as a polling operation, causing excessive notifies across the link in cases where the pipe remains in the same state.

#### 5.8.5.1.4 Pipe States

Figure 5.11 below specifies the precise operation of a transaction pipe. The formal definitions of the states and the actions and consequences associated with the state transactions are described in the text that follows.

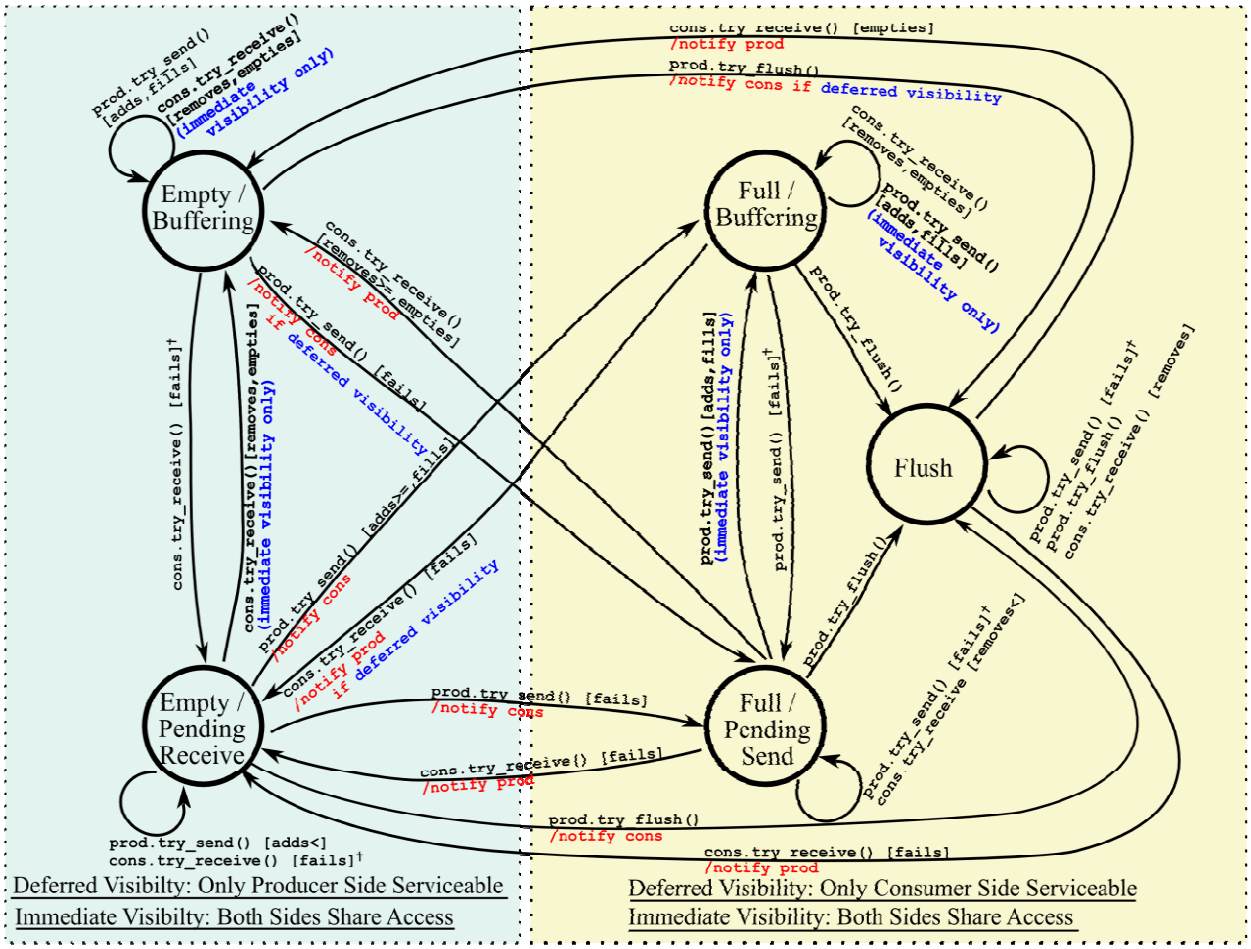



Figure 5.11 Pipe State Diagram

† Note: In this diagram, states from which input actions occur with `[fails]†` notations indicate that special semantics need to be considered depending on whether the pipe is in deferred visibility mode or immediate visibility mode.

- For deferred visibility mode, from these states, only a [fails] is possible for the indicated action type. If `prod.try_send()` operation is attempted from the flush state, full/buffering state or

full/pending send state, [adds] or [fills] actions are not applicable (they will always fail). Similarly, if `cons.try_receive()` operation is attempted from the empty/pending and empty/pending receive state, [removes] or [empties] actions are not applicable.

- For immediate visibility mode, if `prod.try_send()` operation is attempted from the flush state, [adds] or [fills] actions are not applicable (they will always fail). However, from the empty/buffering state and empty/pending receive state, [removes] and [empties] actions are possible if the pipe is not empty. Similarly from the full/buffering state and full/pending send state, [adds] and [fills] actions are possible if the pipe is not full.

The following are descriptions of the pipe states:

- **Empty/Buffering State**:

  A pipe in deferred visibility mode transitions to the *empty/buffering state* from the *full/pending send state* or the *flush state* when a consumer receive operation empties the pipe. The pipe remains in this state as long as the producer is able to successfully send its requested number of elements. Even when BUFFER_MAX_ELEMENTS elements are added, the elements are still not visible to the consumer, thus the pipe remains "empty" from the consumer's point of view. Only when a failed operation occurs will there be a change of pipe state. A failed receive operation causes the pipe to transition to the *empty/pending receive state*. A failed send operation causes the pipe to transition to the *full/pending send state* where only the consumer is serviceable.

  A pipe in immediate visibility mode can transition to the *empty/buffering state* from three different pipe states. From the *flush state*, the transition occurs when a consumer receive operation empties the pipe. From the *full/pending send state*, the transition occurs when the consumer empties the pipe or removes enough elements to meet the notification threshold. From the *empty/pending receive state*, the transition occurs when there is no longer a pending receive, that is, when a consumer receive operation is successful in removing its requested number of elements from the pipe. Identical to deferred visibility mode, the pipe remains in this state as long as the producer is able to successfully send its requested number of elements. Once elements are added to the pipe by the producer, the consumer also becomes serviceable in this "buffering" state and can remove the added elements at any time.

- **Full/Buffering State**:

  A pipe in deferred visibility mode transitions to the *full/buffering state* from the *empty/pending receive state* when a producer send operation fills the pipe. The pipe remains in this state as long as the consumer is able to successfully receive its requested number of elements. Even when all the elements are removed, the empty pipe is still not visible to the producer, thus the pipe remains "full" from the producer's point of view. Only when a failed operation occurs will there be a change of pipe state. A failed send operation causes the pipe to transition to the *full/pending send state*. A failed receive operation causes the pipe to transition to the *empty/pending receive state* where only the producer is serviceable.

  A pipe in immediate visibility mode transitions to the *full/buffering state* from the *empty/pending receive state* when the producer fills the pipe or adds enough elements to meet the notification threshold. In addition, it also transitions to the *full/buffering state* from the *full/pending send state* when there is no longer a pending send. That is, when a producer send operation is successful in adding its requested number of elements to the pipe. Identical to deferred visibility mode, the pipe remains in this state as long as the consumer is able to successfully receive its requested number of elements. Once elements are removed from the pipe by the consumer, the producer also becomes serviceable in this "buffering" state and can add new elements to the empty space of the pipe at any time.

- **Empty/pending receive state:**

  A pipe transitions to the *empty/pending receive state* when a consumer receive operation fails. This occurs when the consumer attempts to receive more elements than what is available in the pipe, thus causing the pipe to become empty before satisfying the receive request. The pipe remains in this state as long as the producer is able to successfully send its requested number of

elements without filling the pipe or meeting the notification threshold. Unlike the *empty/buffering state*, in this “pending” state, it does not require a failed operation for a change of pipe state. Just filling the pipe or meeting the notification threshold is enough. This constitutes a *pull operation* where the consumer tries to "pull" data from the pipe and wants to be notified when the pipe becomes full or meets the notification threshold (see push/pull description above).

- **Full/pending send state**:

  A pipe transitions to the *full/pending send state* when a producer send operation fails, unless the pipe is in the *flush state*. This occurs when the producer attempts to send more elements than the available empty space in the pipe, thus causing the pipe to become full before satisfying the send request. The pipe remains in this state as long as the consumer is able to successfully receive its requested number of elements without emptying the pipe or meeting the notification threshold. Unlike the *full/buffering state*, in this “pending” state, it does not require a failed operation for a change of pipe state. Just emptying the pipe or meeting the notification threshold is enough. This constitutes a *push operation* where the producer tries to "push" data through the pipe and wants to be notified when the pipe becomes empty or meets the notification threshold (see push/pull description above).

- **Flush State**:

  A pipe transitions to the *flush state* when a producer flush operation is called on a non-empty pipe. The pipe remains in this state until all elements are removed by the consumer.

#### 5.8.5.2 General Application Use Models for Pipes

Threshold and visibility settings can be combined to configure 3 useful transaction pipe models:

- NOTIFICATION_THRESHOLD=BUFFER_MAX_ELEMENTS, VISIBILITY_MODE=2 (deferred visibility): This is a pipe configuration where notify callbacks occur only when a pipe is empty, full or flushed, and producer and consumer would alternately granted exclusive access to the pipe via notification. This is useful for implementing pipelined or streaming semantics similar to how Unix named pipes or file streams work. Data written by the producer is not visible to the consumer until it is pushed through by becoming full or flushed. Transaction pipes that are configured with this setting shall be referred to as deferred pipes.
- NOTIFICATION_THRESHOLD=BUFFER_MAX_ELEMENTS, VISIBILITY_MODE=1 (immediate visibility): This is a pipe configuration where notify callbacks occur only when a pipe is empty, full or flushed while allowing shared access to the pipe for the consumer and producer. This provides applications the flexibility to overlap send and receive operations while limiting notification frequency for performance. Transaction pipes that are configured with this setting shall be referred to as immediate pipes.
- NOTIFICATION_THRESHOLD=1, VISIBILITY_MODE=1 (immediate visibility): This is a pipe configuration where notify callbacks occur when a pipe is empty, full or flushed as well as when elements are added to an empty pipe or removed from a full pipe. This is useful for implementing fifo semantics similar to how tlm_fifo's work in the OSCI TLM standard. Data written by the producer is immediately visible to the consumer and any consumer doing a blocking get will be immediately notified. Transaction pipes that are configured with this setting shall be referred to as fifos.

No other combinations of `NOTIFICATION_THRESHOLD` and `VISIBILITY_MODE` are defined and will result in an error.

##### 5.8.5.2.1 Deferred Pipe Semantics

The following paragraph explains how the pipe can be configured for use model *#1 ab*ove (deferred pipe semantics). These semantics are in effect when a pipe has been configured for deferred visibility (`VISIBILITY_MODE = 2`). In this visibility mode, the threshold setting is automatically set to `BUFFER_MAX_ELEMENTS` regardless of the user-specified parameter value.

For deferred pipe semantics only one side of the pipe can be serviceable at one time. The goal of deferred pipe semantics is to defer notification and visibility (or data transfer) as long as possible to promote buffered data aggregation and streaming.

Initially the pipe is in an *Empty/Buffering* state. This means that from the point of view of the consumer, the pipe is empty. From the point of view of the producer, the pipe is also empty after initialization but it is serviceable in the sense that at any time it is free to start at adding elements.

Consider two possible scenarios starting from the initial *Empty/Buffering* state:

1. The producer takes the first action of adding elements to the pipe.
2. The consumer takes the first action of attempting to remove elements from the pipe.

In scenario #1, as long as the producer continues to add elements up to and including the point of filling the pipe, the pipe remains in the *Empty/Buffering* state. This is because from the consumer's point of view the pipe is still empty since it has not been notified otherwise – that is, its notification is *deferred*. Once the pipe is filled, if the producer attempts to send another element to the pipe, this attempt will fail since the pipe is full.

At this point a notification is finally sent to the consumer and the pipe enters the *Full/Pending* send state since it assumes that the producer now has a pending send that it wishes to make. At this point, the pipe is only serviceable by the consumer and not the producer.

As long as consumer continues to successfully remove elements without emptying the pipe, the pipe remains in the *Full/Pending Send* state. If, on a last successful receive operation, the pipe is finally emptied, a notification is immediately sent to the producer since it had been in a pending send and wishes to be notified as soon as the pipe is emptied. At this point the pipe returns to the *Empty/Buffering* state and is serviceable only by the producer.

In scenario #2 above, when the pipe is initially in the *Empty/Buffering* state, suppose the consumer takes the first action of attempting to remove an element from the pipe. But this attempt will fail since the pipe is empty. However because the attempt was made, the pipe assumes that a pending receive is in effect by the consumer and so the pipe enters the *Empty/Pending Receive* state.

As long as the producer continues to successfully add elements without filling the pipe, the pipe remains in the *Empty/Pending Receive* state. This is because from the consumer's point of view the pipe is still empty since it has not been notified otherwise – that is, again, its notification is *deferred*. If, on a last successful send operation the pipe is finally filled, a notification is immediately sent to the consumer side since it had a pending receive in effect and wishes to be notified as soon as the pipe is filled. At this point the pipe enters the *Full/Buffering* state and is serviceable only by the consumer.

The difference between the *Full/Buffering* state and the *Full/Pending Send* state is that in the *Full/Buffering* state, when the pipe first becomes empty, this still *does not* trigger a notification to the producer, since the pipe assumes that because there is no pending send the producer is not *pushing* data to the pipe (see *push/pull operation* described above) and therefore it does not wish to be notified right away when the pipe becomes empty. Whereas in the *Full/Pending Send* state the producer is indeed *pushing* data to the pipe and wishes to be notified right way when the pipe becomes empty, so when the pipe first becomes empty, this *does* trigger a notification to the producer.

Conversely difference between the *Empty/Buffering* state and the *Empty/Pending Send* state is that in the *Empty/Buffering* state, when the pipe first becomes full, this still *does not* trigger a notification to the consumer, since the pipe assumes that because there is no pending receive the consumer is not *pulling* data from the pipe (see *push/pull operation* described above) and therefore it does not wish to be notified right away when the pipe becomes full. Whereas in the *Empty/Pending Receive* state the consumer is indeed *pulling* data from the pipe and wishes to be notified right way when the pipe becomes full, so when the pipe first becomes full, this *does* trigger a notification to the consumer.

Given the above definitions, and the state diagram in Figure 5.11, here is a list of conditions that cause pipes with *deferred pipe semantics* to become serviceable:

- The consumer side of a pipe will become serviceable under the following conditions (corresponding to yellow box states in Figure 5.11)

- if a producer-side blocking send operation is blocked or a non-blocking send operation failed to add all requested elements because the pipe became full (i.e. a `prod.try_send()[fails]` action), or
- if the pipe is in an empty/pending receive state and a producer-side blocking or non-blocking send operation successfully adds enough elements to fill the pipe (i.e. a `prod.try_send()[fills]` action), or
- if pipe goes into flush state because of a flush call from the producer-side.

- The producer side of a pipe will become serviceable under the following conditions (corresponding to blue box states in Figure 5.11):
  - if a consumer-side blocking receive operation was blocked or a non-blocking receive operation failed to remove all requested elements because the pipe became empty (i.e. a `cons.try_receive()[fails]` action), or
  - if the pipe is in a flush state and a consumer-side blocking or non-blocking receive operation successfully empties it, or
  - if the pipe is in a full/pending send state and a consumer-side blocking or non-blocking receive operation successfully consumes all `BUFFER_MAX_ELEMENTS` elements (i.e. a `cons.try_receive()[empties]` action).

#### 5.8.5.2.2 Immediate Pipe Semantics

An immediate pipe, designated as use model #2 above, is a transaction pipe with immediate visibility and default pipe notifications (empty, full, or flushed). This pipe configuration provides applications the flexibility to overlap send and receive operations while limiting the number or frequency of notifications for performance.

In this pipe model, there is a single shared view of the pipe for both the producer and the consumer. The producer can send elements to the pipe as long as it is not full. When the pipe becomes full while the producer is still attempting to send more data, the producer is no longer serviceable. The producer will become serviceable again when elements are removed from the pipe by the consumer. When the producer fails to send, the producer application can choose to block waiting for (1) pipe notification, or (2) events such as time events, DPI events, and host-emulator synchronization events. A failed blocking `send` operation would always be blocked waiting for the pipe (empty) notification. For a failed non-blocking `try_send`, the producer application chooses the means of waiting. It can be the same pipe notification for minimal notification overhead, or it can leverage other events to implement finer flow control of the data between producer and consumer to meet its specific needs without incurring performance overheads from frequent notifications.

Similarly, the consumer can receive elements from the pipe as long as it is not empty. When the pipe becomes empty while the consumer is still attempting to receive more data, the consumer is no longer serviceable. The consumer will become serviceable again when elements are added to the pipe by the producer. When the consumer fails to receive, the consumer application can also choose to block waiting for (1) pipe notification, or (2) events such as time events, DPI events, and host-emulator synchronization events. A failed blocking `receive` operation would always be blocked waiting for the pipe (full or flushed) notification. For a failed non-blocking `try_receive`, the consumer application chooses the means of waiting. It can be the same pipe notification for minimal notification overhead, or it can leverage other events to implement finer flow control of the data between producer and consumer to meet its specific needs without incurring performance overheads from frequent notifications.

Let's take a brief look at the specific states and transitions for an immediate pipe:

- Empty/Buffering and Full/Buffering: Consider a newly created immediate pipe. Initially, the pipe is empty and is placed in the Empty/Buffering state. The pipe would remain in this state as long as there is no failed send or receive operation. That is, as long as (1) the producer does not try to send more elements than the pipe can hold and (2) the consumer does not try to receive more elements than the producer has already placed in the pipe, data can continue to be transferred smoothly from producer to consumer without requiring any pipe notifications. Likewise, a pipe that has been

transitioned to the Full/Buffering state would remain in that state as long as there is no failed send or receive operation, allowing efficient data transfer without notification.

- Full/Pending Send: Consider the scenario when the pipe becomes full while the producer is still trying to send more data. This failed send operation causes the pipe to transition to the Full/Pending Send state. At this point, the producer is no longer serviceable and is typically suspended waiting for some trigger events. If the producer is waiting for pipe notification, then it will not wake up until the pipe is completely emptied by the consumer. A pipe that is emptied transitions to the Empty/Pending Receive state if the consumer is still trying to receive more data; otherwise, it transitions to the Empty/Buffering state. On the other hand, if the producer is waiting for a trigger event other than the pipe notification, it can wake up prior to the pipe becoming empty. If the producer is successful in sending the pending number of elements to the pipe, the pipe transitions to the Full/Buffering state, reflecting that there is no longer any pending send.
- Empty/Pending Receive: Consider the scenario when the pipe becomes empty while the consumer is still trying to receive more data. This failed receive operation causes the pipe to transition to the Empty/Pending Receive state. At this point, the consumer is no longer serviceable and is typically suspended waiting for some trigger events. If the consumer is waiting for pipe notification, then it will not wake up until the pipe is completely filled by the producer. A pipe that is filled transitions to the Full/Pending Send state if the producer is still trying to send more data; otherwise, it transitions to the Full/Buffering state. On the other hand, if the consumer is waiting for a trigger event other than the pipe notification, it can wake up prior to the pipe becoming full. If the consumer is successful in receiving the pending number of elements from the pipe, the pipe transitions to the Empty/Buffering state, reflecting that there is no longer any pending receive.

When using blocking pipe interface, both immediate pipes and deferred pipes (use model #1) behave the same way as to when an operation is unblocked. The difference lies in as to when an operation is blocked. For the deferred pipe, blocking occurs when the pipe is perceived to be full or empty. For the immediate pipe, blocking only occurs when the pipe is actually full when sending and empty when receiving.

When using non-blocking interface, an immediate pipe behaves similarly to a fifo (use model #3) as both pipe transaction models are configured for immediate visibility, allowing overlapping send and receive operations. The difference lies in as to when notification occurs when there is a pending receive or pending send. For the fifo, notification occurs as soon as an element is added to an empty pipe or an element is removed from a full pipe. For the immediate pipe, notification only occurs when the pipe is filled before the pending receive is satisfied or when the pipe is emptied before the pending send is satisfied.

#### 5.8.5.2.3 Fifo Semantics

The following paragraph explains how the pipe can be configured for use model #3 above (fifo semantics). These semantics are in effect when the threshold setting is 1 and the pipe has been configured for immediate visibility (VISIBILITY_MODE = 1).

The easiest way to describe fifo semantics to start with the assumption that there is some non-zero number of elements in the pipe and it is in one of the two *buffering* states.

At this point both the producer and consumer are serviceable meaning that elements can be both added and removed.

If the producer continues to add elements until the pipe fills, the next time it attempts to add an element, it will fail and the pipe will transition to the *Full/Pending Send* state meaning that a send was attempted, failed, and the producer is now assumed to have a pending send in effect. At this point, only the consumer is serviceable since elements can only be removed, not added. Once the consumer has removed at least one element, a notification is sent to the producer that indicates it can start adding elements again, and the pipe goes back to one of the two *buffering* states (specifically in this case, to state *empty/buffering*).

So now the pipe is back in the *empty/buffering* state.

At this point if the consumer continues to remove elements until the pipe is empty, the next time it attempts to remove an element, it will fail and the pipe will transition to the *Empty/Pending Receive* state meaning that a receive was attempted, failed, and the consumer is now assumed to have a pending receive in effect. At this point, only the producer is serviceable since elements can only be added, not removed. Once the producer has

added at least one element, a notification is sent to the consumer that indicates it can start removing elements again, and the pipe goes back to one of the two *buffering* states (specifically in this case, to state *full/buffering*).

Given the above definitions, and the state diagram in Figure 5.11, here is a list of conditions that cause pipes that have *fifo semantics* to become serviceable:

- Both the consumer side and the producer side of a pipe will become serviceable after any operation that causes the pipe to go into one of the two *buffering* states,
- In addition to above, the consumer side is serviceable after any operation that causes the pipe to go into the *full/pending send* or *flush* state,
- In addition to the above, the producer side becomes serviceable after any operation that causes the pipe to go into the *empty/pending send* state.

