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

Version 1.1.0

January 13th, 2005

Notices

The information contained in this manual represents the definition of the SCE-MI as reviewed and released by Accellera in January 2005.

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

brian_bailey@acm.org

or to the address below.

The current Working Group's website address is

www.eda.org/itc

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

Published as: SCE-MI Reference Manual
Version 1.1.0, January 13th, 2005.

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.

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 SCE-MI Reference Manual Version 1.1.0 or to prior versions of this standard.

Jason Andrews	Axis	
Brian Bailey	Independent Consultants	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	

Revision history:

Version 1.0	05/29/03
Version 1.1	1/13/05

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

Table of Contents

1. Overview

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 multichannel communication interface which addresses these challenges and caters to the needs of both emulation end-users and emulation suppliers.

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

- All emulators on the market today have proprietary APIs. 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 vendors, making it possible for IP providers to write a single model.

The SCE-MI can also be used to solve the following emulation 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.

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 untimed *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 untimed, cycle-accurate, and even gate-level, the focus of SCE-MI Version 1.1.0 is to interface purely untimed 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 subcycle-accurate simulation environments.
— It is possible, but not ideal, to use this to bridge cycle accurate simulation environments.
— It is best used for bridging an untimed simulation environment with a cycle-accurate simulation environment.

See Appendix D for some recommendations on connecting to event-based simulation environments.

NOTE—There are many references in the document to SystemC (see [B2]) as the modeling environment for untimed 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 untimed modeling. As such, it has been chosen for use in many of the examples in this document.

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 (see [B2]). 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 untimed 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 *italicized*. See Chapter 3 for further definitions of these terms.

1.6 Contents of this standard

The organization of the remainder of this standard is

— Chapter 2 (References) provides references to other applicable standards that are assumed or required for this standard.
— Chapter 3 (Definitions) defines terms used throughout this standard.
— Chapter 4 (Use model) provides an overall description and use model for the SCE Modeling Interface (SCE-MI).
— Chapter 5 (Formal specification) is a formal functional specification of the API itself.

- Appendix A (Tutorial) is a tutorial showing the use model in a simple application.
- Appendix B (Multi-clock hardware side interface example) provides a simple multi-clock, multi-transactor schematic example and its VHDL code listing.
- Appendix C (VHDL SceMiMacros package) provides a VHDL package which can be used to supply SCE-MI macro component declarations to an application.
- Appendix D (Applying the SCE-MI to event-based systems) provides some recommendations on connecting to event-based simulation environments.
- Appendix E (Sample header files for the SCE-MI) provides headers for both C and C++ implementations.
- Appendix F (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.

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

IEEE Std 1364-2001, IEEE Standard for Verilog Hardware Description Language.

3. Definitions

For the purposes of this standard, the following terms and definitions apply. The *IEEE Standard Dictionary of Electrical and Electronics Terms* [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 5) when, in fact, it is a disguised BCA model (see Figure 6).

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 untimed 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*, *DUT*, and `SceMiClockPort` macros. By convention, the top level netlist module the user supplies to the *infrastructure linker* is called `Bridge` and, for Verilog (see IEEE Std 1364-2001)[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 *untimed* 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. Figure 1 depicts such a configuration, where the Standard Co-Emulation API - Modeling Interface (SCE-MI) is being used as the abstraction bridge. See 3.2 for definitions of the acronyms used here.

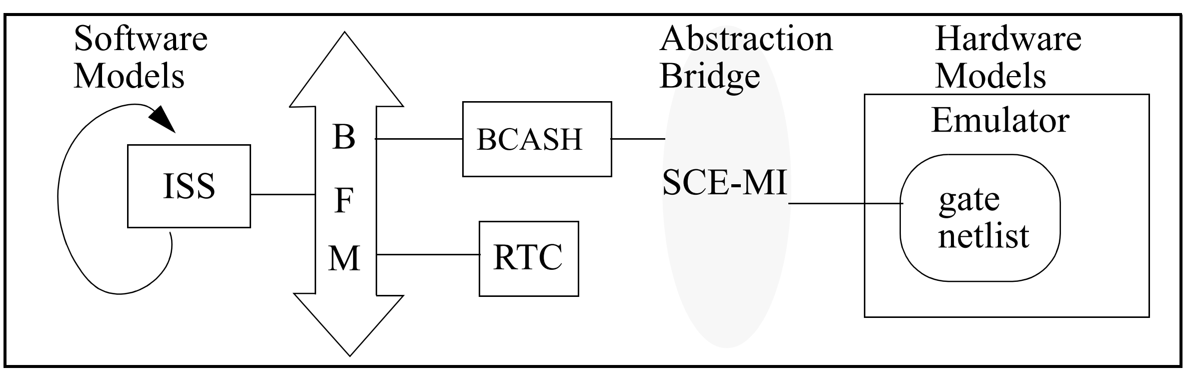


Figure 1—Using the SCE-MI as an abstraction bridge

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

Another illustration can be seen in Figure 4.

3.1.7 **controlled clock (cclock):** The clock 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 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 2 illustrates such a configuration. (See 3.2 for definitions of the acronyms used here.)

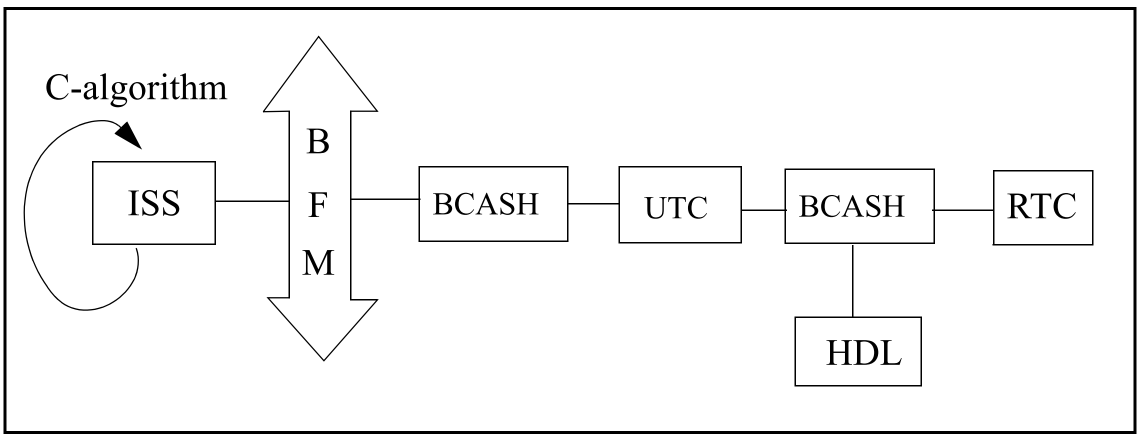


Figure 2—Modeling abstraction gaskets

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 message passing interface, such as SCE-MI.

3.1.10 **cycle stamping:** A process 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 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 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 3.