#### 5.8.5.2.4 Visibility Modes

The two different visibility modes of pipe operation described above together cover a wide range of use models for co-emulation needs, offering interoperability without (1) sacrificing ease-of-use and performance and (2) robbing application the flexibility to leverage other synchronization events, such as time events and DPI events, to more effectively control data flow between producer and consumer. The following table gives a simple at-a-glance comparison between the two modes.

| | Deferred Visibility Mode | Immediate Visibility Mode |
|---|---|---|
| Pipe view and visibility | Split views between producer and consumer. Producer's perspective and consumer's perspective are synchronized only at discrete notification point (empty, full, flush). | A single shared pipe: Pipe view is the same between producer and consumer, allowing access to pipe elements at any time. |
| Sample use models | Well defined "non-overlapped" or discrete data movements between producer and consumer: when notified pipe empty, full, or flushed. | Non-blocking pipe interface allows "overlapping" or continuous data movements between producer and consumer: when execution control is acquired.<br><br>Synchronization events other than pipe notifications such as DPI events can be leveraged to synchronize producer and consumer to meet application-specific needs (programmable granularity of synchronization for performance). |

*Given the above definitions, the number of notify callbacks is minimal and optimizes the performance by allowing the HDL side to run as long as possible.*
*The above semantics can be implemented using standard DPI calls.*
*The semantics do not depend on any infrastructure level capabilities outside a SCE-MI 2 Pipes compliant implementation.*
*The semantics of notify for a given pipe is only dependent on the state of the pipe that notify corresponds to.*
*The above semantics do not depend on whether a threaded application (such as SystemC) is used or not used on the C-side and whether the threaded application is linked or not with the HDL side simulation kernel. The semantics do not depend on whether the C side uses only non-blocking calls or if the C-side Pipes implemented blocking calls by using Pipes C-side non blocking calls in conjunction with the notify callback calls.*
*An attempt to send the message from the C-side on an input pipe where the specified number of elements was not entirely consumed by the pipe leaves a partial message on the C side. This message is recognized as "pending message" and the input pipe is set to a "pending message state". When the input pipe becomes empty, the notify callback (for the input pipe) will be called. This allows the C side to send the rest of the pending message or additional elements from the pending message into the pipe.*

*The C-side OK to send notify() callback will not be called when an input pipe becomes empty unless it is in a pending message, in a flush state or if the pipe is empty at simulation start.. This allows the HDL side to request a new message by using a blocking or non-blocking HDL side receive call or an empty pipe causing OK to send notify() callback to be called. These semantics also avoids sending any excessive notifications when the transactor consumed the pervious message and wishes to stay in idle state or when it knows that the test has ended.*
*The notify callback functions typically get registered at initialization time although they can be registered at any time. Additionally, previously registered notify function pointer can, at any time, be replaced with another.*
*The notify_ok_to_send() and notify_ok_to_receive() functions are callbacks that can be called directly or indirectly from within the thread-neutral implementation code to notify thread-aware application code on the C side when it is OK to send or receive. By implementing the bodies of these functions a user can put in thread specific code that takes some action such as posting to a SystemC sc_event.*
*So the key here is that the data transfer and query functions have thread-neutral implementation. And the notify functions are callbacks called from within thread-neutral code that can be filled in by some application wishing to create a thread-aware adapter that implements blocking send() and receive() functions.*

#### 5.8.5.3 Transaction pipes - non-blocking operations

On the C side, the non-blocking pipe access interface consists of *calling* functions and *callback* functions classified as *data transfer* operations, *query* operations, and *notify* operations for each pipe direction,

- Data transfer operations:

```
scemi_pipe_c_try_send()
scemi_pipe_c_try_receive()
scemi_pipe_c_try_flush()
```

- Query operations:

```
scemi_pipe_c_can_send()
scemi_pipe_c_can_receive()
scemi_pipe_c_in_flush_state()
```

- Notify operations:

```
(*scemi_pipe_c_notify_ok_to_send)()
(*scemi_pipe_c_notify_ok_to_receive)()
```

- User data storage and retrieval:

```
scemi_pipe_put_user_data()
scemi_pipe_get_user_data()
```

##### 5.8.5.3.1 Non-blocking data transfer operations

These are the basic non-blocking send/receive functions to access a transaction pipe from the C side. The *send* function is called to attempt to send transactions to an input pipe. The *receive* function is called to attempt to receive transactions from an output pipe. The *flush* function is called to attempt to flush an input pipe.

```
int scemi_pipe_c_try_send(   // return: #requested elements
                             //         that are actually sent
    void *pipe_handle,       // input: pipe handle
    int byte_offset,         // input: byte offset into data, below
    int num_elements,        // input: #elements to be sent
    const svBitVecVal *data, // input: data
    svBit eom );             // input: end-of-message marker flag

int scemi_pipe_c_try_receive(// return: #requested elements
                             //         that are actually received
    void *pipe_handle,       //  input: pipe handle
    int byte_offset,         //  input: byte offset into data, below
    int num_elements,        //  input: #elements to be read
    svBitVecVal *data,       // output: data
    svBit *eom );            // output: end-of-message marker flag

int scemi_pipe_c_try_flush(  // indication of whether flush was successful
    void *pipe_handle );     // input: pipe handle
```

Note: Take into account the following properties:

- The arguments, `pipe_handle num_elements, data,` and `eom` are identical to those described for the blocking function, `scemi_pipe_c_send()` described in section 5.8.4.1.1.
- The `byte_offset` argument is the byte offset within the data buffer designated by `data`.
- The `scemi_pipe_c_try_send/ scemi_pipe_c_try_receive` functions return the number of elements actually transferred.
- The `scemi_pipe_c_in_flush_state()` function returns 1 if the pipe is in the Flush state as described in section 5.8.4.3. Note that this information is needed for determining the exit condition of a blocking receive request.
- The `scemi_pipe_c_try_flush` function returns 1 if flush successful, 0 if not

By using the `byte_offset` argument, it is possible to create blocking functions that operate on unlimited data buffers on the C side. Despite the fact that pipe buffers are statically sized on the HDL-side with the `BUFFER_MAX_ELEMENTS` parameter and HDL-side `send()/try_send()` and `receive()/try_receive()` calls are limited to `PAYLOAD_MAX_ELEMENTS` elements per call (see section 5.8.5.1.3), on the C-side neither limitation exists and calls to blocking/non-blocking send or receive can be made with buffers of any size. Even if buffers in the internal implementation are of limited size, multiple calls to the non-blocking send/receive functions can be made until all the data is transferred. This makes it easy to build blocking data transfer functions that handle buffers of unlimited size on top of the non-blocking data transfer functions. Each call to the non-blocking function is made with the same base data buffer pointer but an increasing byte offset. Each call returns the actual number of elements transferred. This number can be translated to in increment amount for the byte offset to be passed to the next call in the loop - without changing the base `svBitVecVal *data` pointer.

#### 5.8.5.3.2 Non-blocking query operations

These are the status query functions for transaction pipes. They can be called by an application from the C side to see if a send or receive operation can be performed on a pipe.

```
int scemi_pipe_c_can_send(  // return: #elements that can be sent
    void *pipe_handle );    // input: pipe handle

int scemi_pipe_c_can_receive( // return: #elements that can be received
    void *pipe_handle );      // input: pipe handle

svBit scemi_pipe_c_in_flush_state( // return: whether pipe is in Flush state
    void *pipe_handle );           // input: pipe handle
```

Note the following properties:

- The arguments, `pipe_handle,` and `num_elements` are identical to those described for the blocking function, `scemi_pipe_c_send()` described in section 5.8.4.1.1.
- The functions return the number of elements that currently could be transferred in the pipe, i.e. the amount of room in an input pipe or number of elements available in an output pipe.

#### 5.8.5.3.3 Non-blocking notify operations

The following is a function declaration for notification callback functions that are used to notify the C side that an operation is possible on an input or output transaction pipe.

```
typedef void (*scemi_pipe_notify_callback)(
    void *context );       // input: C model context
```

The following is a type declaration of the notify callback handles.

```
typedef void * scemi_pipe_notify_callback_handle;
```

The notify callback handle type is opaque and the underlying type is implementation defined.

All notification callbacks must be registered using the following call:

```
scemi_pipe_notify_callback_handle scemi_pipe_set_notify_callback(
     void *pipe_handle,     // input: pipe handle
     scemi_pipe_notify_callback notify_callback,
                            // input: notify callback function
     void *notify_context,  // input: notify context
     int callback_threshold );
                        // input: threshold for invoking notify callback function
```

Note the following properties:

- `pipe_handle` - the handle identifying the specific pipe as derived from the unique combination of the HDL scope and the pipe ID (see section 5.8.3)
- `notify_callback` - the name of the user callback function being registered.
- `notify_context` - the user-defined context object to be passed to the function whenever it is called.
- `callback_threshold` - the user-defined callback threshold that must be met whenever the function is called.
    - `callback_threshold = 0`: When a callback function is registered with `callback_threshold = 0`, it corresponds to a persistent static callback.
        - Persistent: The callback function remains valid until it is explicitly de-registered by calling `scemi_pipe_clear_notify_callback` as described below.
        - Static Condition: The callback function is invoked called by the infrastructure only when pipe notification occurs as dictated by the pipe state diagram in Section 5.8.5.1.4
    - `callback_threshold > 0`: When a callback function is registered with `callback_threshold > 0`, it corresponds to a one-time dynamic callback.
        - One-time: The callback function remains valid only until it is invoked or it is explicitly de-registered by calling `scemi_pipe_clear_notify_callback`. Once invoked, the callback function is implicitly de-registered.
        - Dynamic Condition: The callback function is invoked by the infrastructure only when the callback_threshold condition is met or when an output pipe is flushed. This allows an application to dynamically specify the minimal or ideal condition when the callback should be invoked. The infrastructure is not required to invoke the callback function immediately upon meeting the threshold condition and is allowed to delay the callback until the next pipe notification. For an input pipe, dynamic callback can occur whenever `scemi_pipe_c_can_send() >= callback_threshold`. For an output pipe, dynamic callback can occur whenever `scemi_pipe_c_can_receive() >= callback_threshold` or when the pipe is flushed. For example, if an application wants

to receive 10 elements from an immediate output pipe with buffer size = 50, it can simply register a dynamic callback function with `callback_threshold = 10`. This gives the infrastructure freedom to invoke the callback once the pipe contains 10 elements, which can occur before pipe notification of a filled or flushed pipe. When the callback threshold is met prior to pipe notification threshold, there is a range of time when the callback can be invoked. The exact time of callback can vary from implementation to implementation, but must be repeatable. To ensure identical implementation-independent callback timing, callback_threshold should be set to the notification_threshold of the pipe.

The return value of `scemi_pipe_set_notify_callback` is a notify callback handle that uniquely identifies the callback. Any number of callbacks can be registered for each pipe. On notify events, if more than one callback is eligible to be called (as determined by the `callback_threshold` argument), the eligible callbacks will be called in the order of registration.

If a callback modifies the pipe state in any way, e.g., by removing elements from the pipe, any downstream callbacks will see the effect of this operation.

This handle can be used to remove a callback at any time using the following function:

```
void scemi_pipe_clear_notify_callback(
   scemi_pipe_notify_callback_handle notify_callback_handle );
        // input: notify callback handle
```

The `scemi_pipe_clear_notify_callback()` registers errors under the following conditions:

- The notify_handle value is not a value returned by a previous `scemi_pipe_set_notify_callback()` call.
- The callback was already removed by a prior call of `scemi_pipe_clear_notify_callback()`.
- The callback was registered with a callback_threshold value greater than 0 and the callback has expired and been removed by the infrastructure.

The following call can be used to retrieve a notify context object for a given pipe:

```
void *scemi_pipe_get_notify_context( // return: notify context object pointer
    scemi_pipe_notify_callback_handle notify_callback_handle);
                                    // input: notify handle
```

This call is useful to determine whether or not a notify context object is being established for the first time. It is guaranteed that this call will return a NULL if a notify context has not yet been established. This is useful for performing first time initializations inside pipe operations rather than requiring initialization to be performed outside of them. See the example of the blocking send function implementation in section 5.8.4.1.1 for an example of how this might be done.

Note the following properties:

- `pipe_handle` - the handle identifying the specific pipe as derived from the unique combination of the HDL scope and the pipe ID (see section 5.8.3)
- `notify_callback` - the name of the user callback function being registered.
- `notify_context` - the user defined context object to be passed to the function whenever it is called

The following call can be used to retrieve the *bytes per element* for a given pipe:

```
int scemi_pipe_get_bytes_per_element( // return: bytes per element
    void *pipe_handle );              // input: pipe handle
```

The following call can be used to retrieve *direction* of a given pipe:

```
svBit scemi_pipe_get_direction( // return: 1 for input pipe, 0 for output pipe
    void *pipe_handle );        // input: pipe handle
```

Appendix B: shows how dynamic callbacks can be used to implement a user defined blocking send function on top of a non-blocking send function.

#### 5.8.5.3.4 Non-blocking user data storage and retrieval

These are the functions for storing and retrieving user data for transaction pipes. These *put* and *get* functions provide data storage on a per-pipe basis, controllable by a user-defined key. They can be called by an application from the C side.

```
void scemi_pipe_put_user_data( // return: 0 if success, 1 if error
    void *pipe_handle,      // input: pipe handle
    void *user_key,         // input: user key
    void *user_data);       // input: user data

void *scemi_pipe_get_user_data(
    void *pipe_handle,      // input: pipe handle
    void *user_key);        // input: user key
```

The `scemi_pipe_put_user_data` function is used to store a user data pointer for later retrieval by `scemi_pipe_get_user_data`. The `user_key` is a user-defined key generated by the application. It should be unique from all other user keys to guarantee unique data storage. It is recommended that the address of static functions or variables in the application's C code be used as the user key. It is important that general applications should refrain from using NULL value as the user key. The NULL value user key is reserved for use only by the infrastructure blocking API. It is an error to call `scemi_pipe_put_user_data` with an invalid `pipe_handle` or with a NULL `user_data`.

The `scemi_pipe_get_user_data` function is used to retrieve a user data pointer that was previously stored by a call to `scemi_pipe_put_user_data`. This function returns NULL when called with an invalid `pipe_handle` or in the event that a prior call to `scemi_pipe_put_user_data` was never made with the same `pipe_handle` and `user_key` arguments. Otherwise, the stored user data pointer is returned.

#### 5.8.5.3.5 HDL Side Access

The bold text in the *input and output pipe* SystemVerilog interface declarations below shows declarations for the *non-blocking send and receive* functions which make up the API for the non-blocking operations of the HDL endpoint an input and output pipe:

```
interface scemi_input_pipe();
    parameter BYTES_PER_ELEMENT = 1;
    parameter PAYLOAD_MAX_ELEMENTS = 1;
    parameter BUFFER_MAX_ELEMENTS = <implementation specified>;
    parameter VISIBILITY_MODE = 0;
    parameter NOTIFICATION_THRESHOLD = BUFFER_MAX_ELEMENTS;

localparam PAYLOAD_MAX_BITS = PAYLOAD_MAX_ELEMENTS * BYTES_PER_ELEMENT * 8;

       ...


function int try_receive( // return: #requested elements
                          //         that are actually received
    input  int byte_offset,   //  input: byte_offset within data array
    input  int num_elements,  //  input: #elements to be read
    output bit [PAYLOAD_MAX_BITS-1:0] data, // output: data
    output bit eom );         // output: end-of-message marker flag
            <implementation goes here>
endfunction

function int can_receive();  // return: #elements that can be received
            <implementation goes here>
endfunction

    ...
endinterface

interface scemi_output_pipe();
    parameter BYTES_PER_ELEMENT = 1;
    parameter PAYLOAD_MAX_ELEMENTS = 1;
    parameter BUFFER_MAX_ELEMENTS = <implementation specified>;
    parameter VISIBILITY_MODE = 0;
    parameter NOTIFICATION_THRESHOLD = BUFFER_MAX_ELEMENTS;


  localparam PAYLOAD_MAX_BITS = PAYLOAD_MAX_ELEMENTS * BYTES_PER_ELEMENT * 8;

    ...

  function int try_send(   // return: #requested elements
                           //         that are actually sent
    input int byte_offset,    // input: byte_offset within data array
    input int num_elements,   // input: #elements to be sent
    input bit [PAYLOAD_MAX_BITS-1:0] data, // input: data
    input bit eom );          // input: end-of-message marker flag
            <implementation goes here>
  endfunction

  function int can_send();   // return: #elements that can be sent
            <implementation goes here>
  endfunction
    ...
endinterface
```

The parameters statically specified in both interfaces are, by definition, known at HDL compile time, and have the following meanings:

- `BYTES_PER_ELEMENT` defaults to 1 and is used to specify the number of bytes in each element of data that can be passed to the pipe.
- `PAYLOAD_MAX_ELEMENTS` defaults to 1 and is used to specify the number of elements that can be passed in any single call to a pipe `send()`, `receive()`, `try_send()`, or

try_receive() function. This limit pertains to the HDL-side only, not the C-side. It shall be considered an error to pass a num_elements value to any of these calls that exceeds PAYLOAD_MAX_ELEMENTS.

- BUFFER_MAX_ELEMENTS defaults to an implementation specified value and can be overridden by the user to specify the maximum number of elements a pipe can contain at compile time. In any pipe implementation, if a non-blocking try_send() operation is attempted from either the C-side or the HDL-side on any pipe containing BUFFER_MAX_ELEMENTS, that try_send() call must return a value of 0, indicating that no elements were transferred.
- VISIBILITY_MODE is set to 1 to place a pipe in immediate visibility mode or to 2 to place a pipe in deferred visibility mode. By default, this parameter is set to 0, denoting an illegal pipe interface with unspecified mode of operation.
- NOTIFICATION_THRESHOLD can be set to either 1 or BUFFER_MAX_ELEMENTS. The default notification threshold for a pipe is BUFFER_MAX_ELEMENTS. When the notification threshold is set to BUFFER_MAX_ELEMENTS, notifications due to send and receive operations that occur when the pipe becomes full or empty, i.e. after adding BUFFER_MAX_ELEMENTS elements to an empty pipe or removing BUFFER_MAX_ELEMENTS elements from a full pipe. When the notification threshold is set to 1, notifications due to send and receive operations occur immediately.

The highlighted try_receive(), can_receive(), try_send() and can_send() functions in the scemi_input_pipe and scemi_output_pipe SystemVerilog interface declarations shown above are the non-blocking receive/send and query functions to access a transaction pipe from the HDL-side endpoint. The try_receive() function is called to attempt to receive transactions from an input pipe. The try_send() function is called to attempt to send transactions to an output pipe. The can_receive() function can be called to query the number of receivable elements in an input pipe. The can_send() function can be called to query the number of elements can could currently be taken by the pipe.

Note the following properties:

- The arguments, num_elements, data, and eom are identical to those described for the blocking task, send() described in section. 5.8.4.2.1.
- The byte_offset argument is the byte offset within the data buffer designated by data.
- The try_send/try_receive functions return the number of elements actually transferred.
- The can_send/can_receive functions return the number of elements that could be potentially be transferred.

By using the byte_offset argument, it is possible to create blocking functions that operate on large data buffers on the HDL-side. Even if buffers in the internal implementation are of limited size, multiple calls to the non-blocking send/receive functions can be made until all the data is transferred. This makes it easy to build blocking data transfer functions that handle buffers of large size on top of the non-blocking data transfer functions. Each call to the non-blocking function is made with the same base data buffer pointer but an increasing byte offset. Each call returns the actual number of elements transferred. This number can be translated to an increment amount for the byte offset to be passed to the next call in the loop - without changing the base data vector.

Note: the HDL-side blocking send() and receive() functions do not implement the behavior described in the previous paragraph. Each call to send() and receive() is limited to PAYLOAD_MAX_ELEMENTS.

#### 5.8.5.4 Transaction pipes are deterministic

Transaction pipes are designed to guarantee deterministic time advance on the HDL side.

Determinism is guaranteed because consumption of data from an input pipe on the HDL side or production of data to an output pipe will always occur on the same clock cycles from one simulation to another or even from one implementation to another.

This property also holds true for all non-blocking pipe operations on the HDL side.

#### 5.8.5.4.1 Repeatable Behaviors for Pipes

Section 5.8.5.2 describes how a transaction pipe must behave individually, but does not specify how events of one pipe are processed with respect to events from other pipes and events from other sources such as time delay events, DPI events, and other synchronization events. When there are multiple events scheduled at the same simulation time, possibly from different threads, the processing order of these simultaneous events is unspecified but repeatable. Each implementation includes its own event scheduler which must produce simulation results that are repeatable. For example, at any given simulation time, there can be:

- Events from multiple pipes
- Multiple DPI function calls
- Multiple pipe events and multiple DPI function calls

Although simulations must be repeatable for a particular implementation, user applications should not rely on a particular processing order of simultaneous events to produce its desired/expected behavior.

### 5.8.5.5 Example Reference Implementations for Building a Complete Blocking API from the Non-Blocking API

The examples below are just one possible implementation of a reference model of how blocking access functions could be implemented in a given threading system. Implementations are not required to do it this way as long as they accomplish exactly the same semantics and functional operation. This means that the timing of notifications and returns from blocking calls must be identical for all implementations in accordance with the notification semantics detailed in the state diagram in section 5.8.5.1.4.

The reference implementations below assume a SystemC threading system.

#### 5.8.5.5.1 Blocking Send Reference Implementation

The following example shows how this can be used to implement the blocking send function on top of the non-blocking send function:

```
static void notify_ok_to_send_or_receive(
        void *context ) // input: notify context
{
    sc_event *me = (sc_event *)context;
    me->notify();
}

void scemi_pipe_c_send(
    void *pipe_handle,         // input: pipe handle
    int num_elements,          // input: #elements to be written
    const svBitVecVal *data,   // input: data
    svBit eom )                // input: end-of-message marker flag
{
    int byte_offset = 0, elements_sent;

    while( num_elements ){
        elements_sent =
            scemi_pipe_c_try_send(
                pipe_handle, byte_offset, num_elements, data, eom );

        // if( pipe is full ) wait until OK to send more
        if( elements_sent == 0 ){
            sc_event *ok_to_send = (sc_event *)
                scemi_pipe_get_user_data( pipe_handle, NULL );

            // if( notify ok_to_send context has not yet been set up ) ...
            if( ok_to_send == NULL ){
                ok_to_send = new sc_event;
                scemi_pipe_set_notify_callback( pipe_handle,
                   notify_ok_to_send_or_receive,
                   ok_to_send, 0 /*persistent callback*/ );
                scemi_pipe_put_user_data( pipe_handle, NULL, ok_to_send);
            }
            wait( *ok_to_send );
        }
        else {
            byte_offset += elements_sent
                * scemi_pipe_get_bytes_per_element( pipe_handle );
            num_elements -= elements_sent;
        }
    }
}
```

The execution remains inside this send function repeatedly calling `scemi_pipe_c_try_send()` until all elements in an arbitrarily sized user buffer have been transferred. Each call to `scemi_pipe_c_try_send()` returns the number of elements transferred in that call.

That number is used to increment the `byte_offset` within the user's `data` buffer.

Between the calls, the thread waits on the `ok_to_send` event and suspends execution until there is a possibility of more room in the pipe for data.

If this event has not yet been created, it is created and passed as the context when the notify callback is registered for the first time.

#### 5.8.5.5.2 Blocking Flush Reference Implementation

In a similar fashion, a blocking flush call can be implemented over the non-blocking flush call as follows:

```
void scemi_pipe_c_flush(
    void *pipe_handle )      // input: pipe handle
{
    sc_event *ok_to_flush
        = (sc_event *)scemi_pipe_get_user_data( pipe_handle, NULL );

    // if( notify ok_to_flush context has not yet been set up ) ...
    if( ok_to_flush == NULL ){
        ok_to_flush = new sc_event;
        scemi_pipe_set_notify_callback( pipe_handle,
            notify_ok_to_send_or_receive,
            ok_to_flush, 0 /*persistent callback*/ );
        scemi_pipe_put_user_data( pipe_handle, NULL, ok_to_flush);
    }

    while( !scemi_pipe_c_try_flush(pipe_handle) )
        wait( *ok_to_flush );
}
```

Note that for this implementation the notify callback function, `notify_ok_to_send_or_receive()` shown in the previous section, can be reused without modification.

To understand better how `scemi_pipe_c_try_flush()` works it should be noted that this is not always implicitly successful. By contrast, blocking `scemi_pipe_c_flush()` is always implicitly successful.

The whole purpose of non-blocking calls in general is to test for success within a blocking function and not return if the success is not there.

The easiest way to look at *flush* is to look at *send*.

A *blocking send* is always implicitly successful. It can be implemented using a loop of *non-blocking sends* that are not always implicitly successful. The example of an implementation of `scemi_pipe_c_send()` shown above illustrates this.

The *blocking flush* works in exactly the same way.