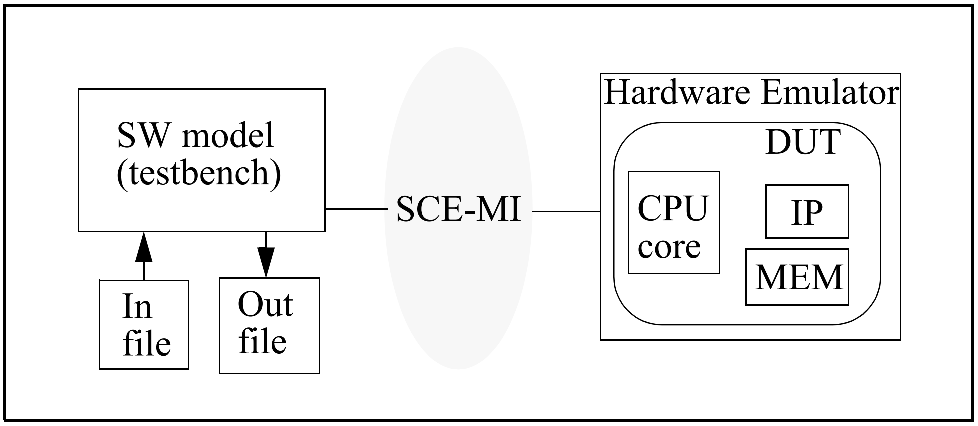


Figure 3—Modeling a DUT via an abstraction bridge

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 untimed. This is accomplished by a translation of untimed 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 untimed communication is handled by *message port proxy* interfaces to the message channels. See Figure 6 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, 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 or other hardware simulation platform). 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 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 vendor to implement the hardware side of the SCE-MI infrastructure, 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.

3.1.24 **posedge:** This refers to the rising edge of a clock.

3.1.25 **service loop:** This function or method call 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 untimed transaction to a series of cycle-accurate clocked events, or, conversely, composes a series of clocked events into a single message. When receiving messages, transactors have the ability to "freeze" *controlled time* long enough to allow message decomposition operations to complete before presenting clocked data to a DUT. And when sending messages, they can freeze controlled time and allow message composition operations to complete before new clocked data is flooded in from a DUT.

3.1.31 **uncontrolled clock (uclock):** A free-running system clock, 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, 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 `uclock`s 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 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 untimed model has no notion of a clock. Rather, its operation is triggered by arriving messages and it can, in turn, trigger operations in other untimed 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
DUT	Device or Design Under Test
EDA	Electronic Design Automation
HDL	Hardware Description Language
ISS	Instruction Set Simulator
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 model

The SCE-MI provides a message-passing environment which connects a model written in HDL to a model running on a workstation. 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. This interface is intended to be used in several different use models and by several different groups of users.

4.1 High-level description

Figure 4 shows a high-level view of how SCE-MI interconnects untimed software models to structural hardware *transactor* and DUT models.

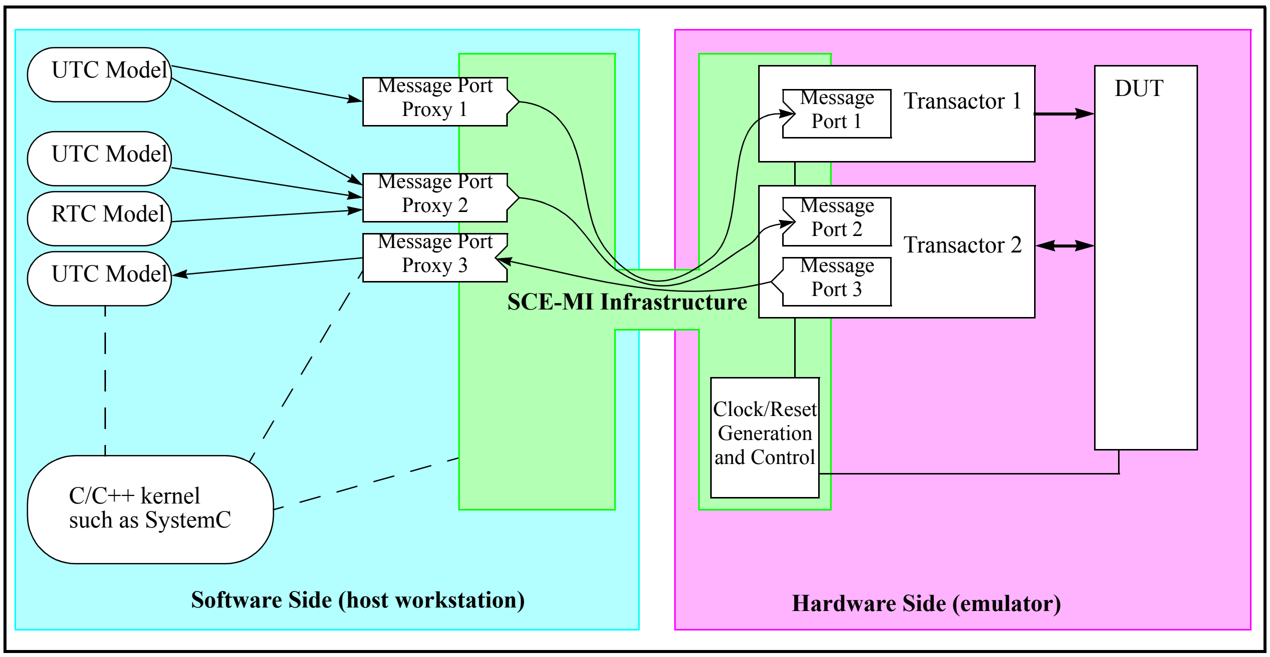


Figure 4—High-level view of run-time components

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 that gives API access to the channel. The end on the hardware side is a *message port macro*, 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. 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 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.

4.2.2 Single-threaded environments

In a single-threaded environment, calls to the service loop function 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 implementor* (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 macros (see 5.2,) except the `SceMiClockPort` macro, whose interface is easy to understand because all it really does is furnish a clock and a reset.

Similarly, on the *software side*, the end-users can also rely on the transactor implementors 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 implementor

The transactor implementor is familiar with the SCE-MI, but is not concerned with its implementation. The transactor implementor 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 implementor can supply *proxy models* on the software side which provide untimed "sockets" to the transactors.

Using the models is like using any other vendor-supplied, 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 implementor is interested in furnishing a working implementation of an SCE-MI that runs on some vendor-supplied 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 5). 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 5 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.

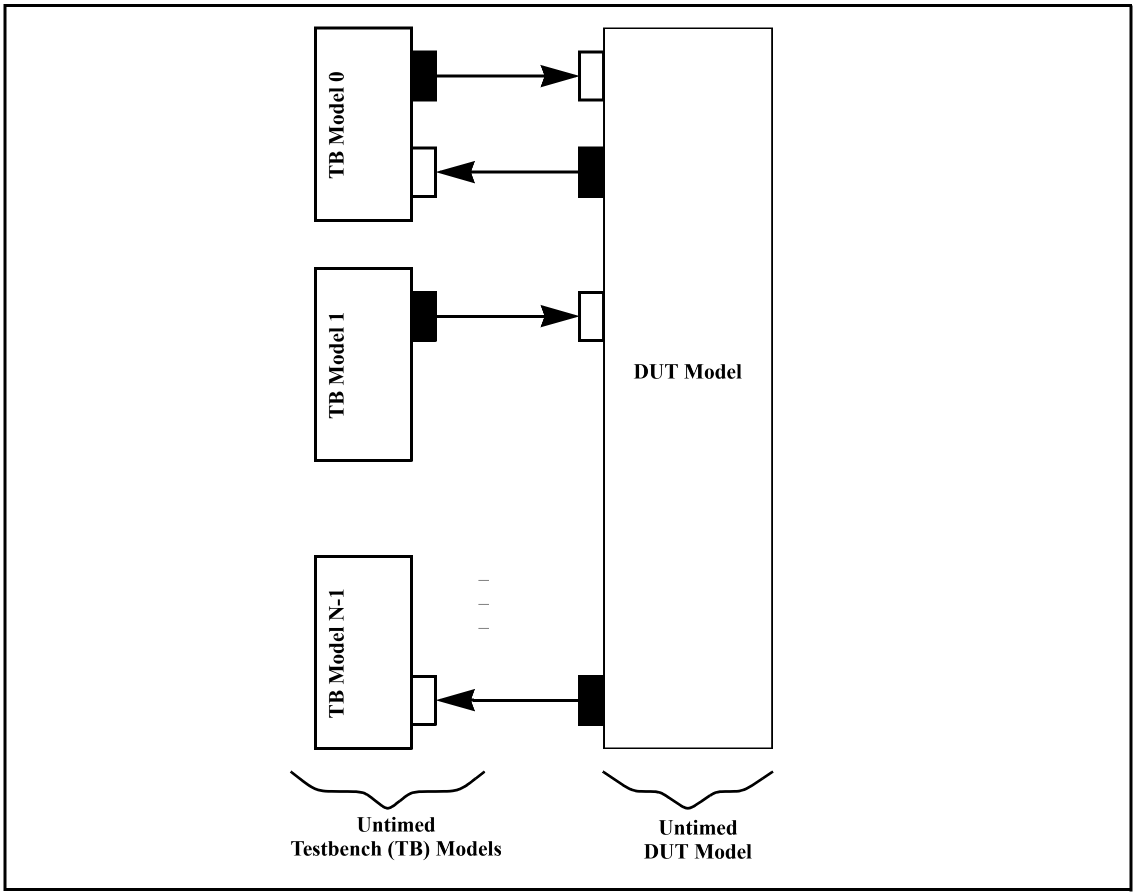


Figure 5—Untimed software testbench and DUT models

4.4.2 Cycle-accurate hardware level modeling abstraction

Take the purely untimed system shown in Figure 5, "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 6 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 5). 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 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 4.3, the *end-user* only has to know how to interface the DUT proxy models on the software side of Figure 6 with the transactor models on the hardware side; whereas, the *transactor implementor* authors the proxy and transactor models using the SCE-MI message port and clock control components between them, and provides those models to the end-user.

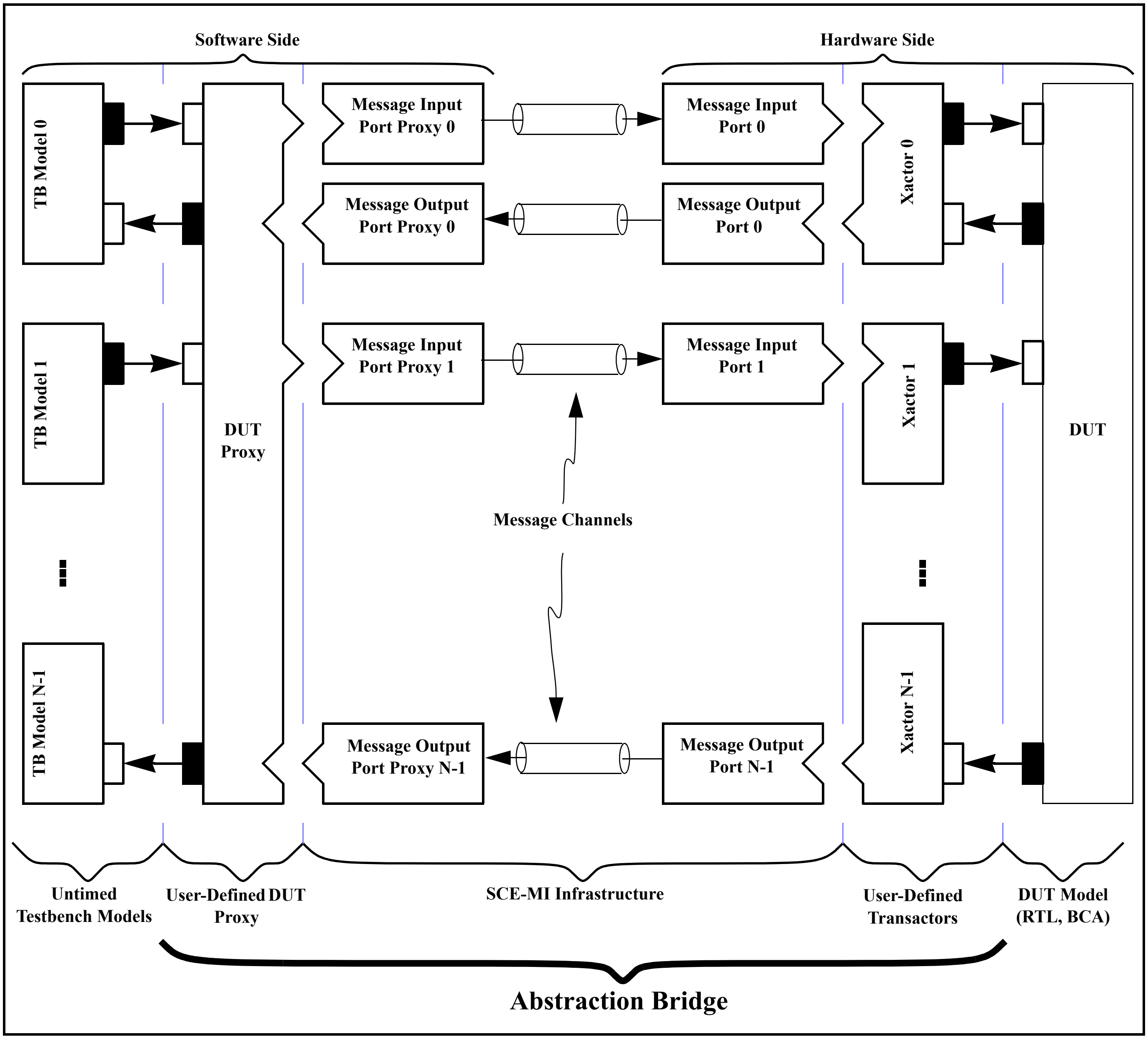


Figure 6— Multi-channel abstraction bridge architecture

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 5 (and, in substantially more detail, in [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 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.

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 7. 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 7. 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 7.