*The non-blocking flush is probably something users should never use. It is mainly there to complete the thread-neutral API that provides full functionality for implementation of higher level thread-aware API calls that are blocking.*

#### 5.8.5.5.3 Blocking Receive Reference Implementation

Finally, to complete the full blocking API reference model, the blocking receive function can be implemented over the non-blocking receive call as follows:

```
void scemi_pipe_c_receive(
    void *pipe_handle,       //  input: pipe handle
    int num_elements,        //  input: #elements to be read
    int *num_elements_valid, // output: #elements that are valid
    svBitVecVal *data,       // output: data
    svBit *eom )             // output: end-of-message marker flag (and flush)
{
    int byte_offset = 0, elements_received;
    svBit is_in_flush_state = 0;
    *num_elements_valid = 0;
    *eom = 0;

 while( num_elements ){
     is_in_flush_state = scemi_pipe_c_in_flush_state( pipe_handle);
        elements_received =
            scemi_pipe_c_try_receive( pipe_handle, byte_offset,
                num_elements, data, eom );
     *num_elements_valid += elements_received;

     if (*eom || is_in_flush_state)
         return;

     num_elements -= elements_received;
     if (num_elements > 0) {   /* Wait until OK to receive more */
         byte_offset += elements_received
                * scemi_pipe_get_bytes_per_element( pipe_handle );
         sc_event *ok_to_receive
                = (sc_event *)scemi_pipe_get_user_data( pipe_handle, NULL );
         if( ok_to_receive == NULL ){  /* set up ok_to_receive if NULL */
                ok_to_receive = new sc_event;
                scemi_pipe_set_notify_callback( pipe_handle,
                    notify_ok_to_send_or_receive,
                    ok_to_receive, 0 /*persistent callback*/ );
                scemi_pipe_put_user_data( pipe_handle, NULL, ok_to_receive);
            }
            wait( *ok_to_receive );
     }
    }
}
```

The structure of the blocking receive call is similar to the blocking send except that special provision must be made to properly update the `num_elements_valid` output argument which does not exist in the send call. As described above, `num_elements_valid` can be less than `num_elements` requested in the case where the producer has done a flush.

Additionally, it is necessary to detect a condition in which a flush has occurred but the pipe is empty. This can happen if a producer wrote some elements to the pipe but did not flush until some clock cycles later. Meanwhile, if control yielded to the C side and the C-side consumed those elements, it would not have known yet that a flush was going to occur so it would have looped back around to execute the `wait( *ok_to_receive )` statement. But when the flush finally does occur, the pipe would have been empty. In this case the blocking receive function must return immediately since its original request has technically been satisfied.

#### 5.8.5.6 Query of buffer depth

By default, depth of a transaction pipe is assumed to be implementation defined. The user can query (but not override) this default on any individual pipe with the following C-side calls:

```
int scemi_pipe_get_depth( // return: current depth (in elements) of the pipe
    void *pipe_handle );  // input: pipe handle
```

Note the following properties:

- The pipe_handle is the handle identifying the specific pipe as derived from the unique combination of the HDL scope and the pipe ID (see section 5.8.2).
- The depth of any pipe is always expressed in elements. Each element's size is determined by the statically defined BYTES_PER_ELEMENT in the HDL endpoint instantiation of a pipe.

# Appendix A: Macro based use-model Tutorial

*(informative)*

## A.1 Hardware side interfacing

The hardware side interface of the SCE-MI consists of a set of parameterized macros which can be instantiated inside transactors that are to interact with the SCE-MI infrastructure. The macros are parameterized so, at the point of instantiation, the user can easily specify crucial parameters that determine the dimensions of the SCE-MI bridge to software. It is the job of the infrastructure linker to learn the values of these parameters, customize implementation components, and insert them underneath the macros accordingly.

The following four macros fully characterize how the hardware side interface of the SCE-MI is presented to the transactors and the DUT:

- `SceMiMessageInPort` macro
- `SceMiMessageOutPort` macro
- `SceMiClockControl` macro
- `SceMiClockPort` macro

Any number of these macros can be instantiated as needed. One `SceMiMessageInPort` macro shall be instantiated for each required message input channel and one `SceMiMessageOutPort` macro for each output channel. Message port macro bit-widths are parameterized at the point of instantiation.

Exactly one `SceMiClockPort` macro is instantiated for each defined clock in the system. This `SceMiClockPort` macro instance shall (via a set of parameters) fully characterize a particular clock. The `SceMiClockPort` macro is instantiated at the top level and provides a controlled clock and reset directly to the DUT. The `SceMiClockPort` macro instance is named and assigned a reference `ClockNum` parameter that is used to associate it with one or more counterpart `SceMiClockControl` macros inside one or more transactors. The `SceMiClockControl` macro is used by its transactor for all clock controlling operations for its associated clock. These two macros are mutually associated by the `ClockNum` parameter and every `SceMiClockPort` macro shall have a minimum of one `SceMiClockControl` macro associated with it.

The infrastructure linker (not the user) is responsible for properly hooking up these, essentially empty, macro instances to the internally generated SCE-MI infrastructure and clock generation circuitry.

### Required dimensions

The following parameters, specified at the points of instantiation of the macros, fully specify the required dimensions of the SCE-MI infrastructure components (see 5.3.1 for more details):

- number of transactors
- number of input and output channels
- name and width of each channel
- number of controlled clocks
- name, clock ratio, and duty cycle of each controlled clock

### Hardware side interface connections

Figure A.1 shows a simple example of how a transactor and DUT might connect to the hardware side interface of the SCE-MI.

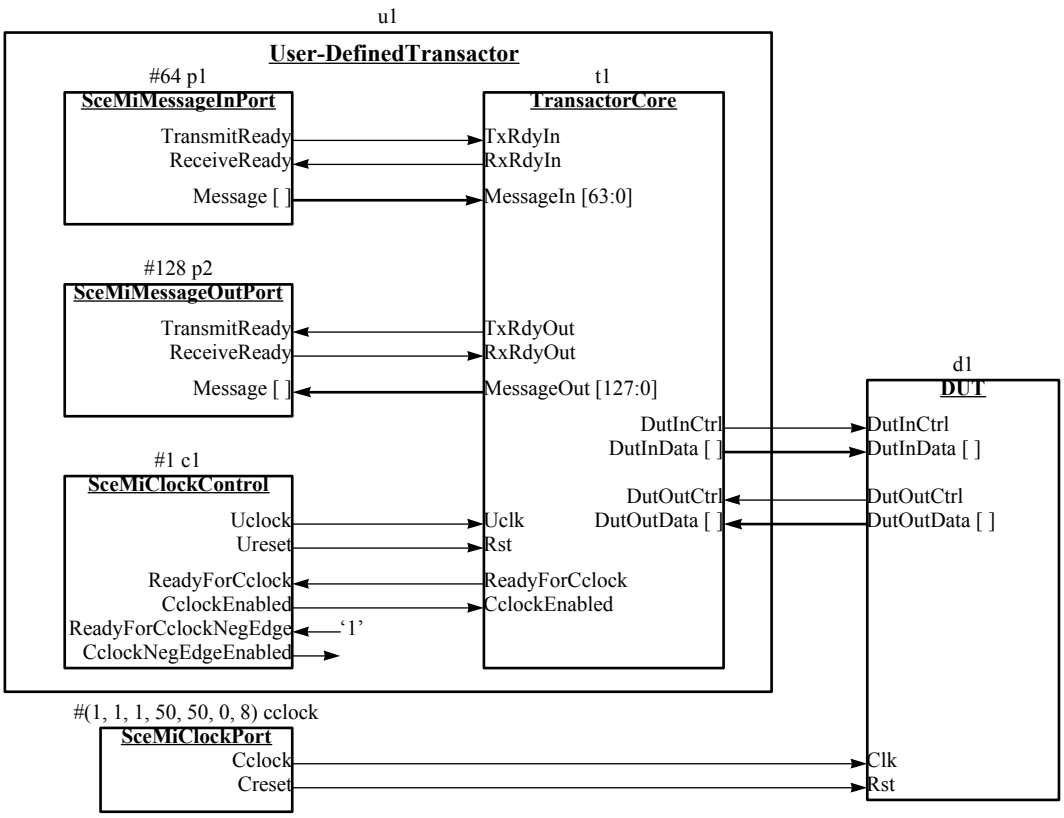



Figure A.1 Connection of SCE-MI macros on hardware side to transactor and DUT

This example features a single transactor interacting with a DUT and interfacing to the software side through a `SceMiMessageInPort` and a `SceMiMessageOutPort`. In addition, it defines a single clock that is controlled by the transactor internally using the `SceMiClockControl` macro. This clock drives the DUT from the top level through a `SceMiClockPort` macro.

A key point of this example is only the transactor implementor (see Section 4.3) needs to be aware of all the SCE-MI interface macros (except for the `SceMiClockPort`). Because the transactor encapsulates the message port macros and the `SceMiClockControl` macro, the end-user only has to be aware of how to hook-up to the transactor itself and to the `SceMiClockPort` macro.

# SceMiClockPort macro instantiation

The `SceMiClockPort` macro instantiation is where all the clock parameters are specified. The numbers shown in the component instantiation label (see Figure A.1) as:

```
#(1, 1, 1, 50, 50, 0, 8) cclock
```

map to the parameters defined for the `SceMiClockPort` macro (see 5.2.4). They are summarized here:

```
ClockNum = 1
RatioNumerator = 1
RatioDenominator = 1
DutyHi = 50
DutyLo = 50
Phase = 0
ResetCycles = 8
```

Of these parameters, the `ClockNum` parameter is used to uniquely identify this particular clock and also to associate it with its one or more counterpart `SceMiClockControl` macros, which shall be parameterized to the same

`ClockNum` value, in this case 1. In addition to learning the clock specification parameters, the infrastructure linker also learns the name of each clock by looking at the instance label of each `SceMiClockPort` instance, in this case `cclock`.

Similarly, message ports have a parameterized `PortWidth` parameter.

## Analyzing the netlist

To summarize, the infrastructure linker learns the following specific information from analyzing this netlist.

- It has a single transactor called Bridge.u1 (assuming top level module is called Bridge).
- It has a single "divide-by-1" controlled clock called cclock.
- The controlled clock has a 1/1 ratio which, when enabled, is ideally (depending on implementation) the same frequency as the uncontrolled clock.
- The controlled clock is parametrized to 50/50 duty cycle with 0 phase shift (a user can also specify a don't care duty cycle - see 5.2.4.3 for details).
- The controlled reset is parametrized to eight controlled clock cycles of reset.
- It has a single SceMiMessageInPort called p1, parametrized to bit-with of 64.
- It has a single SceMiMessageOutPort called p2, parametrized to bit-width of 128.

A more complicated example which involves two transactors and three clocks is shown in Appendix C.

# A.2 The Routed tutorial

The Routed tutorial documents a real-life example which uses the SCE-MI to interface between untimed software models modeled in SystemC, and hardware models of transactors and a DUT modeled in RTL Verilog. This tutorial illustrates how the macro-based interface of the SCE-MI can be applied in a multi-threaded SystemC environment. It assumes some familiarity with the concepts of SystemC including abstract ports, autonomous threads, slave threads, module and port definition, and module instantiation and interconnect. See Reference [B2] for a description of these concepts.

## What the design does

The Routed design is a small design that simulates air passengers traveling from Origins to Destinations by traversing various interconnected Pipes and Hubs in a `RouteMap`. In this design, the Origins and Destinations are the transactors and the `RouteMap` model is the DUT. Each Origin transactor interfaces to a `SceMiMessageInPort` to gain access to messages arriving from the software side. Each Destination transactor interfaces to a `SceMiMessageOutPort` to send messages to the software side. There is also an OrigDest module that has both an Origin and Destination transactor contained within it.

The "world" consists of these Origins:

```
Anchorage, Cupertino, Noida, SealBeach, UK, or Waltham,
```

and these Destinations:

```
Anchorage, Cupertino, Maui, SealBeach, or UK.
```

Travel from any Origin to any Destination is possible by traversing the `RouteMap` (DUT) containing the following Pipe-interconnected Hubs:

```
Chicago, Dallas, Newark, SanFran, or Seattle.
```

Each controlled-clock cycle represents one hour of travel or layover time.

Figure A.2 shows how the Routed world is interconnected. The numbers shown by the directed arcs are the travel time (in hours) to travel the indicated Pipe. Layover time in each Hub is two hours.

The `RouteMap` is initialized by injecting `TeachRoute` messages for the entire system through the Waltham Origin transactor. Each `TeachRoute` message contains a piece of routing information addressed to a particular Hub to load the route into its `RouteTable` module (see Figure A.5). Using this simple mechanism, the software-side `RouteConfig` model progressively teaches each Hub its routes (via Waltham) so that it can, in turn, pass additional `TeachRoute` tokens to Hubs more distant from Waltham. In other words, by first teaching closer hubs,

the `RouteMap` learns to pass routes bound for more distant hubs. This process continues until the entire mesh is initialized, at which point it is ready to serve as a backbone for all air travel activity.

After initiating the route configuration, the testbench then executes the itineraries of four passengers over a period of 22 days. Each itinerary consists of several legs, each with a scheduled departure from a specified Origin and a specified Destination. The scheduled leg is sent as a message token to its designated Origin transactor. The transactor needs to count the number of clocks until the specified departure time before sending the token into the `RouteMap` mesh.

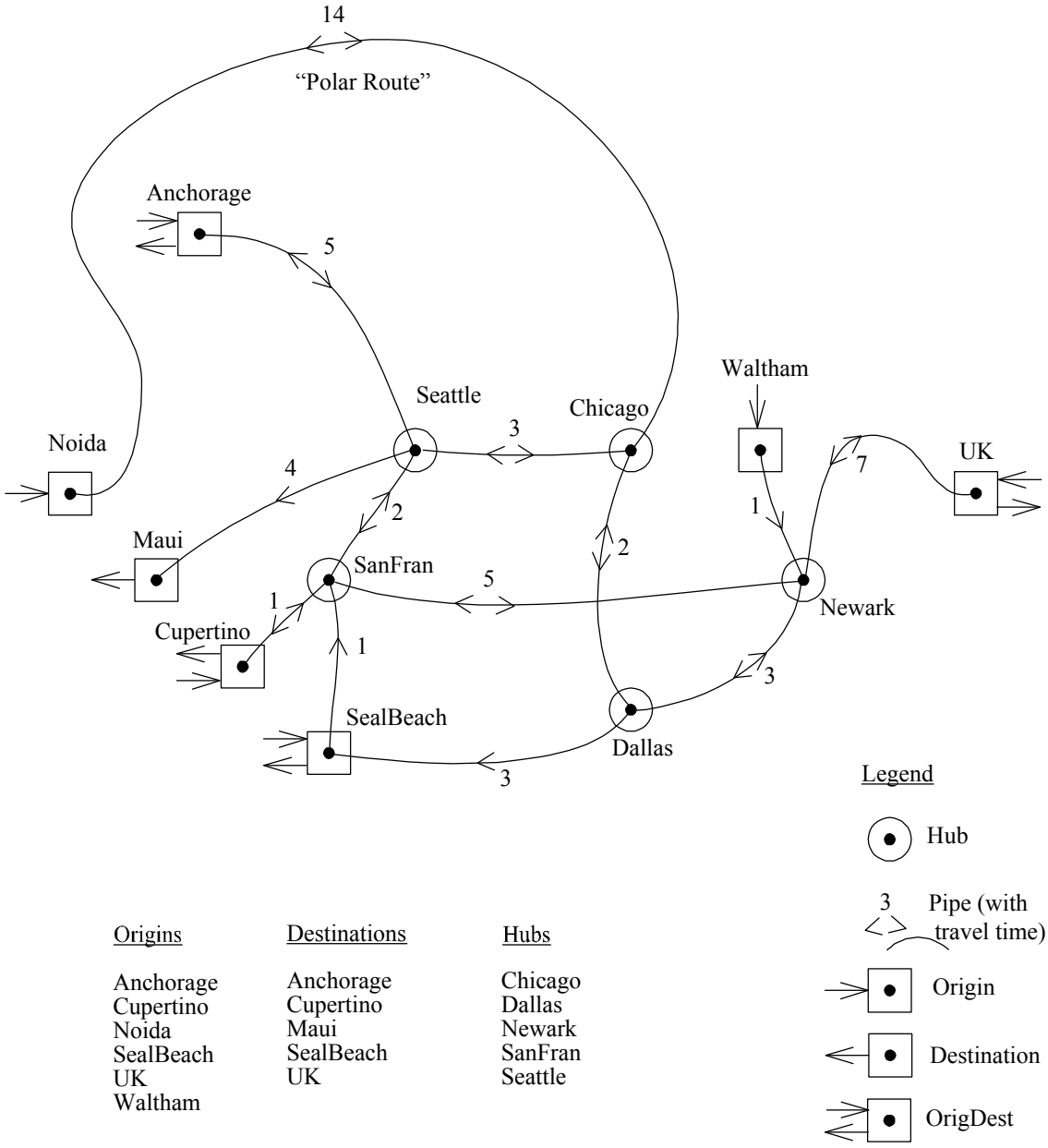



Figure A.2 The `Routed` world

## System hierarchy

The hierarchy of the whole system is textually shown in the following subsections.

### Software side hierarchy

The software side hierarchy of models is as follows.

```
System
Testbench
Calendar <--> ClockAdvancer
Scheduler <--> OrigDest, Origin, Destination
RouteConfig
SceMiDispatcher
```

Notice the interactions shown between the `Calendar` and `Scheduler` software side models and the `OrigDest`, Origin, and Destination hardware side models occur over SCE-MI message channels.

### Hardware side hierarchy

The hierarchy of the hardware side components instantiated under the Bridge netlist is shown here.

Bridge

```
SceMiClockPort
OrigDest anchorage, cupertino, sealBeach, UK
    Origin
        SceMiMessageInPort
        SceMiClockControl
    Destination
        SceMiMessageOutPort
        SceMiClockControl

Origin noida, waltham

Destination maui

RouteMap
    Hub chicagoHub, dallasHub, newarkHub, sanFranHub, seattleHub
        Funnel
        Nozzle
        RouteTable

    Pipe

ClockAdvancer
    SceMiMessageInPort
    SceMiMessageOutPort
    SceMiClockControl
```

Notice at the Bridge level, only the `SceMiClockPort` macro, transactor components, and the DUT appear. The `SceMiMessageInPort`, `SceMiMessageOutPort`, and `SceMiClockControl` macros are encapsulated within the Origin and Destination transactors. The `ClockAdvancer` transactor has both message input and output ports, in addition to the required `SceMiClockControl` macro.

## Hardware side

The hardware side of this example consists of a bridge netlist which instantiates the DUT, transactors, and the clock ports. The transactors in turn communicate with the DUT and instantiate the message port macros, as shown in Figure A.3.

### Bridge

The bridge between the hardware and software side of the design is depicted in Figure A.3. Notice this diagram more or less follows the structure of the generalized abstraction bridge shown in Figure 4.4 in section 4.4.2. The design uses 13 message channels in all: two message (input and output) channels for the `Calendar <-> ClockAdvancer` connection, six message input channels for the `Scheduler <-> Origin` connections, and five output channels for the `Scheduler <-> Destination` connections.

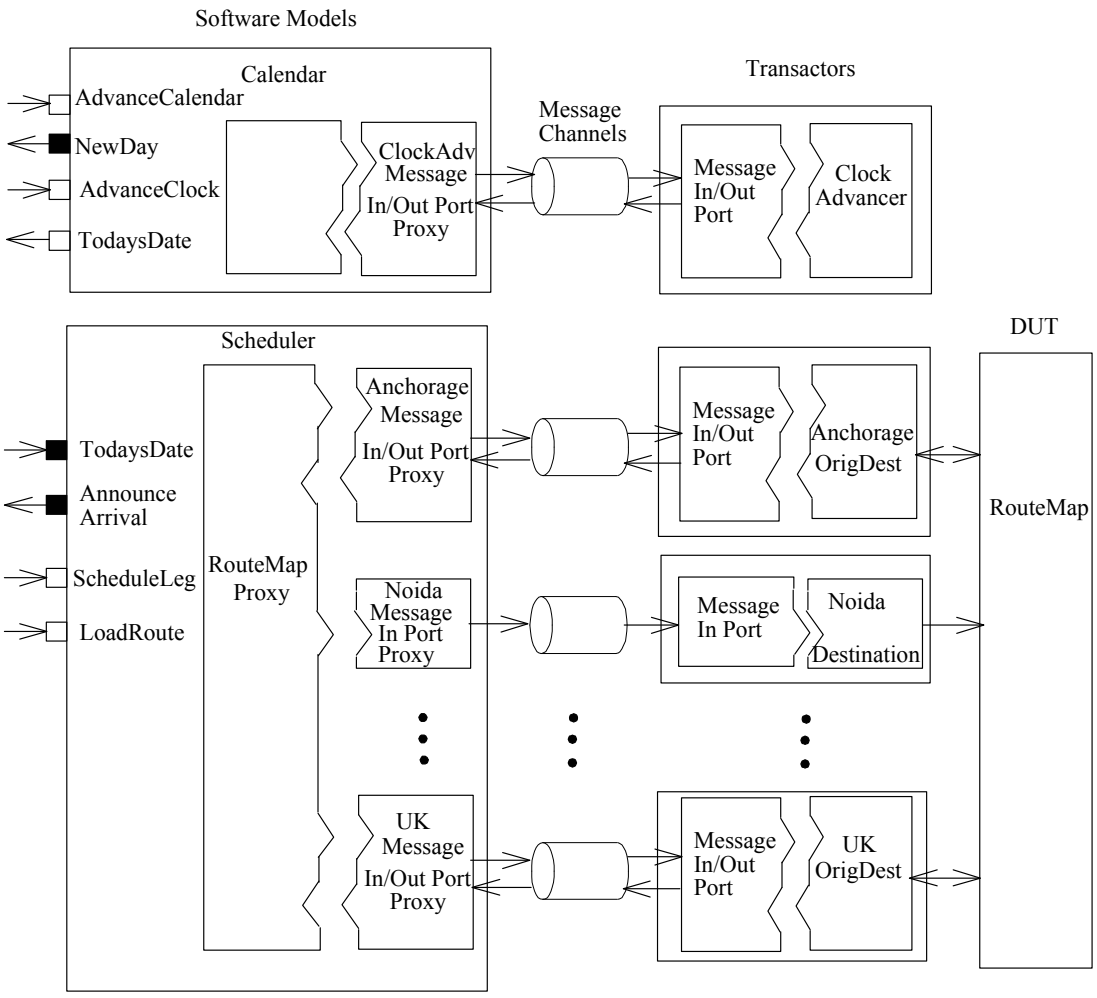



Figure A.3 The bridge

The two software models that interact with the hardware side are the Calendar model and the Scheduler model. These models encapsulate message port proxies which give them direct access to the message channels leading to the Origin and Destination transactors on the hardware side. These two software models are the only ones that are aware of the SCE-MI link. They converse with the other models through SystemC abstract ports.

On the hardware side, there is a set of Origin and Destination transactors which service the message channels that interface with the Scheduler and route tokens to or from the DUT. Some locations, such as Anchorage and the UK, are both Origin and Destination (called `OrigDest`).

In addition, there is a `ClockAdvancer` transactor which interfaces directly with the Calendar model. The `ClockAdvancer` is a stand-alone transactor which does not converse with the DUT. Its only job is to allow time to advance a day at a time (see A.2.3.5 for more details).

## DUT and transactor interconnect

Figure A.4 shows a representative portion of the `RouteMap` to illustrate how it interconnects DUT components to form the `RouteMap` mesh.