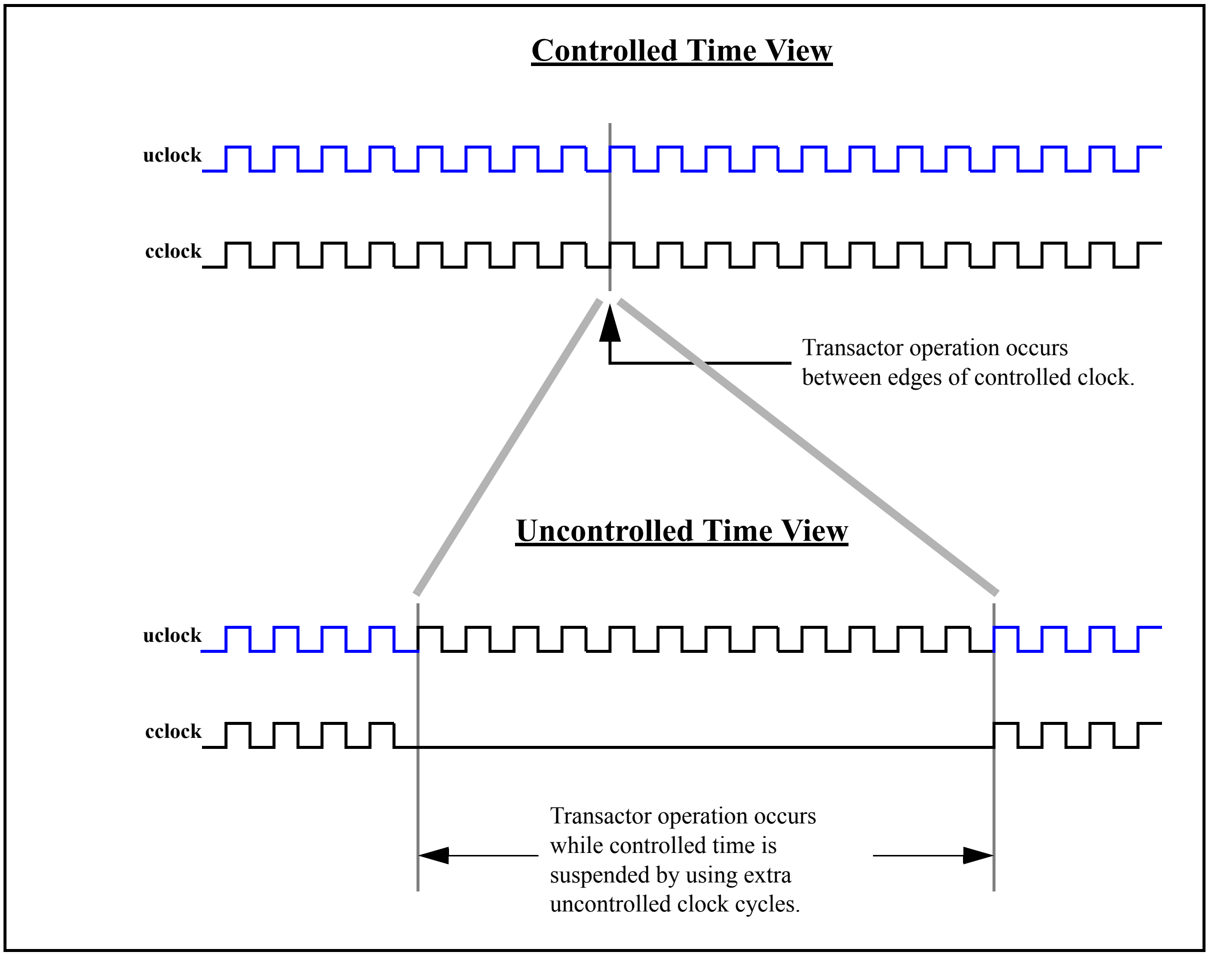


Figure 7—Controlled and uncontrolled time views

In the SCE-MI use model, 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 a 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 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.

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 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 interfaces

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.

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. 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.

5.2 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:
— After a ready request (`TransmitReady` or `ReceiveReady`) has been asserted, it cannot be removed until a data transfer has taken place.
— After `TransmitReady` has been asserted, the data must be held constant otherwise the result is undefined.

The waveforms in Figure 8 depict several dual-ready handshake scenarios.

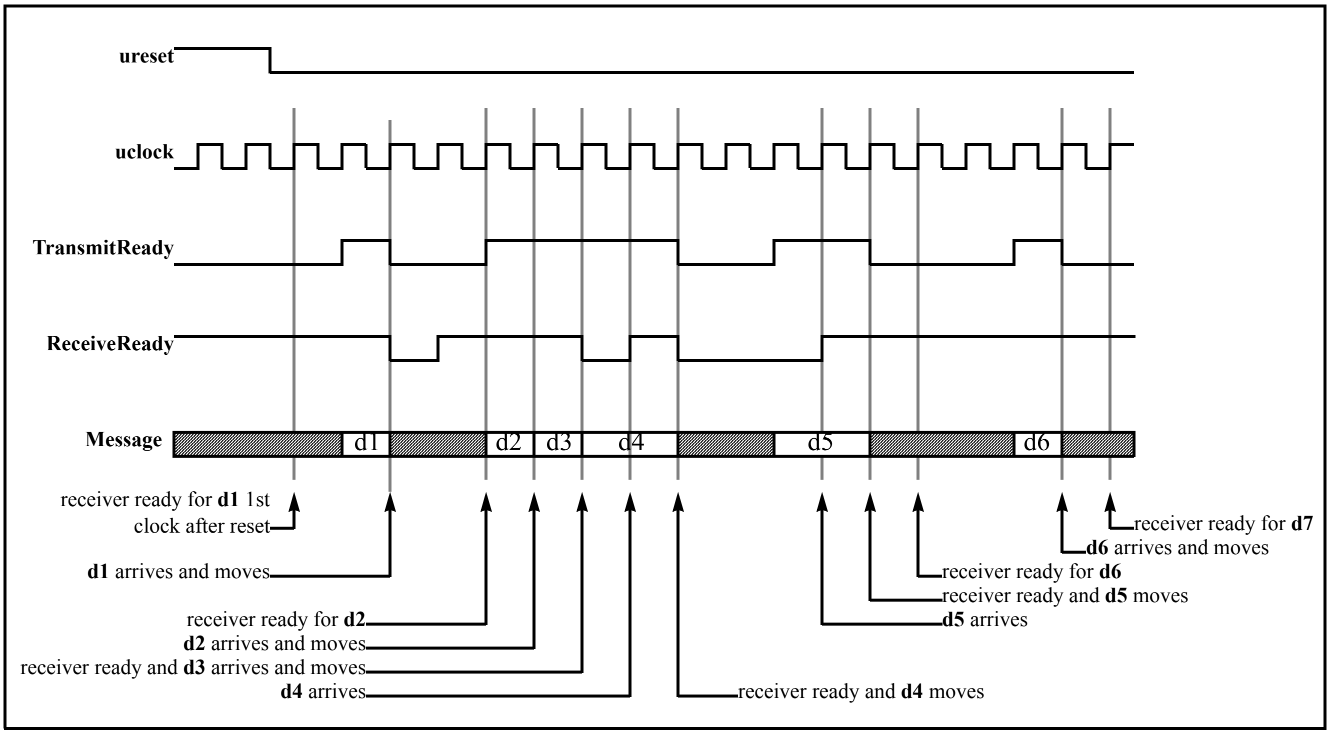


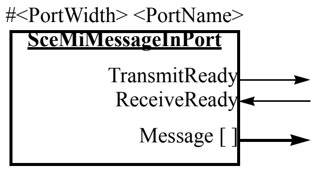
Figure 8—Dual-ready handshake protocol

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 9 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 9—`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 8). 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 `::Service-Loop()`.

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 8. 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 8 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 10 show what typically occurs when input-ready callbacks are enabled. It shows four possible scenarios where an input-ready notification occurs.

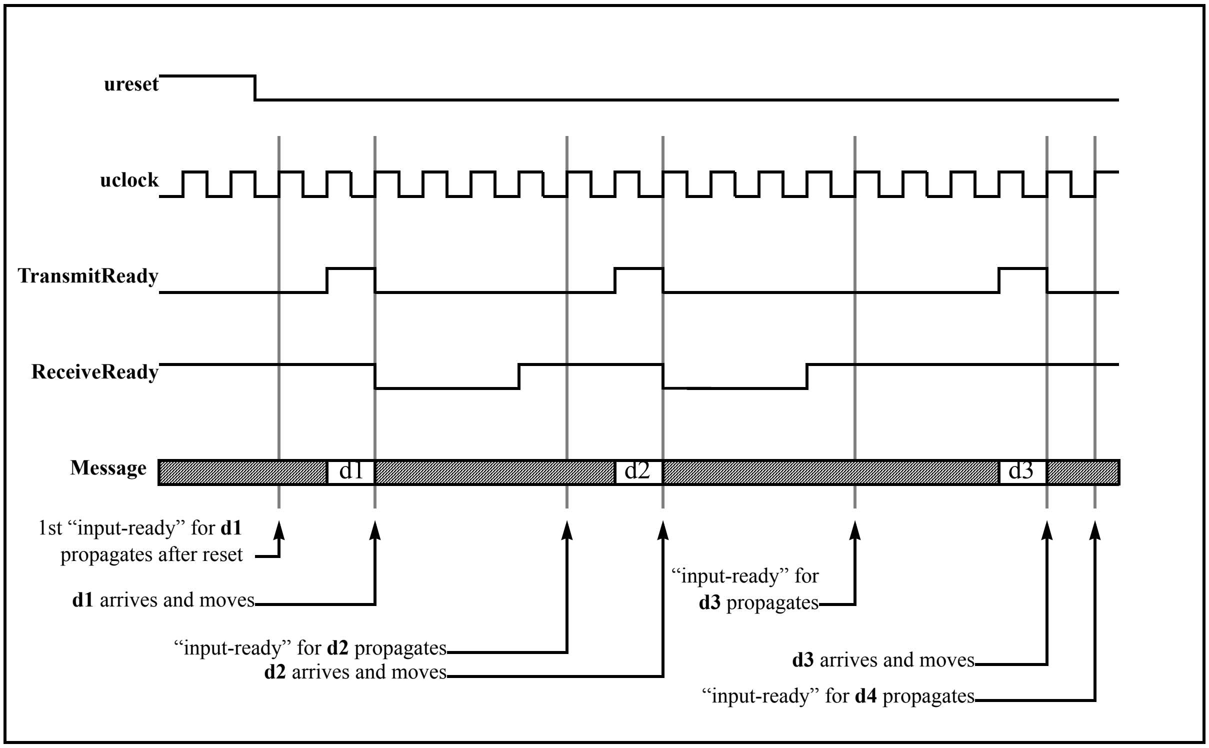


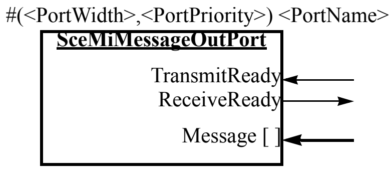
Figure 10—`SceMiMessageInPort` handshake waveforms with input-ready propagation

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 11 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 11—`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:

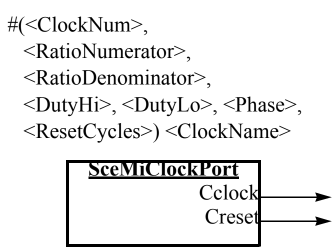
a) Messages from a single message out port are delivered to software in the same time order in which they are delivered to the port.

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 parametrized 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 12 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 12—`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.

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 `SceMiClock-Port` 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 12 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 API 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 12 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 (deasserted) 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 (deasserted) and remains low for the remaining duration of the simulation. Some applications require 2-edged resets at the beginning of a simulation.
— For multiple `cclock`s, 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 quantify shown above to achieve proper alignment of multiple `cclock`s 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 `cclock`s 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 `cclock`s with phases of `0`, this means rising edges of these clocks shall coincide with the point of alignment. For `cclock`s with phases > `0`, those edges occur some time after the point of alignment. Every `cclock` edge must occur on a `uclock` edge.

Figure 13 shows an assortment of cclocks with the uclock and creset. It also shows how those cclocks behave at the point of alignment.

In Figure 13, 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.

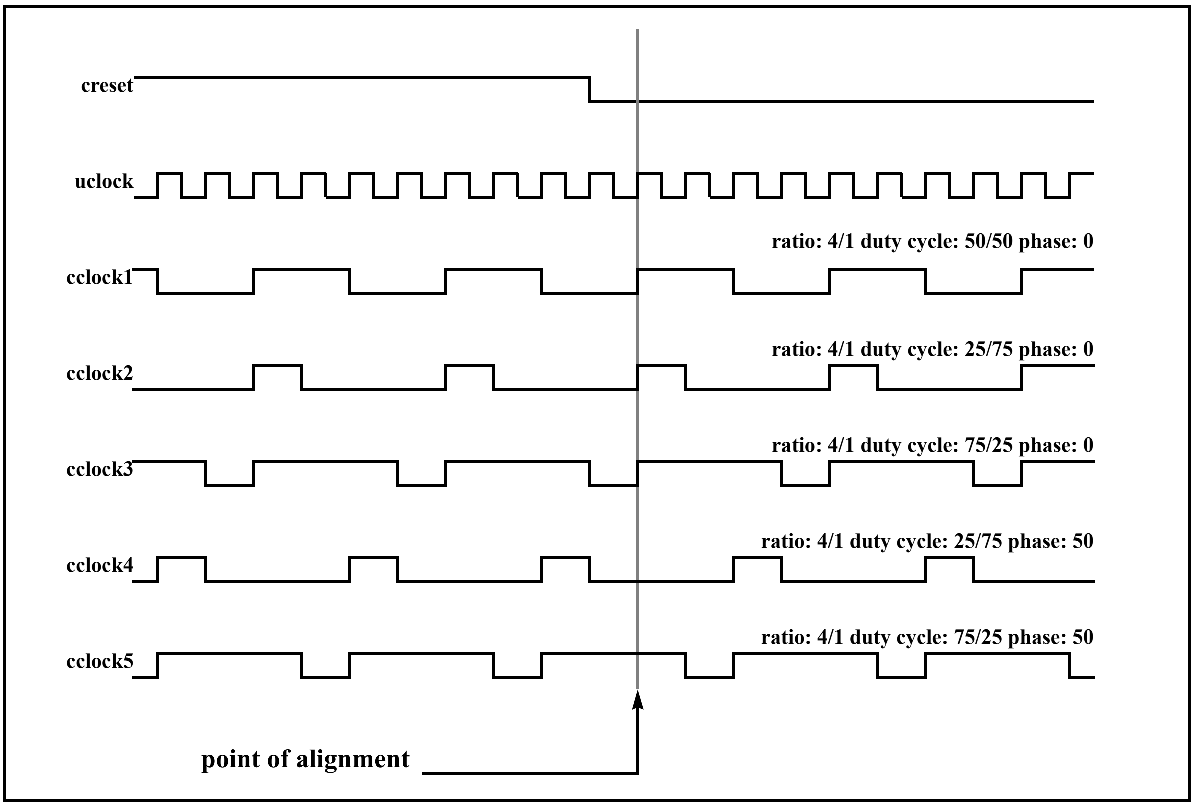
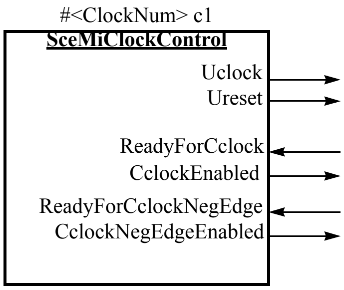


Figure 13—Multi-clock alignment

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 14 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 14—`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 `SceMiClock-Control` 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 `uclock`s 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 `uclock`s. 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 deasserted, 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 deasserted, 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.