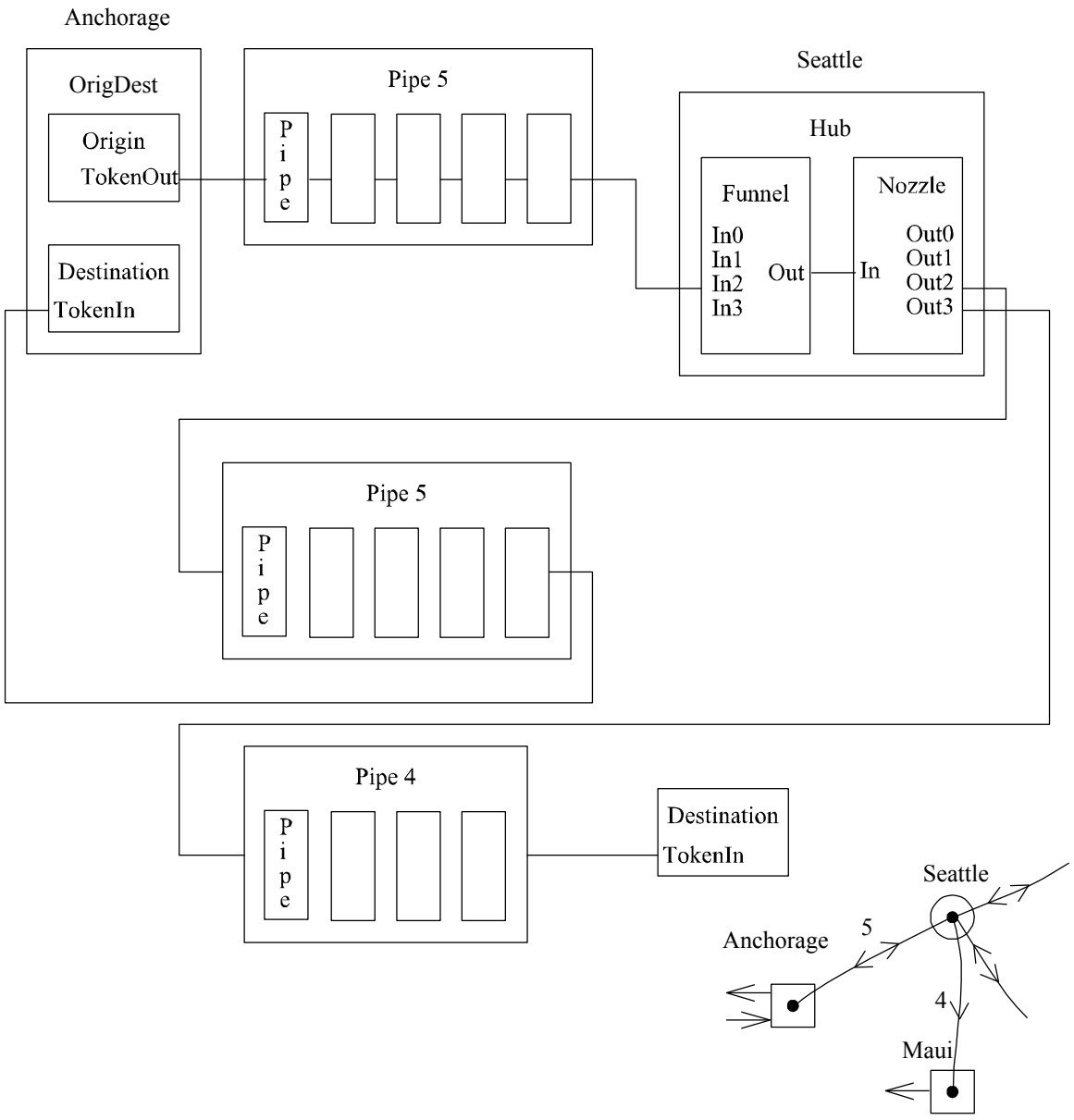



Figure A.4 DUT and transactor interconnect

Pipes are inserted between two Hubs or between an Origin or Destination transactor and a Hub. Longer Pipes can be created by cascading primitive one-hour Pipes to form the proper length. Each Pipe primitive represents one hour of travel (one clock). In this diagram, a Pipe4 model is inserted between the Seattle Hub and Maui Destination for a four-hour flight leg. Since travel can occur in either direction between Anchorage and Seattle, a Pipe5 is inserted between them for each direction.

## DUT and transactor components

Figure A.5 shows the structure of the DUT and transactor components.

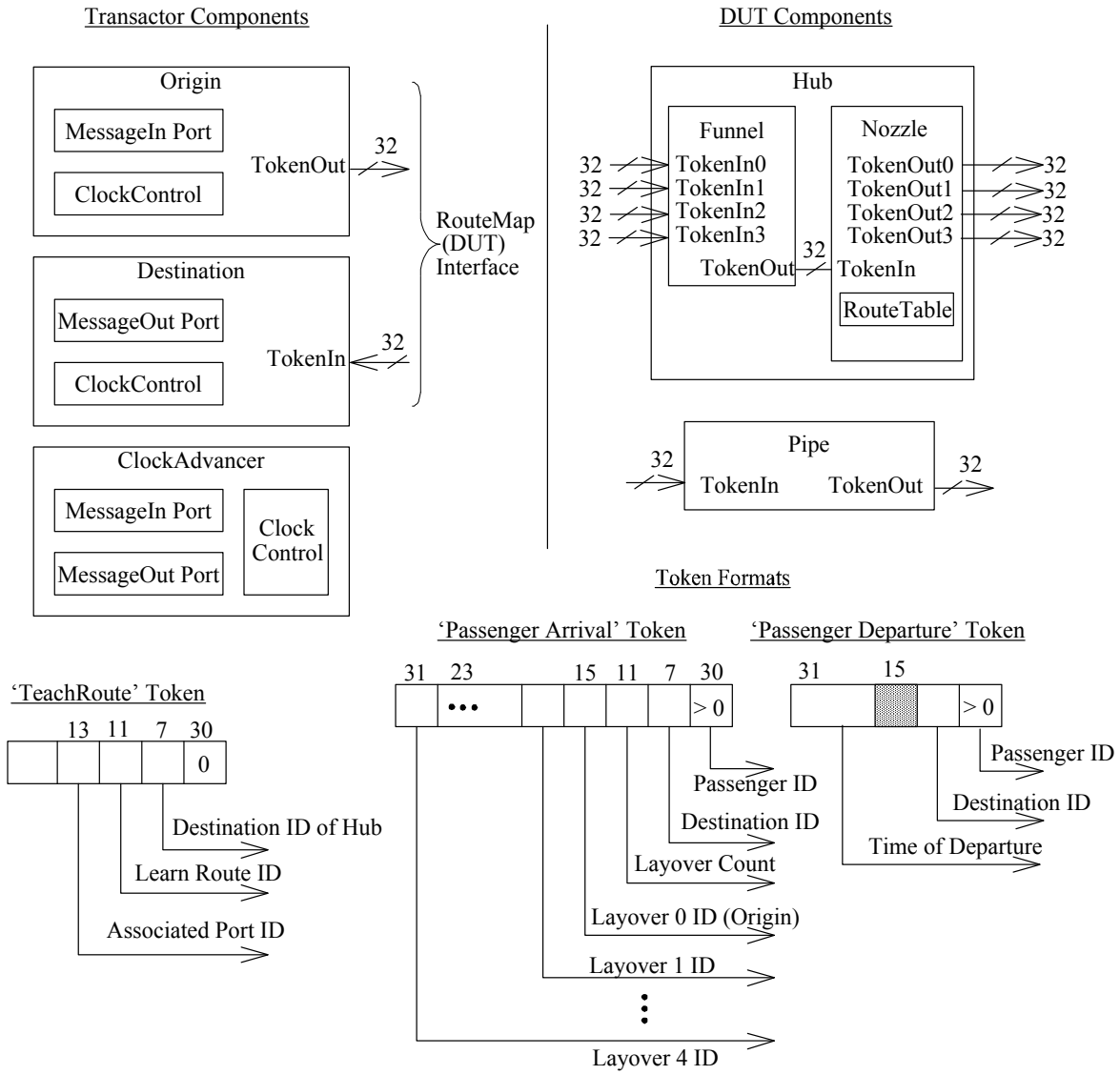



Figure A.5 DUT and transactor components

Each Origin transactor contains a clock-control macro and a message-input port macro to receive departure tokens from the Scheduler on the software side. Each received token is passed to the `TokenOut` port when the scheduled departure time has matured. Although the Origin transactor has a clock-control macro, it does not actively control the clock. Its only use of the clock-control macro is to monitor the `ReadyForCclock` signal to know on which `uclocks` the `cclock` is active, so it can properly count `cclocks` until the scheduled departure time of a pending departure token.

Each Destination transactor contains a clock-control macro and a message-output port macro to send arrival tokens back to the Scheduler on the software side. The arrival tokens represent a passenger emerging from the `RouteMap` mesh and arriving at a Destination through its `TokenIn` port. See A.2.3.4 for a detailed description of the Destination transactor. This transactor was chosen because it provides a simple example of clock control and message port interfacing.

Each token is a 32-bit vector signal. There are no handshakes in the system. Rather, the tokens are "self announcing." Normally, 0's (zeroes) are clocked through the mesh so if, on any given cycle, a Hub or Destination senses a non-zero value on its input port, it knows it has received a token that needs to be processed.

Token formats are also shown in Figure A.5. A departure token contains the passenger ID, destination ID, and scheduled time of departure. As the departure token travels through the mesh, it collects layover information consisting of the IDs of all the Hubs encountered before reaching its Destination, which is transformed into an arrival token. The arrival token then has a complete record of layover information which is passed back to the software side and displayed to the console.

A Hub consists of a Funnel which accepts tokens from a maximum of four different sources and a Nozzle which routes a token to a maximum of four different destinations. The Nozzle contains a small `RouteTable` which is initialized at the beginning of the simulation with routing information by receiving `TeachRoute` tokens.

## The Destination transactor: interfacing with the DUT and controlling the clock

The Destination transactor accepts tokens arriving from a point-of-exit on the `RouteMap` and passes them to the message output port.

The Destination transactor uses clock control to avoid losing potentially successive tokens arriving from the `RouteMap` (through the `TokenIn` input) to this destination portal. It de-asserts the `readyForCclock` if a token comes in, but the message output port is not able to take it because of tokens simultaneously arriving at other destination portals. This way, it guarantees that the entire `RouteMap` is disabled until all tokens are off-loaded from the requesting Destination transactors.

The Verilog source code for the Destination transactor is shown in the following listing.

```
module Destination (
      //inputs                              outputs
      //--------------------------          ----------------------------
        // DUT port interface
        TokenIn );
        input [31:0] TokenIn;
//  {
    wire [3:0] destID;
    reg readyForCclock;
    reg outTransmitReady;
    reg [31:0] outMessage;

    assign destID = TokenIn[7:4];

    SceMiClockControl sceMiClockControl(
      //Inputs                            Outputs
      //----------------------------      ----------------------------
                                          .Uclock(uclock),
                                          .Ureset(ureset),
        .ReadyForCclock(readyForCclock),    .CclockEnabled(cclockEnabled),
       .ReadyForCclockNegEdge(1'b1),      .CclockNegEdgeEnabled() );

    SceMiMessageOutPort #32 sceMiMessageOutPort(
      //Inputs                            Outputs
      //----------------------------      ----------------------------
        .TransmitReady(outTransmitReady),  .ReceiveReady(outReceiveReady),
        .Message(outMessage) );

    always@( posedge uclock ) begin // {
        if( ureset == 1 ) begin
            readyForCclock <= 1;
            outMessage <= 0;
            outTransmitReady <= 0;
        end
        else begin // {
          // if( DUT clock has been disabled )
          // It means that this destination transactor is waiting to
          // unload its pending token and does not want to re-enable the
          // DUT until that token has been offloaded or else it may
          //     loose arriving tokens in subsequent DUT clocks.
            if( readyForCclock == 0 ) begin

                // When the SceMiMessageOutPort finally signals acceptance
                // of the token, we can re-enable the DUT clock.
                if( outReceiveReady ) begin
                    readyForCclock <= 1;
                    outTransmitReady <= 0;
                end
            end
            else if( cclockEnabled && destID != 0 ) begin
                outMessage <= TokenIn;
                outTransmitReady <= 1;

                // if( token arrives but portal is not ready )
                //     Stop the assembly line ! (a.k.a. disable the DUT)
                if( outReceiveReady == 0 )
                    readyForCclock <= 0;
            end
            else if( outTransmitReady == 1 && outReceiveReady == 1 )
               outTransmitReady <= 0;
        end // }
    end // }
endmodule // }
```

### The ClockAdvancer transactor: controlling time advance

The `ClockAdvancer` transactor simply counts controlled clocks until the requested number of cycles has transpired, then sends back a reply transaction.

The Verilog source code for the `ClockAdvancer` is listed here.

```
 module ClockAdvancer(
      //inputs                          outputs
      //--------------------------      ----------------------------
        Uclock );

        parameter ClockNum = 1;
        parameter SampleWidth = 32;
//  {
    // Internal signals
    wire [31:0] advanceDelta;
    reg  [31:0] cycleCount;

    wire inReceiveReady;
    reg outTransmitReady;
    reg readyForCclock;
    wire [SampleWidth-1:0] inMessage, outMessage;

    assign inReceiveReady = 1;
    assign advanceDelta = inMessage[31:0];
    assign outMessage = 0;

    SceMiClockControl #(ClockNum) sceMiClockControl(
      //Inputs                          Outputs
      //----------------------------    ----------------------------
                                       .Uclock(uclock), .Ureset(ureset),
      .ReadyForCclock(readyForCclock), .CclockEnabled(cclockEnabled),
      .ReadyForCclockNegEdge(1'b1),    .CclockNegEdgeEnabled() );

    SceMiMessageInPort #(SampleWidth)32 sceMiMessageInPort(
      //Inputs                          Outputs
      //----------------------------    ----------------------------
       .ReceiveReady(inReceiveReady),   .TransmitReady(inTransmitReady),
                                        .Message(inMessage) );

    SceMiMessageOutPort #32 sceMiMessageOutPort(
      //Inputs                           Outputs
      //----------------------------     ----------------------------
     .TransmitReady(outTransmitReady),  .ReceiveReady(outReceiveReady),
     .Message(outMessage) );

    always @(posedge uclock) begin // {
        if (ureset) begin
            outTransmitReady <= 0;
            cycleCount <= 0;
            readyForCclock <= 0;
        end

        else begin // {
            // Start operation command
            if( inTransmitReady &&
                    !outTransmitReady ) begin
                cycleCount <= advanceDelta;
                readyForCclock <= 1;
            end

            if( readyForCclock && cclockEnabled ) begin
                if (cycleCount == 1)  begin
                    outTransmitReady <= 1;
                    readyForCclock <= 0;
                end
                cycleCount <= cycleCount - 1;
            end
```

```
            if (outReceiveReady == 1 && outTransmitReady == 1)
                outTransmitReady <= 0;
        end // }
    end // }
endmodule // }
```

Notice the `SceMiClockControl` macro references the same `cclock` as that in the Destination transactor (i.e., it uses the default `ClockNum=1`). This means the `ClockAdvancer` and the Destination transactor share in the control of the same `cclock`. In fact there is only one `cclock` in the entire system that is specified at the default 1/1 ratio.

Also, although the `ClockAdvancer` handshakes with the message output port, the data that it sends is always 0. This is because the only thing that the software side needs from the `ClockAdvancer` is the cycle stamp, which is automatically included in each message output response (see 5.4.7.1).

# The software side

The software side of the Routed design is written completely in SystemC and C++. It is compiled as an executable program that links with the SCE-MI software side.

## The System model: interconnect of SystemC modules

The System model is the top level "software netlist" of SystemC modules (`SC_MODULE()`). It specifies the construction and interconnect of the component models as well. A block diagram of the System model is shown in Figure A.6.

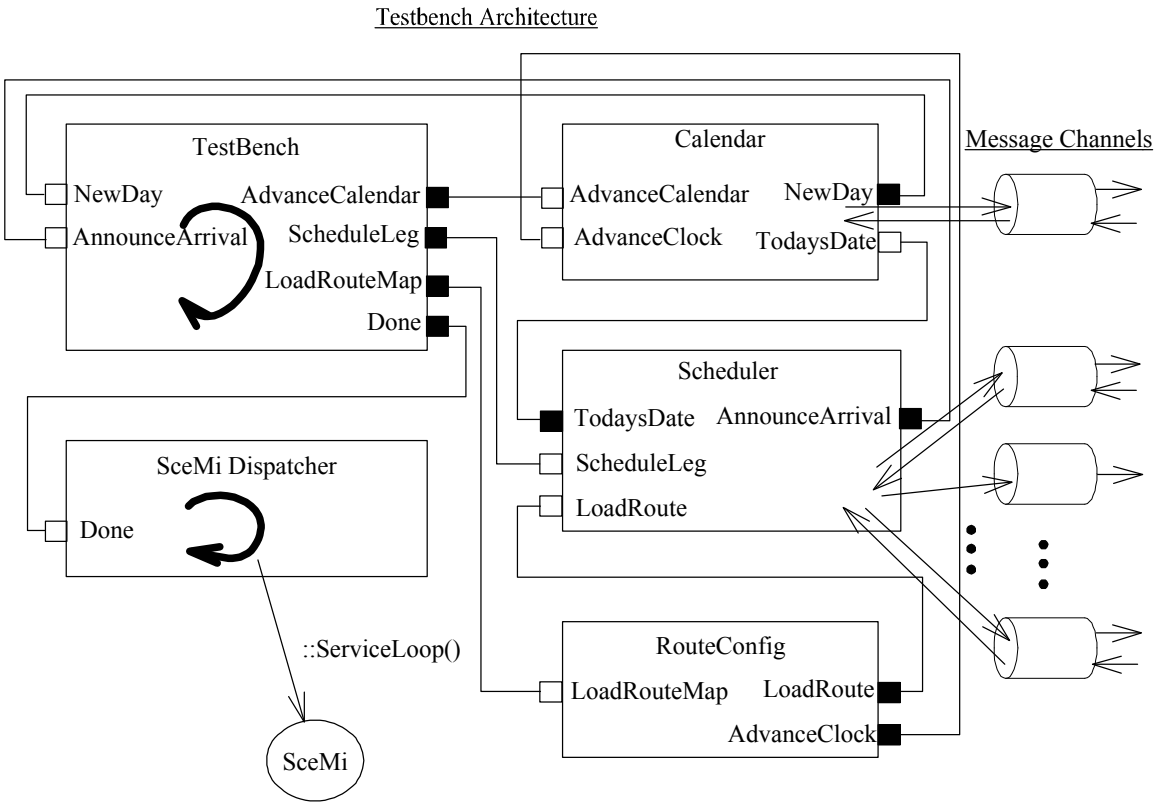



Figure A.6 Interconnect of SystemC models

The source code for the System model is shown here.

```
class System: public sc_module {
  public:
    sc_link_mp<unsigned>                          newDay;
    sc_link_mp<const Routed::ArrivalRecord *> announceArrival;
    sc_link_mp<unsigned>                          advanceCalendar;
    sc_link_mp<const Routed::Itinerary *>     scheduleLeg;
    sc_link_mp<>                                  loadRouteMap;
    sc_link_mp<>                                  done;

    sc_link_mp<>                                  advanceClock;
    sc_link_mp<Routed::Date>                  todaysDate;

    sc_link_mp<const Routed::Route *>         loadRoute;

    //-------------------------------------------------------
    // Module declarations
    Testbench   *testbench;
    Calendar    *calendar;
    Scheduler   *scheduler;
    RouteConfig *routeConfig;

    SceMiDispatcher *dispatcher;

    System( sc_module_name name,  SceMi *sceMi ) : sc_module( name ) {
        testbench = new Testbench( "testbench" );
            testbench->NewDay(          newDay );
            testbench->AnnounceArrival( announceArrival );
            testbench->AdvanceCalendar( advanceCalendar );
            testbench->ScheduleLeg(     scheduleLeg );
            testbench->LoadRouteMap(    loadRouteMap );
            testbench->Done(            done );

        calendar = new Calendar( "calendar", sceMi );
            calendar->AdvanceCalendar( advanceCalendar );
            calendar->AdvanceClock(    advanceClock );
            calendar->NewDay(          newDay );
            calendar->TodaysDate(      todaysDate );

        scheduler = new Scheduler( "scheduler", sceMi );
            scheduler->TodaysDate(      todaysDate );
            scheduler->ScheduleLeg(     scheduleLeg );
            scheduler->LoadRoute(       loadRoute );
            scheduler->AnnounceArrival( announceArrival );

        routeConfig = new RouteConfig( "routeConfig" );
            routeConfig->LoadRouteMap( loadRouteMap );
            routeConfig->LoadRoute(    loadRoute );
            routeConfig->AdvanceClock( advanceClock );

        dispatcher = new SceMiDispatcher( "dispatcher", sceMi );
            dispatcher->Done( done );
    }
};
```

SystemC interconnect channels are declared as `sc_link_mp<>` data types. These can be thought of as abstract signals that interconnect abstract ports. The parameterized data type associated with each `sc_link_mp<>` denotes the data type of the message the channel is capable of transferring from an output abstract port to an input abstract port.

Notice the `todaysDate` channel is declared with a “by value” data type (i.e., `Routed::Date`), whereas some of the other channels, such as the `announceArrival`, are declared as “by reference” data types (i.e., const

`Routed::ArrivalRecord *`). The former is less efficient, but safer, because the message is passed by value and, therefore, there is no danger of the receiver corrupting the sender's data, or worse, having the sender's data go out-of-scope, leaving the receiver with a possibly dangling reference. However, passing messages by reference is more efficient, but potentially problematic. Declaring them as const pointers helps alleviate some, but not all, of the safety problems.

Module pointers are declared inside the `SC_MODULE(System)` object and constructed in its SystemC constructor ( `SC_CTOR(System)` ). After each child module is constructed, its abstract ports are mapped to one of the declared interconnect channels.

NOTE—SystemC channels, while conceptually the same, are distinctly different from SCE-MI message channels. Both types of channels pass messages, but SystemC channels are designed strictly to pass messages of arbitrary C++ data types between SystemC modules. An entire simulation can be built of just software models communicating with each other. See [B2] for more details about SystemC interconnect channels.

SCE-MI message channels have a completely different interface and are optimized for implementing abstraction bridges between a software subsystem and a hardware subsystem. In the use model presented in this example (see Figure A.6), their interfaces are encapsulated by SystemC models.

The thick round arrows in Figure A.6 represent the SystemC autonomous threads contained in the Testbench and `SceMiDispatcher` modules. These two threads are the only autonomous threads in the system. All the other code is executed inside slave threads.

## The sc_main() routine and error handler

The following listing shows the `sc_main()` routine which is the top-level entry point to the program. The `sc_main()` is required when linking to a SystemC kernel facility, but it is very much like a conventional `main()` C or C++ entry point and has the same program argument passing semantics.

```
int sc_main( int argc, char *argv[] ){
    //-------------------------------------------------------
    // Instantiate SceMi

    SceMi::RegisterErrorHandler( errorHandler, NULL );
    SceMi *sceMi = NULL;

    try {
        int sceMiVersion = SceMi::Version( SCEMI_VERSION_STRING );
        SceMiParameters parameters( "mct" );
        sceMi = SceMi::Init( sceMiVersion, &parameters );

        //-------------------------------------------------------
        // Instantiate the system here.  Autonomous threads nested
        // inside the DispatcherDriver and the Testbench will advance
        // untimed activity.  Such threads are senstitive to UTick defined
        // at the top of this file.
        // -- johnS  8-29-00

        System system( "system", sceMi );

        //-----------------------------------------------
        // Kick off SystemC kernel ...
        cerr << "Let 'er rip !" << endl;
        sc_start(-1);
    }

    catch( string message ) {
        cerr << message << endl;
        cerr << "Fatal Error: Program aborting." << endl;
        if( sceMi ) SceMi::Shutdown( sceMi );
        return -1;
    }
    catch(...) {
        cerr << "Error: Unclassified exception." << endl;
        cerr << "Fatal Error: Program aborting." << endl;
        if( sceMi ) SceMi::Shutdown( sceMi );
        return -1;
    }
    return 0;
}

static void errorHandler( void */*context*/, SceMiEC *ec  ) {
    char buf[32];

    sprintf( buf, "%d", (int)ec->Type );
    string messageText( "SCE-MI Error[" );
    messageText += buf;
    messageText += "]: Function: ";
    messageText += ec->Culprit;
    messageText += "\n";
    messageText += ec->Message;
    throw messageText;
}
```

The first routine defined is the `errorHandler()`. This is the master error-handling function that is registered with the SCE-MI. Whenever an error occurs, this function is called to format the message before throwing a C++ exception. The exceptions are caught in the `catch { ... }` blocks at the end of the `sc_main()` routine, where they are displayed before exiting the program.

Once the error handler is registered, the SCE-MI is initialized by calling `SceMi::Init()`. This method returns a pointer to a SceMi object that manages the whole SCE-MI software side infrastructure.

Next, the System model described in A.2.4.1 is constructed. The constructor ( `SC_CTOR(System)` ) causes all of its child software models to get constructed by calling, in turn, their `SC_CTOR()` constructors.

Once the whole system is statically constructed, models that interface with SCE-MI are given the master SceMi object pointer so they can access its methods, by calling special `::Bind()` accessor methods on those models.

Finally, the SystemC main kernel loop is initialized by calling the `sc_start()` function. The -1 parameter tells it to go indefinitely until the program decides to end (as explained in A.2.4.3).

## The SceMiDispatcher module: interfacing with the SCE-MI service loop

The `SceMiDispatcher` module contains an autonomous thread that yields to the SCE-MI infrastructure so it can service its message port proxies by making repeated calls to the `SceMi::ServiceLoop()` method (see 5.4.3.7). By placing this logic on its own dedicated thread, other models in the system do not have to worry about yielding to the SCE-MI.

The source code for the `SceMiDispatcher` is shown here.

```
class SceMiDispatcher: public sc_module {

  public:
    sc_slave<> Done;

  private:
    SC_HAS_PROCESS(SceMiDispatcher);

    //-------------------------------------------------------
    // Thread declarations

    void dispatchThread(); // Autonomous SCEMI dispatcher thread
    void doneThread();

    //-------------------------------------------------------
    // Context declarations

    SceMi *dSceMi;
    static int dInterruptReceived;

    //-------------------------------------------------------
    // Context declarations
    static void signalHandler( int ){
        cout << "Interrupt received !  Terminating SCEMI" << endl;
        dInterruptReceived = 1;
    }

  public:
    SceMiDispatcher( sc_module_name name,  SceMi *sceMi )
      : sc_module( name ), dSceMi(sceMi)
    {
        //--------------------------------------
        // Thread bindings
        SC_THREAD( dispatchThread );
        sensitive << UTick;
            // Sensitize to global "Untimed Tick" clock to provide for
            // atomic advance of this along with other autonomous threads
            // in the system.  UTick is declared at the top of System.cpp.
            // -- johnS  8-3-00

        // Clients of this dispatcher will be responsible for binding
        // to their respective message port proxies in their respective
        // constructors.

        SC_SLAVE( doneThread, Done );

        signal( SIGINT, signalHandler );
    }
};

int SceMiDispatcher::dInterruptReceived = 0;

void SceMiDispatcher::dispatchThread() {
    // This is all the dispatcher does !!  Deceptively simple, eh ?
    // It just calls the SCEMI dispatcher poll function and returns.
    for(;;){
        wait();
        dSceMi->ServiceLoop();
        if( dInterruptReceived ){
            SceMi::Shutdown( dSceMi );
            exit(1);
        }
    }
```

```
}

void SceMiDispatcher::doneThread() {
     SceMi::Shutdown( dSceMi );
     exit(0);
}
```

Between each call to the service loop, the autonomous thread yields to other threads in the system by calling the `wait()` function. Actually, the only other autonomous thread in the Routed system is the one in the Testbench model. Both of these threads are represented by the thick round arrows in Figure A.6.

The other job of the `SceMiDispatcher` is to shut down the system when it detects a notification on its `Done` port that the simulation is complete. The Done `inslave` port is bound to the slave thread, `::doneThread()`, on construction. The `Done` port is driven from its associated `outmaster` port on the Testbench module, so it is the Testbench that ultimately decides when the simulation is complete (see A.2.4.5).

## Application-specific data types for the Routed system

The following data types are defined in the `Routed.hxx` header file. They are referenced throughout the subsequent discussion. They are data types which are specific to this application.

```
class Routed {
    public:
    typedef enum Parameters {
        NumPassengers = 4,
        NumLocations  = 12,
        MessageSize   = 15
    };
    typedef enum PassengerIDs {
        Nobody,
        BugsBunny,
        DaffyDuck,
        ElmerFudd,
        SylvesterTheCat
    };
    typedef enum LocationIDs {
    //  Location           Origin  Destination Hub
    //  --------           ------  ----------- ---
        Unspecified,
        Anchorage,    //  1: X       X
        Chicago,      //  2:                    X
        Cupertino,    //  3: X       X
        Dallas,       //  4:                    X
        Maui,         //  5:         X
        Newark,       //  6:                    X
        Noida,        //  7: X
        SanFran,      //  8:                    X
        SealBeach,    //  9: X       X
        Seattle,      // 10:                    X
        UK,           // 11: X       X
        Waltham       // 12: X
    };
    typedef struct Itinerary {
        unsigned     DateOfTravel;
        unsigned     TimeOfDeparture;
        PassengerIDs PassengerID;
        LocationIDs  OriginID;
        LocationIDs  DestinationID;
    };
    typedef struct ArrivalRecord {
        PassengerIDs PassengerID;
        unsigned     DateOfArrival;
        unsigned     TimeOfArrival;
```

```
            unsigned     LayoverCount;
            LocationIDs  OriginID;
            LocationIDs  LayoverIDs[4];
            LocationIDs  DestinationID;
        };
        typedef struct Route {
            LocationIDs  RouterID;
            LocationIDs  DestinationID;
            unsigned     PortID;
        };
        typedef struct Date {
            SceMiU64  CycleStamp;
            unsigned Day;
        };
    };
```

## The Testbench model: main control loop

The Testbench model contains a SystemC autonomous thread which serves as the main driver for the Routed design. It looks at the four passenger itineraries and schedule the legs in those itineraries on the appropriate dates and at the appropriate departure times by interacting with the Scheduler model.

The condensed source code for the passenger itinerary declarations for the Testbench model is shown here.

```
const Routed::Itinerary Routed::BugsesTrip[] = {
      /*
      On day,     at,                 departs from,       enroute to,           */
      {        2,      8, BugsBunny,          Anchorage,      Cupertino },
      {        3,      5, BugsBunny,          Cupertino,      UK        },
      {        8,      4, BugsBunny,          UK,             SealBeach },
      {       20,     10, BugsBunny,          SealBeach,      Maui      },
      {        0,      0, BugsBunny,          Unspecified,    Unspecified } };

      const Routed::Itinerary Routed::DaffysTrip[] = {
      /*
      On day,     at,                 departs from,       enroute to,           */
      {        1,      8, DaffyDuck,          Waltham,        Cupertino },
      {        4,      2, DaffyDuck,          Cupertino,      SealBeach },
      {        5,     11, DaffyDuck,          SealBeach,      Anchorage },
      {       10,      3, DaffyDuck,          Anchorage,      UK        },
      {       15,      4, DaffyDuck,          UK,             Cupertino },
      {       22,      7, DaffyDuck,          Cupertino,      Maui      },
      {        0,      0, DaffyDuck,          Unspecified,    Unspecified } };

      const Routed::Itinerary Routed::ElmersTrip[] = {
      /*
      On day,     at,                 departs from,       enroute to,           */
      {        3,      5, ElmerFudd,          SealBeach,      Anchorage },
      {        4,      2, ElmerFudd,          Anchorage,      SealBeach },
      {        8,     15, ElmerFudd,          SealBeach,      Cupertino },
      {       23,      3, ElmerFudd,          Cupertino,      Maui      },
      {        0,      0, ElmerFudd,          Unspecified,    Unspecified } };

      const Routed::Itinerary Routed::SylvestersTrip[] = {
      /*
      On day,     at,                 departs from,       enroute to,           */
      {        1,      1, SylvesterTheCat,    Noida,          SealBeach },
      {        4,      2, SylvesterTheCat,    SealBeach,      Cupertino },
      {        5,     11, SylvesterTheCat,    Cupertino,      UK },
      {       10,      4, SylvesterTheCat,    UK,             SealBeach },
      {       15,      9, SylvesterTheCat,    SealBeach,      Anchorage },
      {       20,      7, SylvesterTheCat,    Anchorage,      Maui      },
      {        0,      0, SylvesterTheCat,    Unspecified,    Unspecified } };
```

```
static const char *passengerNames[] = {
    "Nobody         ",
    "BugsBunny      ",
    "DaffyDuck      ",
    "ElmerFudd      ",
    "SylvesterTheCat" };

static const char *locationNames[] = {
    "Unspecified",
    "Anchorage",
    "Chicago  ",
    "Cupertino",
    "Dallas   ",
    "Maui     ",
    "Newark   ",
    "Noida    ",
    "SanFran  ",
    "SealBeach",
    "Seattle  ",
    "UK       ",
    "Waltham  " };
```

There are four passengers whose itineraries are given as lists of `Routed::Itinerary` records. Each record represents a leg of that passenger's journey consisting of a date of departure, time of departure, passenger, origin, and destination. The `passengerNames` and `locationNames` are strings use for printing messages.

The SystemC module definition (`class sc_module`) for the Testbench model with its standard constructor is shown here.

```
class Testbench: public sc_module {

  public:
    //----------------------------------------------------------
    // Abstract port declarations
    sc_master<>            LoadRouteMap;
    sc_master<>            Done;
    sc_outmaster<unsigned> AdvanceCalendar;
    sc_inslave<unsigned>   NewDay;

    sc_outmaster<const Routed::Itinerary *>   ScheduleLeg;
    sc_inslave<const Routed::ArrivalRecord *> AnnounceArrival;

  private:
    SC_HAS_PROCESS(Testbench);

    //----------------------------------------------------------
    // Context declarations
    unsigned dNumMauiArrivals;
    unsigned dDayNum;
    const Routed::Itinerary *dItineraries[Routed::NumPassengers];

    //----------------------------------------------------------
    // Thread declarations
    void driverThread();  // Autonomous "master" thread.
    void newDayThread() { dDayNum = NewDay; }
    void announceArrivalThread();

    //----------------------------------------------------------
    // Helper declarations

  public:
    Testbench( sc_module_name name )
        : sc_module(name), dNumMauiArrivals(0), dDayNum(0)
```

```
    {
        //--------------------------------------
        // Thread bindings

        // This autonomous thread forms the main body of the TIP driver.
        SC_THREAD( driverThread );
        sensitive << UTick;

        SC_SLAVE( newDayThread, NewDay );
        SC_SLAVE( announceArrivalThread, AnnounceArrival );

        // Initialize itinerary pointers.
        dItineraries[0] = Routed::BugsesTrip;
        dItineraries[1] = Routed::DaffysTrip;
        dItineraries[2] = Routed::ElmersTrip;
        dItineraries[3] = Routed::SylvestersTrip;
    }
};
```

## Main driver loop

The autonomous thread for the main driver loop is shown here.

```
void Testbench::driverThread(){
    LoadRouteMap(); // Signal RouteConfig model to begin
                    // configuration RouteMap.
    unsigned dayNum = dDayNum;
    AdvanceCalendar = 1; // Advance to day 1.

    for(;;){
        wait(); // Wait for day to advance (i.e., 'NewDay' arrives.)

        if( dayNum != dDayNum ){
            unsigned date, minDate = 1000;

            // Check itineraries to see if any passengers are
            // traveling today. If so, advance calendar to tomorrow
            // in case next leg of itinerary is tomorrow.
            for( int i=0; i<Routed::NumPassengers; i++ ){
                if( (date=dItineraries[i]->DateOfTravel) ){
                    if( date == dDayNum ){
                        cout << "On day " << setw(2) << dDayNum << " at "
                             << setw(2) << dItineraries[i]->TimeOfDeparture
                             << ":00 hrs, "
                             << passengerNames[dItineraries[i]->PassengerID]
                             << " departs "
                             << locationNames[dItineraries[i]->OriginID]
                             << " enroute to "
                             << locationNames[dItineraries[i]->DestinationID]
                             << endl;

                        ScheduleLeg = dItineraries[i]++;
                        minDate = dDayNum+1;
                    }
                    else if( date < minDate )
                        minDate = date;
                }
            }
            dayNum = dDayNum;
            AdvanceCalendar = minDate - dDayNum;
        }
    }
}
```

Before entering its main loop, the autonomous `::driverThread()` does two things. First, it triggers the `RouteConfig` model (by signaling the `LoadRouteMap outmaster` port) to teach all the routes to the `RouteTables` of all the Hubs in the `RouteMap`. Each taught route that is injected to the hardware is staggered by one clock, which are done when the `RouteConfig` model signals the `AdvanceClock` port on the Calendar model. Passenger travel in the `RouteMap` is not possible until all the Hubs have been properly programmed with their routes.

Once all the routes have been taught to the `RouteMap`, the Calendar is advanced to day one. This causes the Calendar model to announce the arrival of day one via the `NewDay inslave` port. Once the day change has been detected, the `::driverThread()` then enters into a loop where it schedules any travel on the itineraries scheduled for the current day. If no travel is scheduled, it advances the Calendar to the first day on which travel is scheduled to occur. Legs of each itinerary are scheduled by sending the Itinerary record over the `ScheduleLeg outmaster` port to the `Scheduler` model, which encodes it into a token and sends it to the hardware.

This operation continues for each leg of each itinerary until all passengers have traveled all legs of their trip and have finally arrived at the Maui Destination. This serves as the termination condition, which is conveyed to the `SceMiDispatcher` model by signaling the Done `outmaster` port (see A.2.4.5.2). Upon receiving this notification, the `SceMiDispatcher` model gracefully shuts down the SCE-MI and exits the program with a normal exit status.

## Announcing arrivals

The Testbench model also announces arrivals of passengers at their destinations as they occur. The `::announceArrivalThread()` slave thread detects an arrival by receiving an `ArrivalRecord` on its `AnnounceArrival inslave` port (which was sent from the message output port proxy-receive callback in the Scheduler). It prints out the arrival information to the console. The source code is shown here.

```
void Testbench::announceArrivalThread(){
    const Routed::ArrivalRecord *arrivalRecord = AnnounceArrival;

    cout << "On day " << setw(2) << arrivalRecord->DateOfArrival
         << " at " << setw(2) << arrivalRecord->TimeOfArrival << ":00 hrs,\n"
         << "    " << passengerNames[arrivalRecord->PassengerID]
         << " arrives in " << locationNames[arrivalRecord->DestinationID]
         << "   from " << locationNames[arrivalRecord->OriginID]
         << " after layovers in,";

    for( unsigned i=0; i<arrivalRecord->LayoverCount; i++ )
        cout << "\n        "
             << locationNames[arrivalRecord->LayoverIDs[i]];
    cout << endl;
    // Check for termination condition.
    if(     arrivalRecord->DestinationID == Routed::Maui &&
            ++dNumMauiArrivals == Routed::NumPassengers ){
        cout << "Everyone has arrived in Maui. We're done. Let's party !"
             << endl;
        Done(); // Signal the dispatcher that the simulation has ended.
    }
}
```

## The Scheduler module: interfacing with message port proxies

The SystemC module definition and constructor for the Scheduler model is shown here.

```
class Scheduler: public sc_module {

  public:
    //-------------------------------------------------------
    // Abstract port declarations
    sc_inmaster<Routed::Date>                   TodaysDate;
    sc_inslave<const Routed::Itinerary *>       ScheduleLeg;
    sc_inslave<const Routed::Route *>           LoadRoute;
    sc_outmaster<const Routed::ArrivalRecord *> AnnounceArrival;

  private:
    SC_HAS_PROCESS(Scheduler);

    //-------------------------------------------------------
    // Context declarations
    SceMiMessageData dSendData;
    SceMiMessageInPortProxy *dOriginAnchorage;
    SceMiMessageInPortProxy *dOriginCupertino;
    SceMiMessageInPortProxy *dOriginNoida;
    SceMiMessageInPortProxy *dOriginSealBeach;
    SceMiMessageInPortProxy *dOriginUK;
    SceMiMessageInPortProxy *dOriginWaltham;

    SceMiMessageOutPortProxy *dDestinationAnchorage;
    SceMiMessageOutPortProxy *dDestinationCupertino;
    SceMiMessageOutPortProxy *dDestinationMaui;
    SceMiMessageOutPortProxy *dDestinationSealBeach;
    SceMiMessageOutPortProxy *dDestinationUK;

    Routed::ArrivalRecord dArrivalRecord;

    //-------------------------------------------------------
    // Thread declarations
    void scheduleLegThread();
    void loadRouteThread();

    //-------------------------------------------------------
    // Helper declarations
    static void replyCallback( void *context, const SceMiMessageData *data );
    void announceArrival( SceMiU64 cycleStamp, SceMiU32 arrivalToken );

  public:
    Scheduler( sc_module_name name,  SceMi *sceMi )
      : sc_module( name ),
        dSendData(Routed::MessageSize),
        dOriginAnchorage(NULL),
        dOriginCupertino(NULL),
        dOriginNoida(NULL),
        dOriginSealBeach(NULL),
        dOriginUK(NULL),
        dOriginWaltham(NULL),
        dDestinationAnchorage(NULL),
        dDestinationCupertino(NULL),
        dDestinationMaui(NULL),
        dDestinationSealBeach(NULL),
        dDestinationUK(NULL)
    {
        SC_SLAVE( scheduleLegThread,     ScheduleLeg );
        SC_SLAVE( loadRouteThread,       LoadRoute );

        // Establish message input portals.
//      SceMiMessageInPortBinding inBinding = { NULL, NULL, NULL };
```

```
            dOriginAnchorage = sceMi->BindMessageInPort(
                "Bridge.anchorage.origin", "sceMiMessageInPort", NULL );
            dOriginCupertino = sceMi->BindMessageInPort(
                "Bridge.cupertino.origin", "sceMiMessageInPort", NULL );
            dOriginNoida     = sceMi->BindMessageInPort(
                "Bridge.noida",            "sceMiMessageInPort", NULL );
            dOriginSealBeach = sceMi->BindMessageInPort(
                "Bridge.sealBeach.origin", "sceMiMessageInPort", NULL );
            dOriginUK        = sceMi->BindMessageInPort(
                "Bridge.UK.origin",        "sceMiMessageInPort", NULL );
            dOriginWaltham   = sceMi->BindMessageInPort(
                "Bridge.waltham",          "sceMiMessageInPort", NULL );

            // Establish message output portals.
            SceMiMessageOutPortBinding outBinding = { this, replyCallback, NULL };
            dDestinationAnchorage = sceMi->BindMessageOutPort(
                "Bridge.anchorage.destination", "sceMiMessageOutPort",
                &outBinding );
            dDestinationCupertino = sceMi->BindMessageOutPort(
                "Bridge.cupertino.destination", "sceMiMessageOutPort",
                &outBinding );
            dDestinationMaui      = sceMi->BindMessageOutPort(
                "Bridge.maui",                  "sceMiMessageOutPort",
                &outBinding );
            dDestinationSealBeach = sceMi->BindMessageOutPort(
                "Bridge.sealBeach.destination", "sceMiMessageOutPort",
                &outBinding );
            dDestinationUK        = sceMi->BindMessageOutPort(
                "Bridge.UK.destination",        "sceMiMessageOutPort",
                &outBinding );
            }
    };
```

There are two slave threads defined in this model: the `::scheduleLegThread()` and the `::loadRouteThread()`. The `::loadRouteThread()` is responsible for sending `TeachRoute` tokens into the `RouteMap` mesh via the Waltham Origin transactor when the `RouteMap` is first being configured at the beginning of the simulation. This thread is activated each time the `RouteConfig` module wants to teach a new route during its `LoadRouteMap` operation.

The `Scheduler::Bind()` method is called prior to simulation from the `sc_main()` routine (see A.2.4.2). Here is where the SCE-MI message input and output port proxies leading to each of the Origin and Destination transactors are bound. Notice for each of the output port proxies, the output receive callback, `replyCallback()`, is specified in the binding structure. See 5.4.3.6 for more information about message output port binding.

## ::scheduleLegThread()

The `::scheduleLegThread()` is activated when the Scheduler receives `Routed::Itinerary` messages on its `ScheduleLeg inslave` port from the Testbench model. It sends those legs encoded as departure tokens across the message input channels to their designated Origin transactors. The Scheduler has pointers to each of the message input port proxies that are connected to Origin transactors. Each departure token is encoded with the passenger ID and destination ID from the `Routed::Itinerary` record. The source code for the `::scheduleLegThread()` is shown here.

```
void Scheduler::scheduleLegThread(){
    const Routed::Itinerary *leg = ScheduleLeg;

    // Form a 'Passenger Departure' token based on the contents of
    // the given 'Itinerary' record.
    SceMiU32 passengerDepartureToken =
         leg->PassengerID            |
        (leg->DestinationID   <<  4) |
        (leg->OriginID        << 12) |
        (leg->TimeOfDeparture << 16);

    dSendData.Set( 0, passengerDepartureToken );

    switch( leg->OriginID ){
        case Routed::Anchorage: dOriginAnchorage->Send( dSendData );
break;
        case Routed::Cupertino: dOriginCupertino->Send( dSendData );
break;
        case Routed::Noida:     dOriginNoida    ->Send( dSendData );
break;
        case Routed::SealBeach: dOriginSealBeach->Send( dSendData );
break;
        case Routed::UK:        dOriginUK       ->Send( dSendData );
break;
        case Routed::Waltham:   dOriginWaltham  ->Send( dSendData );
break;
        default:
            assert(0);
    }
}
```

## Processing arrivals

The Scheduler is also responsible for processing of arrivals. Once the Calendar is advanced, arrivals can occur at any time over the course of 24 hours (i.e., 24 clocks). Each arrival token is sent by a Destination transactor over a message output port to the Scheduler. The SCE-MI infrastructure dispatches the arriving messages to the `replyCallback()` function registered in the `::Bind()` method. The `replyCallback()` function, in turn, passes the message to the private `::announceArrival()` method (see A.2.4.6.3). The code for the `replyCallback()` function is shown here.

```
void Scheduler::replyCallback( void *context, const SceMiMessageData *data ){
    ((Scheduler *)context)->announceArrival( data->CycleStamp(),
      data->Get(0) ); }
```

## ::announceArrival()

The `::announceArrival()` method processes the arrival token. It converts the encoded arrival token to the `Routed::ArrivalRecord` data type, stamps it with `TodaysDate` (an output from the Calendar), and sends it out through the `AnnounceArrival outmaster` port to the Testbench model, which displays the arrival information to the console as shown here.

```
void Scheduler::announceArrival( SceMiU64 cycleStamp,
                                 SceMiU32 arrivalToken ){
    Routed::Date todaysDate = TodaysDate;
    // Read today's date from Calendar

    dArrivalRecord.DateOfArrival = todaysDate.Day;
    dArrivalRecord.TimeOfArrival = cycleStamp - todaysDate.CycleStamp;
    dArrivalRecord.PassengerID   = (Routed::PassengerIDs)
                                   (  arrivalToken        & 0xf );
    dArrivalRecord.DestinationID = (Routed::LocationIDs)
                                   ( (arrivalToken >>  4) & 0xf );
    dArrivalRecord.OriginID      = (Routed::LocationIDs)
                                   ( (arrivalToken >> 12) & 0xf );
    dArrivalRecord.LayoverCount  =  (arrivalToken >>  8) & 0xf  ;
    assert( dArrivalRecord.LayoverCount < 5 );
    arrivalToken >>= 16;
    for( unsigned i=0; i<dArrivalRecord.LayoverCount; i++ ){
        dArrivalRecord.LayoverIDs[i] = (Routed::LocationIDs)
                                       ( arrivalToken & 0xf );
        arrivalToken >>= 4;
    }
    AnnounceArrival = &dArrivalRecord;
    // Arrival record is passed by reference.
}
```

## The Calendar module: interfacing with the clock advancer

The Calendar model is responsible for advancing time on the `RouteMap` one or more days at a time. Once a set of scheduled departures has been programmed in each Origin transactor which has departures scheduled for a particular day, the Calendar allows the DUT to advance by 24 clocks (i.e., 24 hours) or some multiple of 24 clocks if the next scheduled departure occurs more than one day from now. The `Calendar` advances time by sending a message to the `ClockAdvancer` transactor in the hardware which has direct control of the DUT clock via the `ClockControl` macro. The source code for the `Calendar` module is very similar in structure to that for the Scheduler; therefore, most of it is not shown here.

The `Calendar` model has two slave threads that respond to requests to advance time. The `::advanceCalendarThread()` responds to requests on the `AdvanceCalendar` port to advance a given number of days.