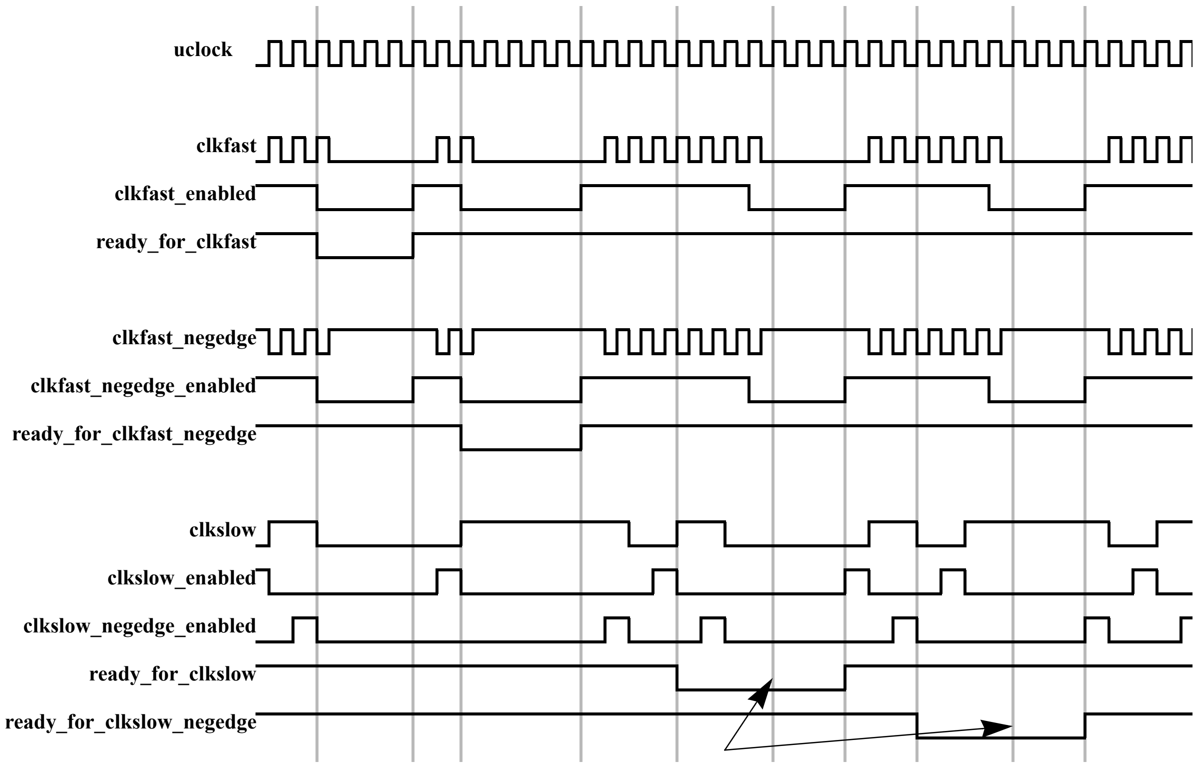


Figure 15— Example of Clock Control Semantics

5.2.5.3 Example of Clock Control Semantics

Figure 15 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.

Deassertion 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.

Deassertion 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.

Deassertion 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.

Deassertion 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.

5.3 Infrastructure linkage

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

As described in 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.

5.3.2 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 implementors of the SCE-MI. The content itself is not intended to be portable.

However, on the software side, the infrastructure implementor 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 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.

5.4.2.1 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.

5.4.2.2 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.

5.4.2.3 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 1
#define SCEMI_MINOR_VERSION 1
#define SCEMI_PATCH_VERSION 0

#define SCEMI_VERSION_STRING "1.1.0"
```

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 adhers 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 deallocate 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 deallocate 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 enqueued 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 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, parametrized 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 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 implementor of the SCE-MI needs to allow the user to specify.

For more details on attribute meanings, see 5.3.1.

Table 1—Minimum set of predefined objects and attributes

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.

Table 1—Minimum set of predefined objects and attributes, *continued*

Object kind	Attribute name	Attribute value type	Meaning
	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.

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 16 shows how the those bits are organized in the data array contained inside the `SceMiMessageData` object.

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

                          31      ...      1, 0   SceMiMessageData word 0
                          63      ...     33,32   SceMiMessageData word 1
                                     71...65,64   SceMiMessageData word 2
```

Figure 16—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.

5.4.5.2 Destructor

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

This destructs the object and frees the data array.

5.4.5.3 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 diagramatically in Figure 17. 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 Multiple `cclock` alignment.

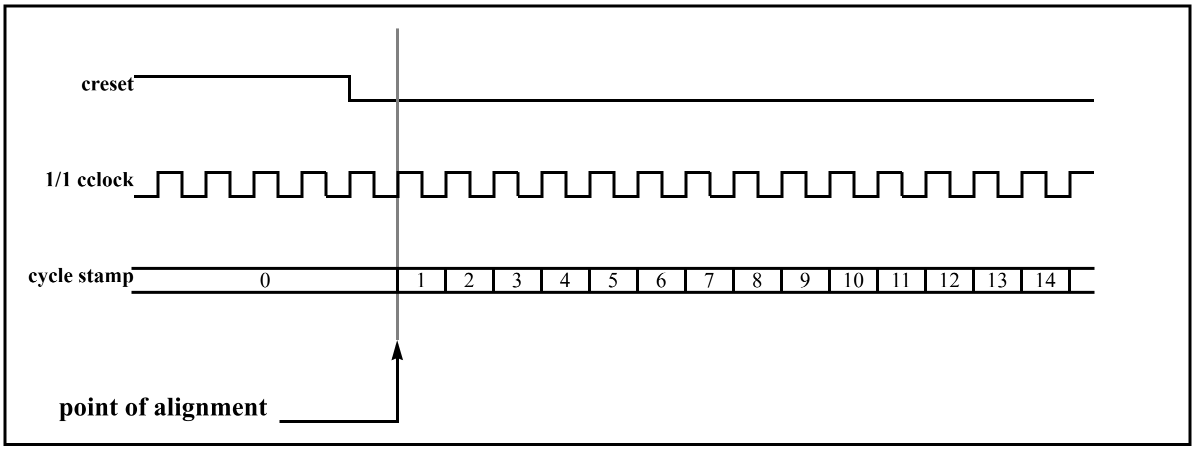


Figure 17—Cycle Stamps

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 5.3.2), 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 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).

5.5.2.1 `SceMiEC` - error handling

```
void
SceMiRegisterErrorHandler(
    SceMiErrorHandler errorHandler,
    void *context );
```

5.5.2.2 `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 );
```

5.5.3.2 Initialization

```
SceMi *
SceMiInit(
    int version,
    const SceMiParameters *parameterObjectHandle,
    SceMiEC *ec );
```

5.5.3.3 SceMi Object Pointer Access

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

5.5.3.4 Shutdown

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

5.5.3.5 Message input port proxy binding

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

5.5.3.6 Message output port proxy binding

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

5.5.3.7 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 );
```

Appendix A

(informative)

Tutorial

A.1 Hardware side interfacing

The hardware side interface of the SCE-MI consists of a set of parametrized macros which can be instantiated inside transactors that are to interact with the SCE-MI infrastructure. The macros are parametrized 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 parametrized 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.

A.1.1 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

A.1.2 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.