## ::advanceClockThread()

The `::advanceClockThread()` responds to requests to advance one clock at a time which occurs during `RouteMap` configuration to stagger the injection of each `TeachRoute` token by one clock. This method is shown here.

```
void Calendar::advanceClockThread(){
    dSendData.Set( 0, 1 );
    // Tell ClockAdvancer to advance by 1 clock.
    dInputPort->Send( dSendData );
    // Send message out on port proxy.

    // Pend until the cycle stamp gets updated by the
    // output port proxy reply callback.
    SceMiU64 currentCycleStamp = dCycleStamp;
    while( dCycleStamp == currentCycleStamp )
        wait();
}
```

Notice this method enters a loop that calls `wait()` to yield to the SystemC kernel. This guarantees the clock has completed its advance before returning. By yielding to the SystemC kernel while it is waiting for this condition, the autonomous `SceMiDispatcher` thread (see A.2.4.3) is naturally given a chance to service the message output

ports. This is necessary to reach the condition the::advanceClockThread() is waiting for, namely, for the Calendar::dCycleStamp data member to change value.

### replyCallback()

The ::dCycleStamp changes value when the ClockAdvancer (on the hardware side) indicates on its output port it has completed its one clock time advance which, in turn, causes the Calendar::replyCallback() function to get called from the SceMi::ServiceLoop(). The replyCallback() function is shown here.

```
void Calendar::replyCallback( void *context,
                              const SceMiMessageData *data ){
    ((Calendar *)context)->dCycleStamp = data->CycleStamp(); }
```

The cycle stamp is updated directly from the ::CycleStamp() method on the SceMiMessageData object. This reflects a count of elapsed controlled clock counts that had occurred from the beginning of the simulation to the time this message was sent from the hardware side. This is a convenient way for software to keep track of elapsed clock time in the hardware. Once the ::dCycleStamp is updated, the wait() loop in the ::advanceClockThread() (see A.2.4.7.1), is released and the function can return.

Keep in mind the ::advanceClockThread() and replyCallback() functions are being called under two different autonomous threads which each frequently yield to each other. The former is called from the autonomous Testbench::driverThread(); the latter is called from the SceMi::ServiceLoop() function which is called from underneath the autonomous SceMiDispatcher::dispatchThread().

This illustrates the clean interaction between a general multi-threaded application software environment and the SCE-MI service loop.

## A.3 Complete example Using SystemC TB Modeling Environment

The following diagram and source code depicts a complete example of a small system modeled using the SCE-MI 2 DPI. The testbench is written in SystemC and consists of a Producer model that produces transactions of a custom user defined class MyType. Those transactions are fed via a C proxy model called PipelineIngressProxy to a transactor on the HDL side called PipelineIngressTransactor. DPI input pipes are used as the conduit between the proxy model and the transactor. The transactor formats the transactions as tokens that are fed to a simple DUT called Pipeline.

After propagating through the DUT the tokens are fed back to a transactor called PipelineEgressTransactor. This transactor, in turn forms transactions to be sent up to the Consumer module in the TB via the PipelineEgressProxy C proxy model (using DPI output pipes as the conduit).

This example also demonstrates compatibility with multi-threaded TB environments such as SystemC. In this case there are 4 threads in the TB which are depicted with the circular arrows. Details of the various models are described in the comments in the accompanying source code.

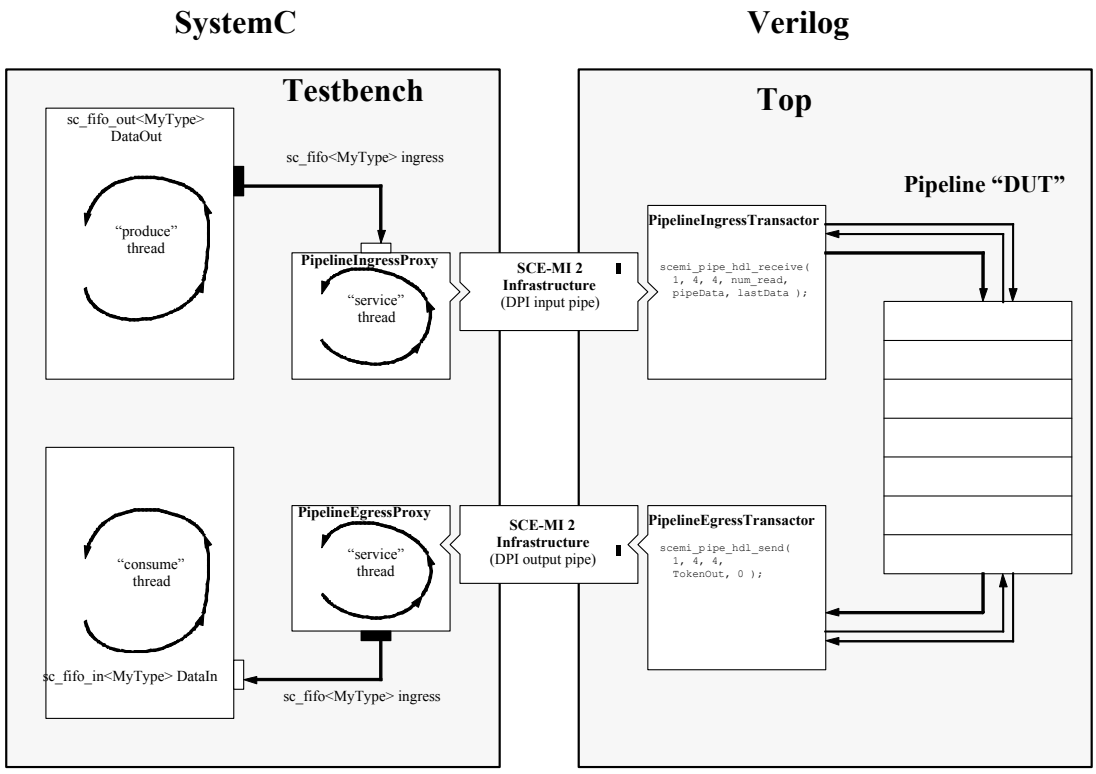



Figure A.7 DPI Interfacing between a SystemC TB and a Verilog DUT

# Testbench

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <vector>

using namespace std;

#include "systemc.h"
#include "mytype.h"
#include "svdpi.h"
#include "scemi.hxx"
#include "scemi_pipes.h"
#include "testbench.h"

static void errorHandler( void */*context*/, SceMiEC *ec ) {
    char buf[128];
    sprintf( buf, "%d", (int)ec->Type );
    string messageText( "SCE-MI Error[" );
    messageText += buf; messageText += "]: Function: ";
    messageText += ec->Culprit; messageText += "\n";
    messageText += ec->Message;
    cerr << messageText;
    throw messageText;
}
//------------------------------------------------------------
// sc_main()
```

```
//
// This is the SystemC "main()" from which all else happens.
//-------------------------------------------------------
int sc_main( int /*argc*/, char */*argv*/[] ){

    // Register error handler with the SCE-MI infrastructure.
    SceMi::RegisterErrorHandler( errorHandler, NULL );
    SceMi *sceMi = NULL;
    try {
        int sceMiVersion = SceMi::Version( SCEMI_VERSION_STRING );
        SceMiParameters parameters( "myconfig" );
        sceMi = SceMi::Init( sceMiVersion, &parameters );

        // Instantiate the top level testbench.
        Testbench testbench( "testbench" );

        // Kick off SystemC kernel ...
        sc_start(-1);
        SceMi::Shutdown( sceMi );
    }


//____________________
// class Testbench \_____________________________________________________________
//
// This is the top level SystemC testbench that instantiates the Producer
// and Consumer modules which drive the Pipeline DUT on the Verilog side.
//------------------------------------------------------------------------------

class Testbench : public sc_module {
    private:
        sc_fifo<MyType> ingressChannel0;
        sc_fifo<MyType> egressChannel0;
        sc_buffer<bool> done0;

        Producer      producer0;
        Consumer      consumer0;
        PipelineIngressProxy pipelineIngressProxy0;
        PipelineEgressProxy  pipelineEgressProxy0;

        vector<MyType> dValues0;

        SC_HAS_PROCESS(Testbench);

        void terminateThread(){
            int doneCount = 0;
            while( doneCount < 1 ){
                wait();
                if( done0.event() ){
                    printf( "Pipeline0 done.\n" );
                    doneCount++;
                }
            }
            printf( "Termination condition detected, calling sc_stop() ...\n" );
            sc_stop();
        }
    public:
        Testbench( sc_module_name name )
          : sc_module(name),
            producer0( "producer0", dValues0 ),
            consumer0( "consumer0" ),
            pipelineIngressProxy0( "pipelineIngressProxy0",
               "Top.pipelineIngressTransactor0" ),
```

```
            pipelineEgressProxy0( "pipelineEgressProxy0",
                "Top.pipelineEgressTransactor0" )
        {   SC_THREAD( terminateThread );
                sensitive << done0;

            // Map all data channel ports.
            producer0.DataOut( ingressChannel0 );
            consumer0.DataIn( egressChannel0 );

            consumer0.Done( done0 );

            pipelineIngressProxy0.Ingress( ingressChannel0 );
            pipelineEgressProxy0.Egress( egressChannel0 );

            // Intialize transaction vectors for Producer modules.
            dValues0.push_back( MyType( 5, 12.0,  3L) );
            dValues0.push_back( MyType(11, 21.0,  1L) );
            dValues0.push_back( MyType( 8,  0.0,  0L) );
};
```

## C Proxies

```
//_________________________________
// class PipelineIngressProxy  \_____________________________________________
//
// The PipelineIngressProxy module accepts data received over a data channel
// (in this case an sc_fifo) from producer sends it to the Pipeline DUT on the
// Verilog side passing transactions over a DPI input pipe to the
// PipelineIngressTransactor.
//-----------------------------------------------------------------------------
class PipelineIngressProxy : public sc_module {
    public:
        sc_fifo_in<MyType> Ingress;
        sc_event dResetComplete;
    private:
        svScope dHdlContext;
        SC_HAS_PROCESS(PipelineIngressProxy);

        void serviceThread(){
            MyType localIngress;

            // Wait for confirmation that reset has occurred ...
            wait( dResetComplete );
            printf( "PipelineIngressProxy::serviceThread(): reset detected !\n" );
            for(;;){
                svBitVecVal pipeData[4];

                // Do blocking read on fifo channel.
                localIngress = Ingress.read();

                pipeData[0] = localIngress.Count;

                // Coerce double to long long integer.
                long long llData = (long long)localIngress.Data;
                pipeData[1] = (svBitVecVal)llData;
                llData >>= 32;
                pipeData[2] = (svBitVecVal)llData;
                pipeData[3] = localIngress.Status;

                // Send transaction to PipelineIngressTransactor via a
                // DPI transaction input pipe.
                svSetScope( dHdlContext );
                scemi_pipe_c_send( 1, 4, 4, pipeData, (localIngress.Status==0) );

                // Flush to input pipe if last transaction.
                if( localIngress.Status == 0 )
                    scemi_pipe_c_flush( 1 );
            }
        }
    public:
        PipelineIngressProxy( sc_module_name name, const char *transactorName )
          : sc_module(name){
            SC_THREAD( serviceThread );
            // Establish binding to HDL transactor module instance.
            dHdlContext = svGetScopeFromName( transactorName );
            // Install 'this' sc_module into HDL scope as user context.
            // Use static function address as unique key.
            svPutUserData( dHdlContext, (void *)(&ResetComplete), this );
        }
};

//-------------------------------------------------------
// ResetComplete()
```

```
//
// Imported DPI function. This is used to sync S/W to the
// H/W reset.
//-------------------------------------------------------

void ResetComplete() {
    // First retrieve local sc_module context from HDL scope
    // using svGetScope(), svGetUserData() with static function
    // address of ResetComplete as unique key:

    PipelineIngressProxy *me = (PipelineIngressProxy *)svGetUserData(
        svGetScope(), (void *)(&ResetComplete) );
    me->dResetComplete.notify();
}


//_________________________________
// class PipelineEgressProxy  \_____________________________________________
//
// The PipelineEgressProxy module waits for output transactions from the DUT
// Pipeline module. Those transactions arrive on a DPI output pipe.
//-------------------------------------------------------------------------

class PipelineEgressProxy : public sc_module {
    public:
        sc_fifo_out<MyType> Egress;

    private:
        svScope dHdlContext;

        SC_HAS_PROCESS(PipelineEgressProxy);

        void serviceEgressThread(){
            long long llData;
            svBitVecVal pipeData[4];
            MyType localEgress;
            char lastData;
            int numRead;

            for(;;){
                // Block on a DPI transaction output pipe for transactions from
                // the H/W side.
                svSetScope( dHdlContext );
                scemi_pipe_c_receive( 1, 4, 4, &numRead, pipeData, &lastData );

                assert( numRead == 4 );

                localEgress.Count = pipeData[0];

                // Extract data from sv bit packed array.
                llData = pipeData[2];
                llData <<= 32;
                llData |= pipeData[1];

                // Coerce long long to double.
                localEgress.Data = (double)llData;

                localEgress.Status = pipeData[3];

                // Send to consumer over fifo channel.
                Egress.write( localEgress );

                if( lastData )
```

```
                    printf( "PipelineEgressProxy: last data received.\n" );
                }
            }
        public:
            PipelineEgressProxy( sc_module_name name, const char *transactorName )
                : sc_module(name)
            {   SC_THREAD( serviceEgressThread );

                // Establish binding to HDL transactor module instance.
                dHdlContext = svGetScopeFromName( transactorName );
            }
};
```

```
//================================================================================
//PipelineIngressTransactor
//
// The PipelineIngressTransactor module contains an DPI input
// pipe that is used to stream transactions from the  PipelineIngressProxy
// on the S/W side and routes to the ingress port of the Pipeline DUT.
//
// A separate PipelineEgressTransator handles output from the pipeline
// (see PipelineEgressTransactor.v).
//
// The 'count' field of the input transaction is used to decide how many
// clock cycles to hold the data for before submitting the transaction to the
// DUT pipeline. Once the transaction is submitted as a "token" to the
// pipeline, transactor immediately blocks on the transaction input pipe
// for the next transaction from the S/W side.
//
// When the transaction sent back to the Consumer module, the 'count' argument
// will indicate the total number of clock cycles the transaction spent in
// hardware. This will be the sum of the number of clock advance cycles the
// data is held prior to submission to the pipeline + the number of stages
// in the pipeline (@1 clock/stage).
//================================================================================
module PipelineIngressTransactor(
      //inputs                              outputs
      //--------------------------          ----------------------------
                                            TokenIn,
        Clock, Reset );
// {                                        output [127:0] TokenIn;
                                            reg    [127:0] TokenIn;
        input Clock, Reset;

    reg [31:0] holdingCount;
    reg done;

    `include "scemi_pipes.vh"

    byte lastData;
    int num_read;
    reg [31:0] receivedCount;
    reg [63:0] receivedData;
    reg [31:0] receivedStatus;

    initial lastData = 0;

    import "DPI-C" context function void ResetComplete();


    always begin // {
        @( posedge Clock );

        while( Reset == 1 ) @( posedge Clock );

        ResetComplete;  // Call DPI import function to sync S/W to reset

        while( lastData == 0 ) begin // {
            // Obtain next transaction from S/W.
            scemi_pipe_hdl_receive( 1, 4, 4, num_read,
                {receivedStatus, receivedData, receivedCount}, lastData );

            holdingCount <= receivedCount;
            @(posedge Clock );
```

```
            // Hold the token for the designated holding period.
            while( holdingCount > 0 ) begin
                holdingCount <= holdingCount - 1;
                @(posedge Clock);
            end

            // Ok, now after the holding period submit the token to the
            // pipeline ...
            TokenIn <= { receivedStatus, receivedData, receivedCount };
            @(posedge Clock);

            // Be sure to reset back to 0 so that only idle tokens propagate
            // through the pipeline until the next valid token arrives.
            TokenIn <= 0;
            @(posedge Clock);
        end // }
        while( 1 ) @(posedge Clock ); // Wait indefinitely now the we're done.
    end // }
endmodule // }


//=============================================================================
// PipelineEgressTransactor
//
// The PipelineEgressTransactor module waits for output from the pipeline and
// returns the data as an output transaction back to the Consumer module on the
// software side by sending it over a DPI output pipe.
//=============================================================================

module PipelineEgressTransactor(
      //inputs                              outputs
      //--------------------------          ----------------------------
        TokenOut,
        Clock, Reset );
// {
        input [127:0] TokenOut;
        input Clock, Reset;

    `include "scemi_pipes.vh"

    wire [31:0] countOut;
    wire [63:0] dataOut;
    wire [31:0] statusOut;

    // FSM States
    parameter GetNextOutput = 3'h0;
    parameter Done          = 3'h1;
    reg [2:0] state;

    assign countOut  = TokenOut[31:0];
    assign dataOut   = TokenOut[95:32];
    assign statusOut = TokenOut[127:96];

    always @(posedge Clock) begin // {
        if( Reset )
            state <= GetNextOutput;
        else begin // {
            case( state ) // {
                GetNextOutput: begin // {
                    // Here we wait for output tokens from the Pipeline module.

                    // if( token detected )
```

```
                    //     Send egress transaction to consumer model via
                    //     DPI output pipe.
                    if( TokenOut != 0 ) begin
                        scemi_pipe_hdl_send( 1, 4, 4,
                            {statusOut, dataOut, countOut}, 0 );
                        if( statusOut == 0 ) begin
                            state <= Done;
                            scemi_pipe_hdl_flush( 1 );
                        end
                    end
                end // }

                Done: begin end
            endcase // }
        end // }
    end // }
endmodule // }
```

# Appendix B: Example using dynamic callbacks

*(Informative)*

The example below uses dynamic callbacks to implement a user-defined blocking send function on top of the non-blocking `scemi_pipe_c_try_send` function.

```
static void notify_ok_to_send (
 void *context ) // input: notify context
{
 sc_event *me = (sc_event *)context;
 me->notify();
}

void my_scemi_pipe_c_send(
 void *pipe_handle,          // input: pipe handle
 int num_elements,           // input: #elements to be written
 const svBitVecVal *data,    // input: data
 svBit eom )                 // input: end-of-message marker flag
{
 int byte_offset = 0, elements_sent;
 int pipe_depth = scemi_pipe_get_depth( pipe_handle );
 while( num_elements ){
     elements_sent =
       scemi_pipe_c_try_send(
              pipe_handle, byte_offset, num_elements, data, eom );
     num_elements -= elements_sent;
     if( elements_sent > 0 ){ /* wait till empty pipe or enough space */
       sc_event *ok_to_send = (sc_event *)
              scemi_pipe_get_user_data( pipe_handle, my_scemi_pipe_c_send );
       if( ok_to_send == NULL ) {
              ok_to_send = new sc_event;
              scemi_pipe_put_user_data( pipe_handle, my_scemi_pipe_c_send,
                     ok_to_send);
       }
       scemi_pipe_set_notify_callback( pipe_handle,
              notify_ok_to_send, ok_to_send,
              (num_elements > pipe_depth) ? pipe_depth : num_elements );
       byte_offset += elements_sent
              * scemi_pipe_get_bytes_per_element( pipe_handle );
     wait( *ok_to_send );
     }
 }
}
```

In `my_scemi_pipe_c_send`, when the pipe does not have enough room to fulfill the send request, a one-time dynamic notification callback will be registered to specify the ideal callback condition. That is, when the pipe has enough room to fulfill the send request, or when the pipe becomes empty for requests that are larger than the pipe depth. This blocking send implementation allows an application to be notified whenever the callback condition is met, possibly earlier and no later than the pipe state diagram dictates.

# Appendix C: VHDL SceMiMacros package

*(Informative)*

The following package can be used to supply SCE-MI macro component declarations to an application. Compile this package into the library SceMi and include it in the application code as:

```
library SceMi;
use SceMi.SceMiMacros.all;
```

Here is the source code for the package:

```
library ieee;
use ieee.std_logic_1164.all;

package SceMiMacros is

    component SceMiMessageInPort
        generic( PortWidth: natural );
        port(
            ReceiveReady  : in  std_logic;
            TransmitReady : out std_logic;
            Message       : out std_logic_vector( PortWidth-1 downto 0 ) );
    end component;

    component SceMiMessageOutPort
        generic( PortWidth: natural; PortPriority: natural:=10 );
        port(
            TransmitReady : in  std_logic;
            ReceiveReady  : out std_logic;
            Message       : in  std_logic_vector( PortWidth-1 downto 0 ) );
    end component;

    component SceMiClockPort
        generic(
            ClockNum         : natural := 1;
            RatioNumerator   : natural := 1;
            RatioDenominator : natural := 1;
            DutyHi           : natural := 0;
            DutyLo           : natural := 100;
            Phase            : natural := 0;
            ResetCycles      : natural := 8 );
        port(
            Cclock : out std_logic;
            Creset : out std_logic );
    end component;

    component SceMiClockControl
        generic( ClockNum: natural := 1 );
        port(
         Uclock,
            Ureset               : out std_logic;
            ReadyForCclock       : in  std_logic;
            CclockEnabled        : out std_logic;
            ReadyForCclockNegEdge : in  std_logic;
            CclockNegEdgeEnabled  : out std_logic );
    end component;
end SceMiMacros;
```

# Appendix D: Macro based Multi-clock hardware side interface example

*(Informative)*

Figure D.1shows the top level structure of a simple multi-clock, multi-transactor example.

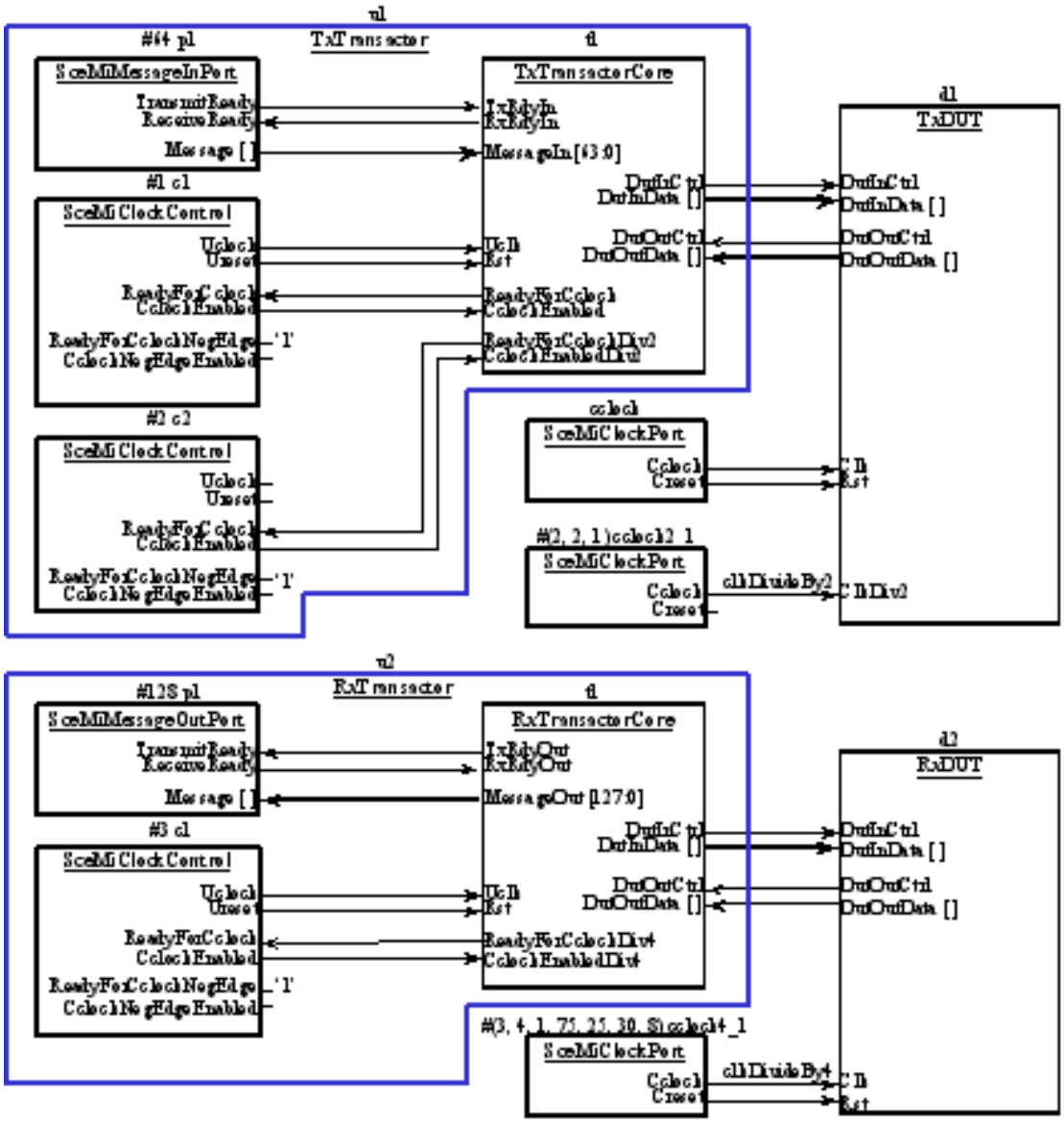



Figure D.1 Multi-clock, multi-transactor example

This design demonstrates the following points.

Three `ClockPort` instances define clocks named `cclock`, `cclock2_1`, and `cclock4_1`.

- Because no parameters are given with the `SceMiClockPort` instance `cclock`, all default parameters are used. This means `cclock` has a `ClockNum=1`, a clock ratio of `1/1`, a *don't care duty cycle*, a phase shift of `0`, and the controlled reset it supplies has an active duration of eight controlled clock cycles.

- The `cclock2_1` instance of `SceMiClockPort` overrides the first three parameters and leaves the rest at their default values. This means `cclock2_1` has a `ClockNum=2`, a clock ratio of `2/1` (i.e., a "divide-by-2" clock), a duty cycle of 50%, a phase shift of `0`, and an eight clock-cycle reset duration.
- The `cclock4_1` instance of `SceMiClockPort` has a `ClockNum=3`, a clock ratio of `4/1` (i.e., a "divide-by-4" clock), a duty cycle of 75%, a phase shift of 30% of the clock period, and an eight clock-cycle reset duration.
- The `TxTransactor` transactor model, named `Bridge.u1`, controls clocks `cclock` and `cclock2_1` since its `SceMiClockControl` macro instances have `ClockNum=1` and `ClockNum=2`, respectively.
- This `TxTransactor` model interfaces to a message input port called `p1` which is parametrized to a bit- width of 64.
- The `RxTransactor` transactor model, named `Bridge.u2`, controls clock `cclock4_1` since its `SceMiClockControl` macro instance has `ClockNum=3`.
- This `RxTransactor` model interfaces to a message input port called `p1` which is parametrized to a bit- width of 128.

The following listing shows some of the VHDL source code for the above schematic.