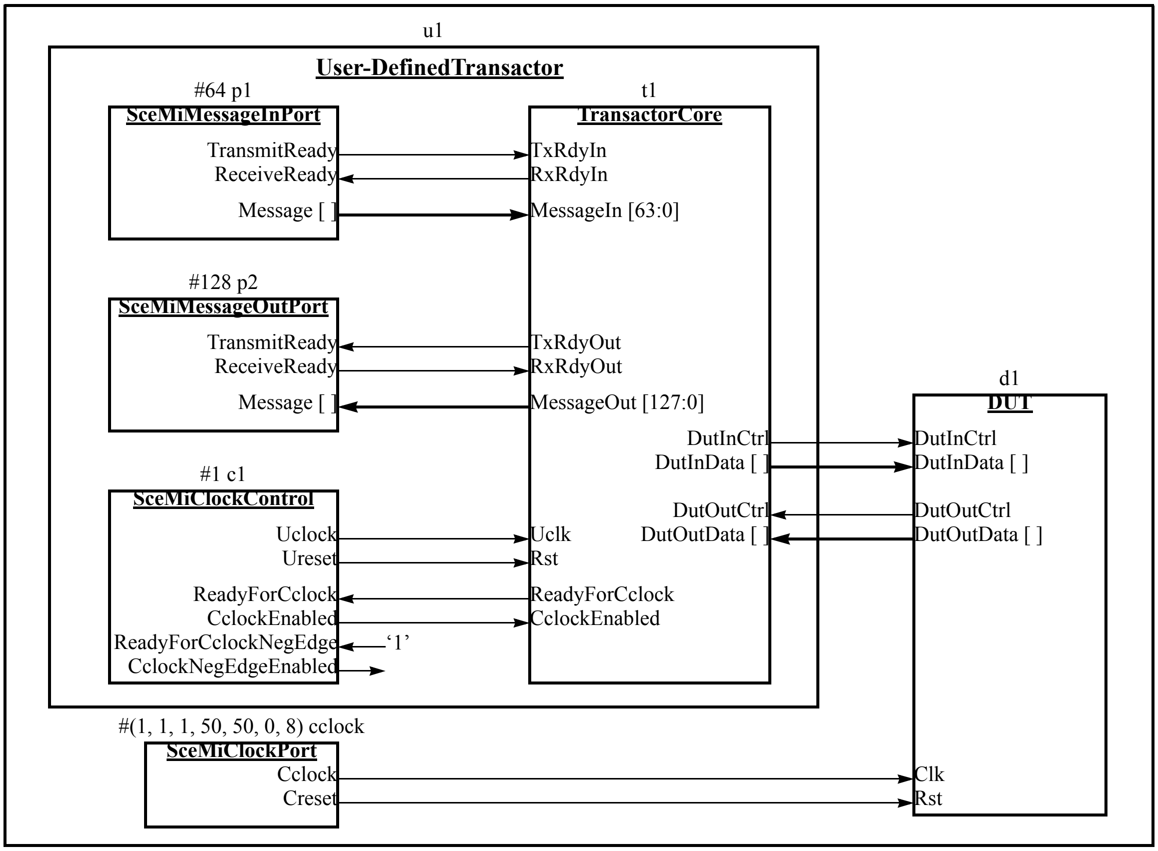


Figure A.1—Connection of SCE-MI macros on hardware side to transactor and DUT

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 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.

A.1.3 `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 parametrized 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 parametrized `PortWidth` parameter.

A.1.4 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 5.2.4.1 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 B.

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 use model 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 [B2] for a description of these concepts.

A.2.1 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.

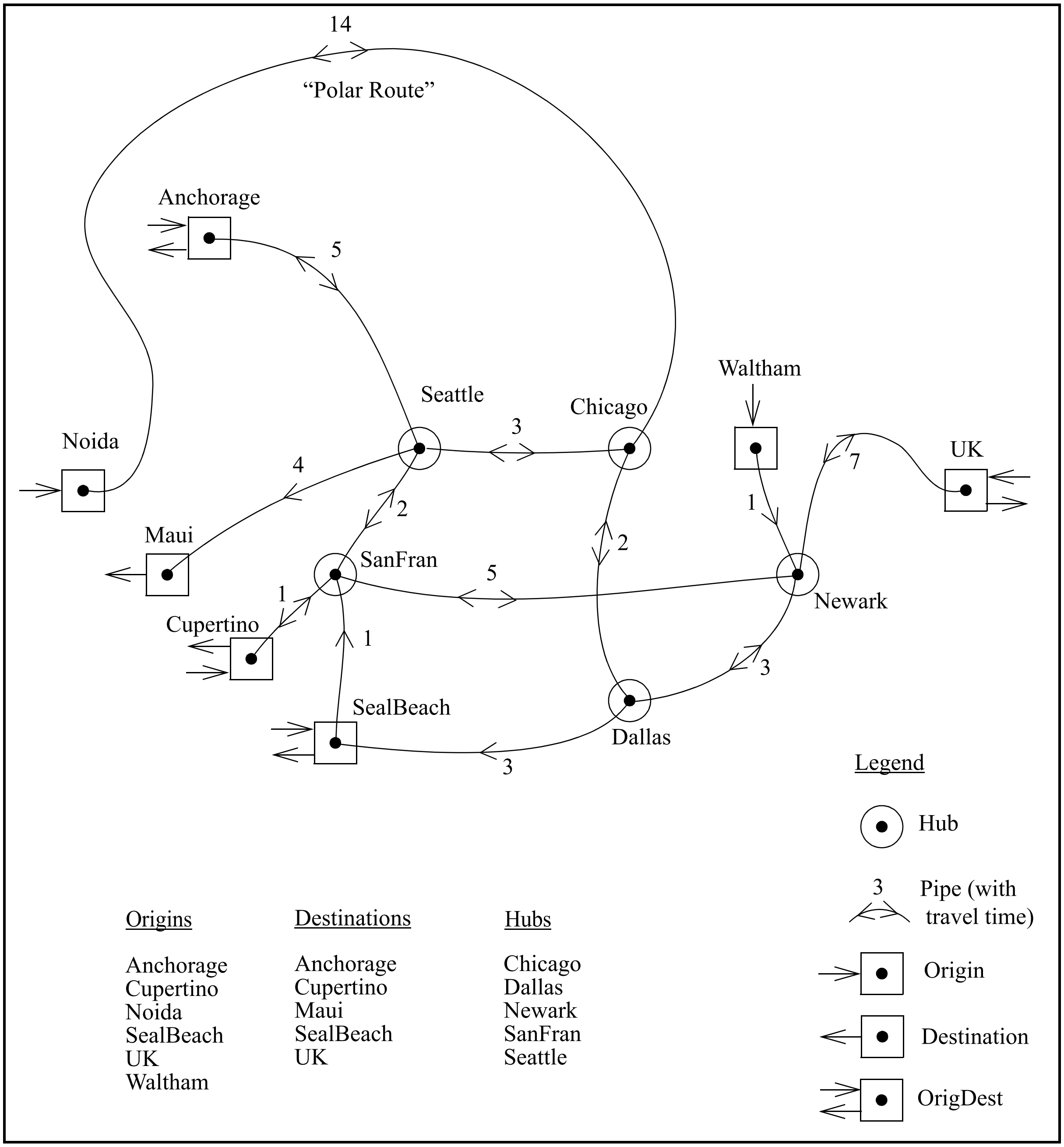


Figure A.2—The `Routed` world

A.2.2 System hierarchy

The hierarchy of the whole system is textually shown in the following subsections.

A.2.2.1 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*.

A.2.2.2 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.

A.2.3 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.

A.2.3.1 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 6. 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.