```
library ieee;
use ieee.std_logic_1164.all;
library SceMi;
use SceMi.SceMiMacros.all;

entity Bridge is end;
architecture Structural of Bridge is
    component TxTransactor is
        port(
            DutInCtrl:  out std_logic;
            DutInData:  out std_logic_vector(31 downto 0);
            DutOutCtrl: in  std_logic;
            DutOutData: in  std_logic_vector(31 downto 0) );
        end component TxTransactor;
    component TxDUT is
        port(
            DutInCtrl:  in  std_logic;
            DutInData:  in  std_logic_vector(31 downto 0);
            DutOutCtrl: out std_logic;
            DutOutData: out std_logic_vector(31 downto 0);
            Clk, Rst, ClkDiv2: in std_logic );
        end component TxDUT;
    component RxTransactor is
        port(
            DutInCtrl:  out std_logic;
            DutInData:  out std_logic_vector(31 downto 0);
            DutOutCtrl: in  std_logic;
            DutOutData: in  std_logic_vector(31 downto 0) );
        end component RxTransactor;
    component RxDUT is
        port(
            DutInCtrl:  in  std_logic;
            DutInData:  in  std_logic_vector(31 downto 0);
            DutOutCtrl: out std_logic;
            DutOutData: out std_logic_vector(31 downto 0);
            Clk, Rst: in std_logic );
        end component RxDUT;
    signal txDutInCtrl, txDutOutCtrl: std_logic;
    signal txDutInData, txDutOutData: std_logic_vector(31 downto 0);
    signal rxDutInCtrl, rxDutOutCtrl: std_logic;
    signal rxDutInData, rxDutOutData: std_logic_vector(31 downto 0);
    signal cclock, creset, clkDivideBy2, clkDivideBy4
            cresetDivideBy4: std_logic;
begin
    u1: TxTransactor port map( txDutInCtrl, txDutInData, txDutOutCtrl,
                               txDutOutData );
    d1: TxDUT        port map( txDutInCtrl, txDutInData, txDutOutCtrl,
                               txDutOutData, cclock, creset, clkDivideBy2 );
    cclock:    SceMiClockPort port map( cclock, creset );
    cclock2_1: SceMiClockPort
        generic map( 2, 2, 1, 50, 50, 0, 8 )
        port map( clkDivideBy2, open );
    u2: RxTransactor port map( txDutInCtrl, txDutInData, txDutOutCtrl,
                               txDutOutData );
    d2: RxDUT        port map( txDutInCtrl, txDutInData, txDutOutCtrl,
                                 txDutOutData, clkDivideBy4, cresetDivideBy4 );
    cclock4_1: SceMiClockPort
        generic map( 3, 4, 1, 75, 25, 30, 8 )
        port map( clkDivideBy2, open );
end;

library ieee;
use ieee.std_logic_1164.all;
```

```
library SceMi;
use SceMi.SceMiMacros.all;

entity TxTransactor is
    port(
        DutInCtrl:  out std_logic;
        DutInData:  out std_logic_vector(31 downto 0);
        DutOutCtrl: in  std_logic;
        DutOutData: in  std_logic_vector(31 downto 0) );
    end;
architecture Structural of TxTransactor is
    component TxTransactorCore is
        port(
            TxRdyIn: in std_logic;           RxRdyIn: out std_logic;
            Message: in std_logic(63 downto 0);
            DutInCtrl:  out std_logic;
            DutInData:  out std_logic_vector(31 downto 0);
            DutOutCtrl: in  std_logic;
            DutOutData: in  std_logic_vector(31 downto 0) );
            Uclk, Rst: in std_logic;
         ReadyForCclock:     in std_logic;
            CclockEnabled:     out std_logic;
            ReadyForCclockDiv2: in std_logic;
            CclockEnabledDiv2: out std_logic;
        end component TxTransactor;
    signal transmitReady, receiveReady: std_logic;
    signal message: std_logic_vector(63 downto 0);
    signal uclock, ureset: std_logic;
    signal readyForCclock, cclockEnabled: std_logic;
    signal readyForCclockDiv2, cclockEnabledDiv2;
begin
    t1: TxTransactorCore port map(
            transmitReady, receiveReady, message,
            DutInCtrl, DutInData, DutOutCtrl, DutOutData,
            uclock, ureset,
           readyForCclock, cclockEnabled, readyForCclockDiv2,
                           cclockEnabledDiv2 );
    p1: SceMiMessageInputPort
            generic map( 64 )
            port map( transmitReady, receiveReady, message );
    c1: SceMiClockControl
            port map( uclock, ureset, readyForCclock, cclockEnabled,
                     '1', open );
    c2: SceMiClockControl
            generic map( 2 )
            port map( open, open, readyForCclockDiv2, cclockEnabledDiv2,
                     '1', open );
end;
```

# Appendix E: Using transaction pipes compatibly with OSCI-TLM applications

*(informative)*

## E.1 TLM Interfaces

The transaction pipes described in this document are designed to dovetail cleanly with OSCI-TLM models. They support the following basic interface operations found in TLM put interfaces and TLM get interfaces which are summarized in the following table:

| **Operations** | **TLM Put Interfaces (`tlm_put_if<T>`)** | **TLM Get Interfaces (`tlm_get_if<T>`)** |
|---|---|---|
| **Blocking Ops** | | |
| `Data transfer` | `::put()` | `::get()` |
| **Non-Blocking Ops** | | |
| `Data transfer` | `::nb_put()` | `::nb_get()` |
| `Query` | `::nb_can_put()` | `::nb_can_get()` |
| `Notify` | `::ok_to_put()` | `::ok_to_get()` |

A typical TLM *put interface* is derived from the following abstract `class tlm_put_if<T>`:

```
template < typename T >
class tlm_blocking_put_if : public virtual sc_interface {
 public:
  virtual void put( const T &t ) = 0;
};

template < typename T >
class tlm_nonblocking_put_if : public virtual sc_interface {
 public:
  virtual bool nb_put( const T &t ) = 0;
  virtual bool nb_can_put( tlm_tag<T> *t = 0 ) const = 0;
  virtual const sc_event &ok_to_put( tlm_tag<T> *t = 0 ) const = 0;
};

template < typename T >
class tlm_put_if :
  public virtual tlm_blocking_put_if< T > ,
  public virtual tlm_nonblocking_put_if< T > {};
```

A typical TLM *get interface* is derived from the following abstract `class tlm_get_if<T>`:

```
template < typename T >
class tlm_blocking_get_if : public virtual sc_interface {
 public:
  virtual T get( tlm_tag<T> *t = 0 ) = 0;
  virtual void get( T &t ) { t = get(); }
};

template < typename T >
class tlm_nonblocking_get_if : public virtual sc_interface {
 public:
  virtual bool nb_get( T &t ) = 0;
  virtual bool nb_can_get( tlm_tag<T> *t = 0 ) const = 0;
  virtual const sc_event &ok_to_get( tlm_tag<T> *t = 0 ) const = 0;
};

template < typename T >
class tlm_get_if :
  public virtual tlm_blocking_get_if< T > ,
  public virtual tlm_nonblocking_get_if< T > {};
```

## E.2 Example of OSCI-TLM compliant proxy model that uses transaction pipes

This proxy class is a derivation of the basic TLM put interface `class tlm_put_if<T>` described in the previous section. Implementations of the required functions of that interface are shown. The example shows how the SCE-MI 2 transaction pipe access functions provide all the necessary functionality to interface this proxy model to the HDL side.

```
//_________________________________                                  _______________
// class PipelineIngressProxy \_________________________________/ johnS 2-5-2006
//
// The PipelineIngressProxy module consumes data received over a data channel
// from producer sends it to the Pipeline DUT on the Verilog side by passing
// transactions over a transaction input pipe to the
// PipelineIngressTransactor.
//-----------------------------------------------------------------------------

template< typename T, const int NUM_WORDS >
class PipelineIngressProxy :
    public sc_module,
    public virtual tlm_put_if<T>
{
  public:
    sc_export< tlm_put_if< T > > put_export;

  private:
    void *m_pipe_handle;
    sc_event m_ok_to_put;

    static void notify_ok_to_put(
            void *context ){           // input: notify context
        sc_event *me = (sc_event *)context;
        me->notify();
    }

    void pack( const T &t, svBitVecVal pipe_data[] );

  public:
     PipelineIngressProxy( sc_module_name name,
         const char *transactor_name )
       : sc_module(name)
    {
        // Bind to channel.
        put_export( *this );

        // Establish binding to transaction input pipe.
        m_pipe_handle = scemi_pipe_c_handle( transactor_name );

        // Register notify "ok to put" callback
        scemi_pipe_set_notify_callback(
            m_pipe_handle, notify_ok_to_put, &m_ok_to_put );
    }

    void put( const T &t ) {
        svBitVecVal pipe_data[ SV_PACKED_DATA_NELEMS(NUM_WORDS*32) ];

        pack( t, pipe_data );

        if( !nb_can_put() )
            wait( m_ok_to_put );

        assert(
            scemi_pipe_c_try_send( m_pipe_handle, 0, 1, pipe_data, 0 ) );
    }

    bool nb_put( const T &t ) {
        if( !nb_can_put() )
            return false;

        svBitVecVal pipe_data[ SV_PACKED_DATA_NELEMS(NUM_WORDS*32) ];
```

```
        pack( t, pipe_data );

        assert(
            scemi_pipe_c_try_send( m_pipe_handle, 0, 1, pipe_data, 0 ) );

        return true;
    }

    bool nb_can_put( tlm_tag<T> *t = 0 ) const {
        return scemi_pipe_c_can_send( m_pipe_handle) == 1; }

    const sc_event &ok_to_put( tlm_tag<T> *t = 0 ) const {
        return m_ok_to_put; }
};

void PipelineIngressProxy<MyType,MyType::NUM_WORDS>::pack(
        const MyType &t, svBitVecVal pipe_data[] )
{
    pipe_data[0] = t.Count;

    // Coerce double to long long integer.
    long long ll_data = (long long)t.Data;

    pipe_data[1] = (svBitVecVal)ll_data;
    ll_data >>= 32;
    pipe_data[2] = (svBitVecVal)ll_data;

    pipe_data[3] = t.Status;
}
```

# Appendix F: Sample Header files for the macro-based SCE-MI

*(informative)*

The ANSI-C file should be used without modification. For the C++ header, extensions are allowable but no modifications can be made to any of the contents that are provided.

## F.1 C++

```
//
// Copyright © 2003-2007 by Accellera
// scemi.h - SCE-MI C++ Interface
//
#ifndef INCLUDED_SCEMI
#define INCLUDED_SCEMI

class SceMiParameters;
class SceMiMessageData;
class SceMiMessageInPortProxy;
class SceMiMessageOutPortProxy;

#define SCEMI_MAJOR_VERSION 2
#define SCEMI_MINOR_VERSION 0
#define SCEMI_PATCH_VERSION 0
#define SCEMI_VERSION_STRING "2.0.0"

/* 32 bit unsigned word type for building and reading messages */
typedef unsigned int SceMiU32;

/* 64 bit unsigned word used for CycleStamps */
typedef unsigned long long SceMiU64;

extern "C" {
typedef int (*SceMiServiceLoopHandler)(void* context, int pending);
};

/*
 * struct SceMiEC - SceMi Error Context
 */

typedef enum {
    SceMiOK,
    SceMiError
} SceMiErrorType;

typedef struct {
    const char* Culprit;   /* The offending function */
    const char* Message;   /* Descriptive message describing problem */
    SceMiErrorType Type;   /* Error code describing the nature of the error */
    int Id;                /* A code to uniquely identify each error */

} SceMiEC;

extern "C" {
typedef void (*SceMiErrorHandler)(void* context, SceMiEC* ec);
};

/*
 * struct SceMiIC - SceMi Informational Message Context
 */

typedef enum {
    SceMiInfo,
    SceMiWarning,
    SceMiNonFatalError
} SceMiInfoType;

typedef struct {
    const char* Originator;
    const char* Message;
```

```
    SceMiInfoType Type;
    int Id;
} SceMiIC;

extern "C" {
typedef void (*SceMiInfoHandler)(void* context, SceMiIC* ic);
};

/*
 * struct SceMiMessageInPortBinding
 *
 * Description
 * -----------
 * This structure defines a tray of callback functions that support
 * propagation of message input readiness back to the software.
 *
 * If an input ready callback is registered (optionally) on a given
 * input port, the port will dispatch the callback whenever becomes
 * ready for more input.
 *
 * Note: All callbacks must take their data and return promptly as they
 * are called possibly deep down in a non-preemptive thread.  Typically,
 * the callback might to some minor manipulation to the context object
 * then return and let a suspended thread resume and do the main process¬ing
 * of the received transaction.
 */

extern "C" {
typedef struct {
    /*
     * This is the user's context object pointer.
     * The application is free to use this pointer for any purposes it
     * wishes.  Neither the class SceMi nor class MessageInputPortProxy do
     * anything with this pointer other than store it and pass it when

     * calling functions.
     */
    void* Context;

    /*
     * Receive a response transaction.  This function is called when data
     * from the message output port arrives.  This callback acts as a proxy
     * for the message output port of the transactor.
     */
    void (*IsReady)(
        void* context);

    /*
     * This function is called from the MessageInputPortProxy destructor
     * to notify the user code that the reference to the 'context' pointer
     * has been deleted.
     */
    int (*Close)(
        void* context);

} SceMiMessageInPortBinding;
};

/*
 * struct SceMiMessageOutPortBinding
 *
 * Description
 * -----------
```

```
 * This structure defines a tray of callback functions are passed to the class
 * SceMi when the application model binds to a message output port proxy and
 * which are called on message receipt and close notification.  It is the means
 * by which the MessageOutputPort forwards received transactions to the C model.
 *
 * Note: All callbacks must take their data and return promptly as they
 * are called possibly deep down in a non-preemptive thread.  Typically,
 * the callback might to some minor manipulation to the context object
 * then return and let a suspended thread resume and do the main process¬ing
 * of the received transaction.
 *
 * Additionally, the message data passed into the receive callback is
 * not guaranteed to remain the same once the callback returns.  All
 * data therein then must be processed while inside the callback.
 */

extern "C" {
typedef struct {
    /*
     * This is the user's context object pointer.
     * The application is free to use this pointer for any purposes it

     * wishes.  Neither the class SceMi nor class SceMiMessageOutPortProxy do
     * anything with this pointer other than store it and pass it when
     * calling callback functions Receive and Close.
     */
    void* Context;

    /*
     * Receive a response transaction.  This function is called when data
     * from the message output port arrives.  This callback acts as a proxy
     * for the message output port of the transactor.
     */
    void (*Receive)(
        void* context,
        const SceMiMessageData* data);

    /*
     * This function is called from the MessageOutputPortProxy destructor
     * to notify the user code that the reference to the 'context' pointer
     * has been deleted.
     */
    int (*Close)(
        void* context);

} SceMiMessageOutPortBinding;
};

class SceMiParameters {

  public:
    // CREATORS

    //
    // This constructor initializes some parameters from the
    // parameters file in the config directory, and some other
    // parameters directly from the config file.
    //
    SceMiParameters(
        const char* paramsfile,
        SceMiEC* ec = 0);

    ~SceMiParameters();
```

```
    // ACCESSORS

    //
    // This accessor returns the number of instances of objects of
    // the specified objectKind name.
    //
    unsigned int NumberOfObjects(
        const char* objectKind,    // Input: Object kind name.
        SceMiEC* ec = 0) const; // Input/Output: Error status.

    //

    // These accessors return an integer or string attribute values of the
    // given object kind.  It is considered an error if the index > number
    // returned by ::NumberOfObjects() or the objectKind and attributeName
    // arguments are unrecognized.
    //
    int AttributeIntegerValue(
        const char* objectKind,    // Input: Object kind name.
        unsigned int index,        // Input: Index of object instance.
        const char* attributeName, // Input: Name of attribute being read.
        SceMiEC* ec = 0) const;    // Input/Output: Error status.

    const char* AttributeStringValue(
        const char* objectKind,    // Input: Object kind name.
        unsigned int index,        // Input: Index of object instance.
        const char* attributeName, // Input: Name of attribute being read.
        SceMiEC* ec = 0) const;    // Input/Output: Error status.

    // MANIPULATORS

    //
    // These manipulators override an integer or string attribute values of the
    // given object kind.  It is considered an error if the index > number
    // returned by ::NumberOfObjects(). or the objectKind and attribute¬Name
    // arguments are unrecognized.
    //
    void OverrideAttributeIntegerValue(
        const char* objectKind,    // Input: Object kind name.
        unsigned int index,        // Input: Index of object instance.
        const char* attributeName, // Input: Name of attribute being read.
        int value,                 // Input: New integer value of attribute.
        SceMiEC* ec = 0);          // Input/Output: Error status.

    void OverrideAttributeStringValue(
        const char* objectKind,    // Input: Object kind name.
        unsigned int index,        // Input: Index of object instance.
        const char* attributeName, // Input: Name of attribute being read.
        const char* value,         // Input: New string value of attribute.
        SceMiEC* ec = 0);          // Input/Output: Error status.
};

//
// class SceMiMessageInPortProxy
//
// Description
// -----------
// The class SceMiMessageInPortProxy presents a C++ proxy for a transactor
// message input port.  The input channel to that transactor is repre¬sented
// by the Send() method.
//
```

```
class SceMiMessageInPortProxy {

  public:
    // ACCESSORS
    const char* TransactorName() const;
    const char* PortName() const;
    unsigned int PortWidth() const;

    //
    // This method sends message to the transactor input port.
    //
    void Send(
        const SceMiMessageData &data, // Message payload to be sent.
        SceMiEC* ec = 0);

    //
    // Replace port binding.
    // The binding argument represents a callback function and context
    // pointer tray (see comments in scemicommontypes.h for struct
    // SceMiMessageInPortBinding).
    //
    void ReplaceBinding(
        const SceMiMessageInPortBinding* binding = 0,
        SceMiEC* ec = 0);
};

//
// class SceMiMessageOutPortProxy
//
// Description
// -----------
// The class SceMiMessageOutPortProxy presents a C++ proxy for a transac¬tor
// message output port.
//
class SceMiMessageOutPortProxy {
  public:
    // ACCESSORS
    const char* TransactorName() const;
    const char* PortName() const;
    unsigned int PortWidth() const;

    //
    // Replace port binding.
    // The binding argument represents a callback function and context
    // pointer tray (see comments in scemicommontypes.h for struct
    // SceMiMessageOutPortBinding).
    //
    void ReplaceBinding(
        const SceMiMessageOutPortBinding* binding = 0,
        SceMiEC* ec = 0);
};

//
// class SceMiMessageData
//

// Description
// -----------
// The class SceMiMessageData represents a fixed length array of data which
// is transferred between models.
//
class SceMiMessageData {
  public:
```

```
    // CREATORS

    //
    // Constructor: The message in port proxy for which
    // this message data object must be suitably sized.
    //
    SceMiMessageData(
        const SceMiMessageInPortProxy& messageInPortProxy,
        SceMiEC* ec = 0);

    ~SceMiMessageData();

    // Return size of vector in bits
    unsigned int WidthInBits() const;

    // Return size of array in 32 bit words.
    unsigned int WidthInWords() const;

    void Set( unsigned i, SceMiU32 word, SceMiEC* ec = 0);

    void SetBit( unsigned i, int bit, SceMiEC* ec = 0);

    void SetBitRange(
        unsigned int i, unsigned int range, SceMiU32 bits, SceMiEC* ec = 0);

    SceMiU32 Get( unsigned i, SceMiEC* ec = 0) const;

    int GetBit( unsigned i, SceMiEC* ec = 0) const;

    SceMiU32 GetBitRange(
        unsigned int i, unsigned int range, SceMiEC* ec = 0) const;

    SceMiU64 CycleStamp() const;
};

//
// class SceMi
//
// Description
// -----------
// This file defines the public interface to class SceMi.
//

class SceMi {
  public:

    //
    // Check version string against supported versions.
    // Returns -1 if passed string not supported.
    // Returns interface version # if it is supported.
    // This interface version # can be passed to SceMi::Init().
    //
    static int Version(
        const char* versionString);

    //
    // This function wraps constructor of class SceMi.  If an instance
    // of class SceMi has been established on a prior call to the
    // SceMi::Init() function, that pointer is returned since a single
    // instance of class SceMi is reusable among all C models.
    // Returns NULL if error occurred, check ec for status or register
    // an error callback.
    //
```

```
// The caller is required to pass in the version of SceMi it is
// expecting to work with.  Call SceMi::Version to convert a version
// string to an integer suitable for this version's "version" argu¬ment.
//
// The caller is also expected to have instantiated a SceMiParameters
// object, and pass a pointer to that object into this function.
//
static SceMi*
Init(
    int version,
    const SceMiParameters* parameters,
    SceMiEC* ec = 0);

//
// Shut down the SCEMI interface.
//
static void
Shutdown(
    SceMi* mct,
    SceMiEC* ec = 0);

//
// Create proxy for message input port.
//
// Pass in the instance name in the bridge netlist of
// the transactor and port to which binding is requested.
//
// The binding argument is a callback function and context
// pointer tray.  For more details, see the comments in
// scemicommontypes.h by struct SceMiMessageInPortBinding.
//
SceMiMessageInPortProxy*
BindMessageInPort(
    const char* transactorName,
    const char* portName,
    const SceMiMessageInPortBinding* binding = 0,

    SceMiEC* ec = 0);

//
// Create proxy for message output port.
//
// Pass in the instance name in the bridge netlist of
// the transactor and port to which binding is requested.
//
// The binding argument is a callback function and context
// pointer tray.  For more details, see the comments in
// scemicommontypes.h by struct SceMiMessageOutPortBinding.
//
SceMiMessageOutPortProxy*
BindMessageOutPort(
    const char* transactorName,
    const char* portName,
    const SceMiMessageOutPortBinding* binding = 0,
    SceMiEC* ec = 0);

//
// Service arriving transactions from the portal.
// Messages enqueued by SceMiMessageOutPortProxy methods, or which are
// are from output transactions that pending dispatch to the
// SceMiMessageOutPortProxy callbacks, may not be handled until
// ServiceLoop() is called.  This function returns the # of output
// messages that were dispatched.
```

```
    //
    // Regarding the service loop handler (aka "g function"):
    // If g is NULL, check for transfers to be performed and
    // dispatch them returning immediately afterwards.  If g is
    // non-NULL, enter into a loop of performing transfers and
    // calling 'g'.  When 'g' returns 0 return from the loop.
    // When 'g' is called, an indication of whether there is at
    // least 1 message pending will be made with the 'pending' flag.
    //
    // The user context object pointer is uninterpreted by
    // ServiceLoop() and is passed straight to the 'g' function.
    //
    int
    ServiceLoop(
        SceMiServiceLoopHandler g = 0,
        void* context = 0,
        SceMiEC* ec = 0);

    //
    // Register an error handler which is called in the event
    // that an error occurs.  If no handler is registered, the
    // default error handler is called.
    //
    static void
    RegisterErrorHandler(
        SceMiErrorHandler errorHandler,
        void* context);


    //
    // Register an info handler which is called in the event
    // that a text message needs to be issued.  If no handler
    // is registered, the message is printed to stdout in
    // Ikos message format.
    //
    static void
    RegisterInfoHandler(
        SceMiInfoHandler infoHandler,
        void* context);
};

#endif
```

## F.2 ANSI-C

```
/*
 * scemi.h
 *
 * Copyright © 2003-2007 by Accellera
 * This file is the header file for the SCEMI C API.
 */
#ifndef INCLUDED_SCEMI
#define INCLUDED_SCEMI

typedef void SceMi;
typedef void SceMiParameters;
typedef void SceMiMessageData;
typedef void SceMiMessageInPortProxy;
typedef void SceMiMessageOutPortProxy;

#define SCEMI_MAJOR_VERSION 2
#define SCEMI_MINOR_VERSION 0
#define SCEMI_PATCH_VERSION 0
#define SCEMI_VERSION_STRING "2.0.0"

/* 32 bit unsigned word type for building and reading messages */
typedef unsigned int SceMiU32;

/* 64 bit unsigned word used for CycleStamps */
typedef unsigned long long SceMiU64;

typedef int (*SceMiServiceLoopHandler)(void* context, int pending);

/*
 * struct SceMiEC - SceMi Error Context
 */

typedef enum {
    SceMiOK,
    SceMiError
} SceMiErrorType;

typedef struct {
    const char* Culprit;   /* The offending function */
    const char* Message;   /* Descriptive message describing problem */
    SceMiErrorType Type;   /* Error code describing the nature of the error */
    int Id;                /* A code to uniquely identify each error */
} SceMiEC;

typedef void (*SceMiErrorHandler)(void* context, SceMiEC* ec);

/*
 * struct SceMiIC - SceMi Informational Message Context

 */

typedef enum {
    SceMiInfo,
    SceMiWarning,
    SceMiNonFatalError
} SceMiInfoType;

typedef struct {
    const char* Originator;
    const char* Message;
    SceMiInfoType Type;
```

```
    int Id;
} SceMiIC;

typedef void (*SceMiInfoHandler)(void* context, SceMiIC* ic);

/*
 * struct SceMiMessageInPortBinding
 *
 * Description
 * -----------
 * This structure defines a tray of callback functions that support
 * propagation of message input readiness back to the software.
 *
 * If an input ready callback is registered (optionally) on a given
 * input port, the port will dispatch the callback whenever becomes
 * ready for more input.
 *
 * Note: All callbacks must take their data and return promptly as they
 * are called possibly deep down in a non-preemptive thread.  Typically,
 * the callback might to some minor manipulation to the context object
 * then return and let a suspended thread resume and do the main process¬ing
 * of the received transaction.
 */

typedef struct {
    /*
     * This is the user's context object pointer.
     * The application is free to use this pointer for any purposes it
     * wishes.  Neither the class SceMi nor class MessageInputPortProxy do
     * anything with this pointer other than store it and pass it when
     * calling functions.
     */
    void* Context;

    /*
     * Receive a response transaction.  This function is called when data
     * from the message output port arrives.  This callback acts as a proxy
     * for the message output port of the transactor.
     */
    void (*IsReady)(
        void* context);


    /*
     * This function is called from the MessageInputPortProxy destructor
     * to notify the user code that the reference to the 'context' pointer
     * has been deleted.
     */
    int (*Close)(
        void* context);

} SceMiMessageInPortBinding;

/*
 * struct SceMiMessageOutPortBinding
 *
 * Description
 * -----------
 * This structure defines a tray of callback functions are passed to the class
 * SceMi when the application model binds to a message output port proxy and
 * which are called on message receipt and close notification.  It is the means
 * by which the MessageOutputPort forwards received transactions to the C model.
 *
```

```
 * Note: All callbacks must take their data and return promptly as they
 * are called possibly deep down in a non-preemptive thread.  Typically,
 * the callback might to some minor manipulation to the context object
 * then return and let a suspended thread resume and do the main process¬ing
 * of the received transaction.
 *
 * Additionally, the message data passed into the receive callback is
 * not guaranteed to remain the same once the callback returns.  All
 * data therein then must be processed while inside the callback.
 */

typedef struct {
    /*
     * This is the user's context object pointer.
     * The application is free to use this pointer for any purposes it
     * wishes.  Neither the class SceMi nor class SceMiMessageOutPortProxy do
     * anything with this pointer other than store it and pass it when
     * calling callback functions Receive and Close.
     */
    void* Context;

    /*
     * Receive a response transaction.  This function is called when data
     * from the message output port arrives.  This callback acts as a proxy
     * for the message output port of the transactor.
     */
    void (*Receive)(

        void* context,
        const SceMiMessageData* data);

    /*
     * This function is called from the MessageOutputPortProxy destructor
     * to notify the user code that the reference to the 'context' pointer
     * has been deleted.
     */
    int (*Close)(
        void* context);

} SceMiMessageOutPortBinding;

/*
 * Register an error handler which is called in the event
 * that an error occurs.  If no handler is registered, the
 * default error handler is called.  The errorHandler will
 * pass back the 'context' object registered by the user
 * when making this function call.  The system makes no
 * assumptions about the 'context' pointer and will not
 * modify it.
 */
void
SceMiRegisterErrorHandler(
    SceMiErrorHandler errorHandler,
    void* context);

/*
 * Register an info handler which is called in the event
 * that an informational text message needs to be printed.
 * If no handler is registered, the message is printed to stdout.
 */
void SceMiRegisterInfoHandler(
    SceMiInfoHandler infoHandler,
    void* context );
```

```
/*
 * Check version string against supported versions.
 * Return -1 if passed string not supported.
 * Return interface version # if it is supported. This interface
 * version # can be passed to the SceMiInit() function.
 */
int
SceMiVersion(
    const char* versionString);

/*
 * This function wraps constructor of class SceMi.  If an instance
 * of class SceMi has been established on a prior call to the
 * the SceMiInit() function, that pointer is returned since a single
 * instance of class SceMi is reusable among all C models.
 *
 * The caller must provide the interface version # it is expecting
 * to work with.  If the caller requests an unsupported version,

 * an error is returned.
 *
 * The caller must also provide a pointer to a filled-in SceMiParameters
 * struct that contains global interface specification parameters.
 *
 * Returns NULL if error occurred, check ec for status or register
 * an error callback.
 */
SceMi*
SceMiInit(
    int version,
    const SceMiParameters* parameters,
    SceMiEC* ec);

/*
 * Shut down the specified SCEMI interface.
 */
void
SceMiShutdown(
    SceMi* mctHandle,
    SceMiEC* ec);

/*
 * Create proxy for message input port.
 *
 * The caller must provide the handle to the initialized SceMi system,
 * as well as the name of the transactor and port to which binding
 * is requested.
 *
 * The 'binding' input is a callback function and context pointer tray.
 * See the comments in scemitypes.h for struct SceMiMessageInPortBinding.
 */
SceMiMessageInPortProxy*
SceMiBindMessageInPort(
    SceMi* mctHandle,
    const char* transactorName,
    const char* portName,
    const SceMiMessageInPortBinding* binding,
    SceMiEC* ec);

/*
 * Create proxy for message output port.
 *
```

```
 * The caller must provide the handle to the initialized SceMi system,
 * as well as the name of the transactor and port to which binding
 * is requested.
 *
 * The 'binding' input is a callback function and context pointer tray.
 * See the comments in scemitypes.h for struct SceMiMessageOutPortBind¬ing.
 */
SceMiMessageOutPortProxy*
SceMiBindMessageOutPort(
    SceMi* mctHandle,

    const char* transactorName,
    const char* portName,
    const SceMiMessageOutPortBinding* binding,
    SceMiEC* ec);

/*
 * Service arriving transactions from the portal.
 * Messages enqueued by SceMiMessageOutPortProxy methods, or which are
 * are from output transactions that pending dispatch to the
 * SceMiMessageOutPortProxy callbacks, may not be handled until
 * ServiceLoop() is called.  This function returns the # of output
 * messages that were dispatched.
 *
 * The 'g' input is a pointer to a user-defined service function.
 * If g is NULL, check for transfers to be performed and
 * dispatch them returning immediately afterwards.  If g is
 * non-NULL, enter into a loop of performing transfers and
 * calling 'g'.  When 'g' returns 0 return from the loop.
 * When 'g' is called, an indication of whether there is at
 * least 1 message pending will be made with the 'pending' flag.
 *
 * The 'context' input is a user context object pointer.
 * This pointer is uninterpreted by the SceMiServiceLoop()
 * method and is passed on to the 'g' callback function.
 */
int
SceMiServiceLoop(
    SceMi* mctHandle,
    SceMiServiceLoopHandler g,
    void* context,
    SceMiEC* ec);


SceMiParameters*
SceMiParametersNew(
    const char* paramsFile,
    SceMiEC* ec);

unsigned int
SceMiParametersNumberOfObjects(
    const SceMiParameters* parametersHandle,
    const char* objectKind,
    SceMiEC* ec);

int
SceMiParametersAttributeIntegerValue(
    const SceMiParameters* parametersHandle,
    const char* objectKind,
    unsigned int index,
    const char* attributeName,
    SceMiEC* ec);
```

```
const char*
SceMiParametersAttributeStringValue(

    const SceMiParameters* parametersHandle,
    const char* objectKind,
    unsigned int index,
    const char* attributeName,
    SceMiEC* ec);

void
SceMiParametersOverrideAttributeIntegerValue(
    SceMiParameters* parametersHandle,
    const char* objectKind,
    unsigned int index,
    const char* attributeName,
    int value,
    SceMiEC* ec);

void
SceMiParametersOverrideAttributeStringValue(
    SceMiParameters* parametersHandle,
    const char* objectKind,
    unsigned int index,
    const char* attributeName,
    const char* value,
    SceMiEC* ec);

/*
 * SceMiMessageData initialization function.
 * This is called to construct a new SceMiMessageData object.
 */
SceMiMessageData*
SceMiMessageDataNew(
    const SceMiMessageInPortProxy* messageInPortProxyHandle,
    SceMiEC* ec);

/*
 * Destroy a SceMiMessageData object previously returned from
 * SceMiMessageDataNew.
 */
void
SceMiMessageDataDelete(
    SceMiMessageData* messageDataHandle);

/*
 * Return size of message data array in 32 bit words.
 */
unsigned int
SceMiMessageDataWidthInBits(
    const SceMiMessageData* messageDataHandle);

/*
 * Return size of array in 32 bit words.
 */
unsigned int
SceMiMessageDataWidthInWords(
    const SceMiMessageData* messageDataHandle);


/*
 * Set value of message data word at given index.
 */
void
```

```
SceMiMessageDataSet(
    SceMiMessageData* messageDataHandle,
    unsigned int i,
    SceMiU32 word,
    SceMiEC* ec);

/*
 * Set bit in message data word at given index.
 */
void
SceMiMessageDataSetBit(
    SceMiMessageData* messageDataHandle,
    unsigned int i,
    int bit,
    SceMiEC* ec);

/*
 * Set bit range in message data word at given index.
 */
void SceMiMessageDataSetBitRange(
    SceMiMessageData* messageDataHandle,
    unsigned int i,
    unsigned int range,
    SceMiU32 bits,
    SceMiEC *ec);

/*
 * Return value of message data word at given index.
 */
SceMiU32
SceMiMessageDataGet(
    const SceMiMessageData* messageDataHandle,
    unsigned int i,
    SceMiEC* ec);

/*
 * Return value of bit in message data word at given index.
 */
int
SceMiMessageDataGetBit(
    const SceMiMessageData* messageDataHandle,
    unsigned int i,
    SceMiEC* ec);

/*
 * Return value of bit range in message data word at given index.
 */
SceMiU32
SceMiMessageDataGetBitRange(

    const SceMiMessageData *messageDataHandle,
    unsigned int i,
    unsigned int range,
    SceMiEC *ec);

/*
 * Get cyclestamp.
 */
SceMiU64
SceMiMessageDataCycleStamp(
    const SceMiMessageData* messageDataHandle);
```

```
/*
 * This method sends a message with the specified payload to the
 * transactor input port.  The data will transparently be delivered
 * to the transactor as 1 or more chunks.
 */
void
SceMiMessageInPortProxySend(
    SceMiMessageInPortProxy* messageInPortProxyHandle,
    const SceMiMessageData* messageDataHandle,
    SceMiEC* ec);

const char*
SceMiMessageInPortProxyTransactorName(
    const SceMiMessageInPortProxy* messageInPortProxyHandle);

const char*
SceMiMessageInPortProxyPortName(
    const SceMiMessageInPortProxy* messageInPortProxyHandle);

unsigned int
SceMiMessageInPortProxyPortWidth(
    const SceMiMessageInPortProxy* messageInPortProxyHandle);

const char*
SceMiMessageOutPortProxyTransactorName(
    const SceMiMessageOutPortProxy* messageOutPortProxyHandle);

const char*
SceMiMessageOutPortProxyPortName(
    const SceMiMessageOutPortProxy* messageOutPortProxyHandle);

unsigned int
SceMiMessageOutPortProxyPortWidth(
    const SceMiMessageOutPortProxy* messageOutPortProxyHandle);

#endif
```

# Appendix G: Sample Header File for Basic Transaction Pipes C-Side API

As described in section 5.8 the infrastructure will provide the implementations of the actual built-in functions for the transaction pipes on both the C side and the HDL side. On the HDL side, the pipe functions and tasks in the SystemVerilog interface can be can be implementation supplied HDL definitions of built-in functions that perform the operations of the pipe inside. For example a pipe call can code that does PLI or DPI calls inside the tasks/functions in the SystemVerilog interface definitions. The way pipe functions are defined is up to the implementation but they must have the exact profiles shown in section 5.8.2.

The following a precise listing of the include file that can be included by the C application to declare the API functions. The name of this file is `scemi_pipes.h`.

```
#ifndef _scemi_pipes_h
#define _scemi_pipes_h

#include "svdpi.h"

#ifdef __cplusplus
extern "C" {
#endif

//--------------------------------------------------------------------------
// scemi_pipe_c_handle()
//
// This function retreives an opaque handle representing a transaction
// input or output pipe given an HDL scope and a pipe ID.
//--------------------------------------------------------------------------

void *scemi_pipe_c_handle(        // return: pipe handle
    const char *endpoint_path );  // input: path to HDL endpoint instance

//--------------------------------------------------------------------------
// scemi_pipe_get_direction()
// scemi_pipe_get_depth()
// scemi_pipe_get_bytes_per_element()

//
// This function returns the direction, depth, and bytes per element of a previous
defined pipe, given the pipe handle
//--------------------------------------------------------------------------

svBit scemi_pipe_get_direction(  // return: 1 for input pipe, 0 for output pipe
    void *pipe_handle );           // input: pipe handle

int scemi_pipe_get_depth( // return: current depth (in elements) of the pipe
    void *pipe_handle );  // input: pipe handle

int scemi_pipe_get_bytes_per_element( // return: bytes per element
    void *pipe_handle );              // input: pipe handle


//--------------------------------------------------------------------------
// scemi_pipe_c_send()
//
// These is the basic blocking send function for a transaction input pipe.
// The passed in data is sent to the pipe. If necessary the calling thread
// is suspended until there is room in the pipe.
//
// The eom arg is a flag which is used for user specified end-of-message (eom)
// indication. It can be used for example to mark the end of a frame containing
// a sequence of transactions.
//
// scemi_pipe_c_receive()
//
// This is the basic blocking receive function for a transaction output pipe.
//
// The eom argument for this call is an output argument. It is set to the
// same settings of the flag passed on the send end of the pipe as described
// above. Thus is can be used by the caller to query whether the current
// read is one for which an eom was specified when the data was written on
// the send end.
//
// Both the send() and receive() calls are thread-aware. They can be
// easily implemented using a simple reference implementation that makes
// use of the non-blocking thread-neutral interface decribed below
```

```
// in conjunction with a selected threading system.
//------------------------------------------------------------------------

void scemi_pipe_c_send(
    void *pipe_handle,       // input: pipe handle
    int num_elements,        // input: #elements to be written
    const svBitVecVal *data, // input: data
    svBit eom );             // input: end-of-message marker flag (and flush)

void scemi_pipe_c_receive(
    void *pipe_handle,       //  input: pipe handle
    int num_elements,        //  input: #elements to be read
    int *num_elements_valid, // output: #elements that are valid
    svBitVecVal *data,       // output: data
    svBit *eom );            // output: end-of-message marker flag (and flush)

//------------------------------------------------------------------------
// scemi_pipe_c_flush()
//
// Flush pipelined data.
//------------------------------------------------------------------------

void scemi_pipe_c_flush(
    void *pipe_handle );    // input: pipe handle

//------------------------------------------------------------------------
// scemi_pipe_c_try_send()
//
// This is the basic non-blocking send function for a transaction input pipe.
// If there is room in the pipe for the indicated number of elements, the
// data is transferred to the pipe and a success status of 1 is returned.
// Otherwise, nothing is done with the data and a status of 0 is returned.
//
// This function is thread-neutral can can be used to create a reference
// implementation of the blocking send function (scemi_pipe_c_send)
// over a selected C-based threading environment.
//
// scemi_pipe_c_try_receive()
//
// This is the basic non-blocking receive function for a transaction output
// pipe. If the indicated number of elements exist in the pipe, the data is
// transferred out of the pipe and a success status of 1 is returned.
// Otherwise, the data in the pipe is left alone and a status of 0 is returned.
//
// This function is thread-neutral can can be used to create a reference
// implementation of the blocking receive function (scemi_pipe_c_receive)
// over a selected C-based threading environment.
//
//------------------------------------------------------------------------

int scemi_pipe_c_try_send(
    void *pipe_handle,       // input: pipe handle
    int byte_offset,         // input: byte offset within data array
    int num_elements,        // input: #elements to be written
    const svBitVecVal *data, // input: data
    svBit eom );             // input: end-of-message marker flag

int scemi_pipe_c_try_receive(
    void *pipe_handle,       //  input: pipe handle
    int byte_offset,         //  input: byte offset within data array
    int num_elements,        //  input: #elements to be read
        svBitVecVal *data,       // output: data
    svBit *eom );            // output: end-of-message marker flag
```

```
//------------------------------------------------------------------------------
// scemi_pipe_c_try_flush()
//------------------------------------------------------------------------------

int scemi_pipe_c_try_flush(
    void *pipe_handle );     // input: pipe handle

//------------------------------------------------------------------------------
// scemi_pipe_c_in_flush_state()
//------------------------------------------------------------------------------

svBit scemi_pipe_c_in_flush_state( // return: whether pipe is in Flush state
void *pipe_handle );               // input: pipe handle

//------------------------------------------------------------------------------
// scemi_pipe_c_can_send()
//
// This is function indicates if there is currently space in the pipe
// for the indicated number of elements meaning that the next call to
// scemi_pipe_c_send() will succeed without requiring a block.
//
// scemi_pipe_c_can_receive()
//
// This is function indicates if there is currently at least the indicated
// number of elements in the pipe meaning that the next call to
// scemi_pipe_c_receive() will succeed without requiring a block.
//
// For both of these calls, a return value 0 indicates that the operation
// cannot succeed, 1 indicates that it can succeed.
//------------------------------------------------------------------------------

int scemi_pipe_c_can_send(
    void *pipe_handle );

int scemi_pipe_c_can_receive(
    void *pipe_handle );

//------------------------------------------------------------------------------
// Notify callback support
//

typedef void (*scemi_pipe_notify_callback)(
    void *context );       // input: C model context

typedef void *scemi_pipe_notify_callback_handle;
    // Handle type denoting registered notify callback.

#ifdef __cplusplus

scemi_pipe_notify_callback_handle scemi_pipe_set_notify_callback(
    void *pipe_handle,      // input: pipe handle
    scemi_pipe_notify_callback notify_callback,
                            // input: notify callback function
    void *notify_context,   // input: notify context
    int callback_threshold=0 ); // input: threshold for notify callback function

#else // __cplusplus

scemi_pipe_notify_callback_handle scemi_pipe_set_notify_callback(
    void *pipe_handle,      // input: pipe handle
    scemi_pipe_notify_callback notify_callback,
                            // input: notify callback function
```

```
    void *notify_context,   // input: notify context
    int callback_threshold ); // input: threshold for notify callback function

#endif // __cplusplus

void scemi_pipe_clear_notify_callback(
    scemi_pipe_notify_callback_handle notify_callback_handle );
     // input: notify callback handle

void *scemi_pipe_get_notify_context( //return: notify context object pointer
    scemi_pipe_notify_handle notify_callback_handle ); // input: notify handle

//------------------------------------------------------------------------
// Per-pipe user data storage support
//

void scemi_pipe_put_user_data(
    void *pipe_handle,      // input: pipe handle
    void *user_key,         // input: user key
    void *user_data);       // input: user data

void *scemi_pipe_get_user_data(
    void *pipe_handle,      // input: pipe handle
    void *user_key);        // input: user key


//------------------------------------------------------------------------
// Autoflush support
//

svBit scemi_pipe_set_eom_auto_flush(
     void *pipe_handle,  // input: pipe handle
     svBit enabled );    // input: enable/disable

#ifdef __cplusplus
} /* extern "C" */
#endif

#endif // _scemi_pipes_h
```

# Appendix H: Bibliography

*(informative)*


[B1] The IEEE Standard Dictionary of Electrical and Electronics Terms, Sixth Edition. (1997)

[B2] SystemC, Version 2.0 User's Guide, www.systemc.org.

[B3] Standard Co-Emulation Modeling Interface SCE-MI) Reference Manual - Version 1.1.0 - January 13th, 2005 - Accellera ITC

[B4] IEEE P1800™/Draft 6 p1: Draft Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language - SystemVerilog Working Group Design Automation Standards Committee IEEE Computer Society

[B5] Using SystemVerilog Now with DPI - Rich Edelman, Doug Warmke

[B6] Integrating SystemC Models with Verilog and SystemVerilog Models Using the SystemVerilog Direct Programming Interface [DPI] - Stuart Sutherland, Sutherland HDL, Inc.