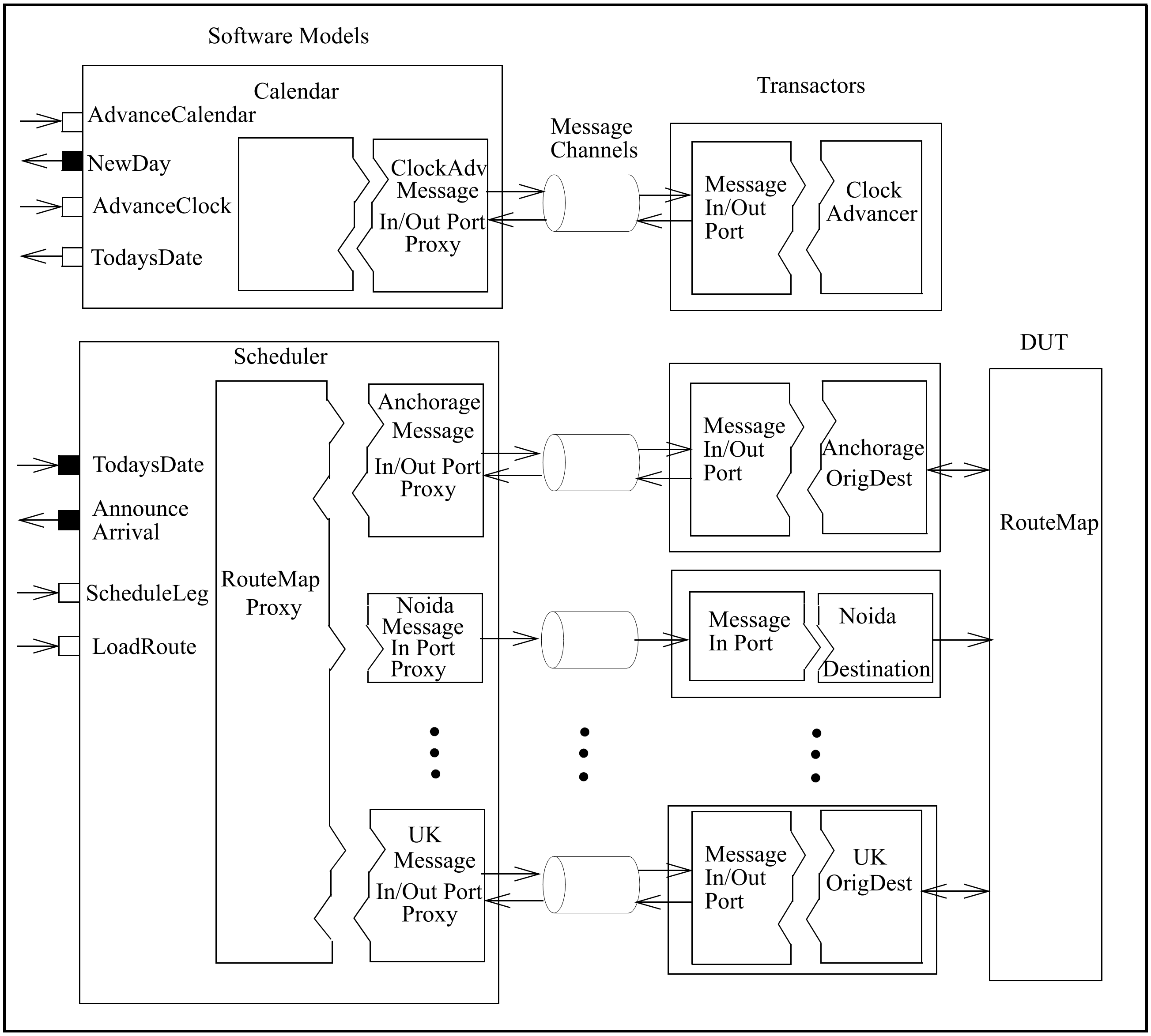


Figure A.3—The bridge

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).

A.2.3.2 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.

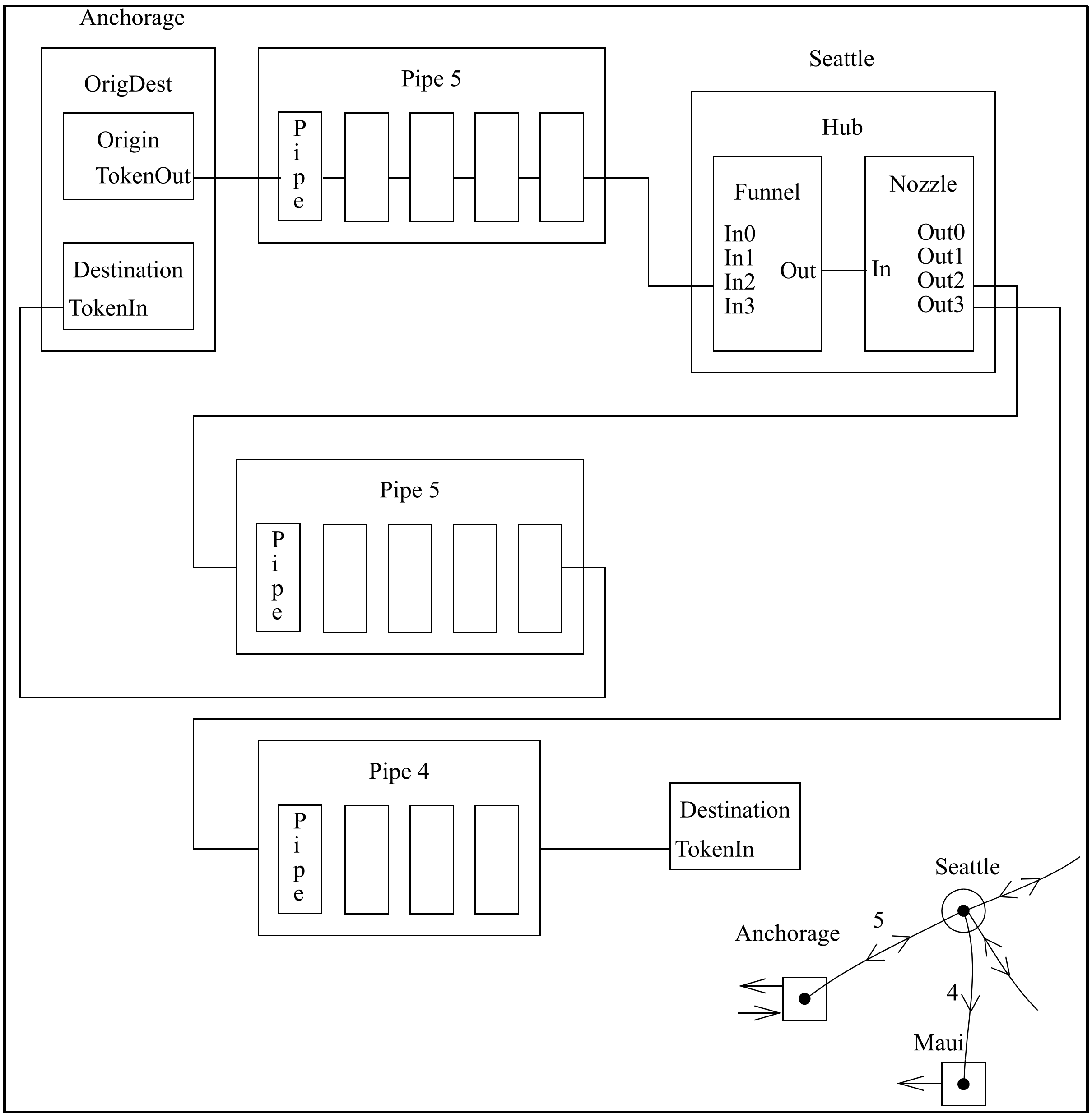


Figure A.4—DUT and transactor interconnect

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.

A.2.3.3 DUT and transactor components

Figure A.5 shows the structure of the DUT and transactor components.

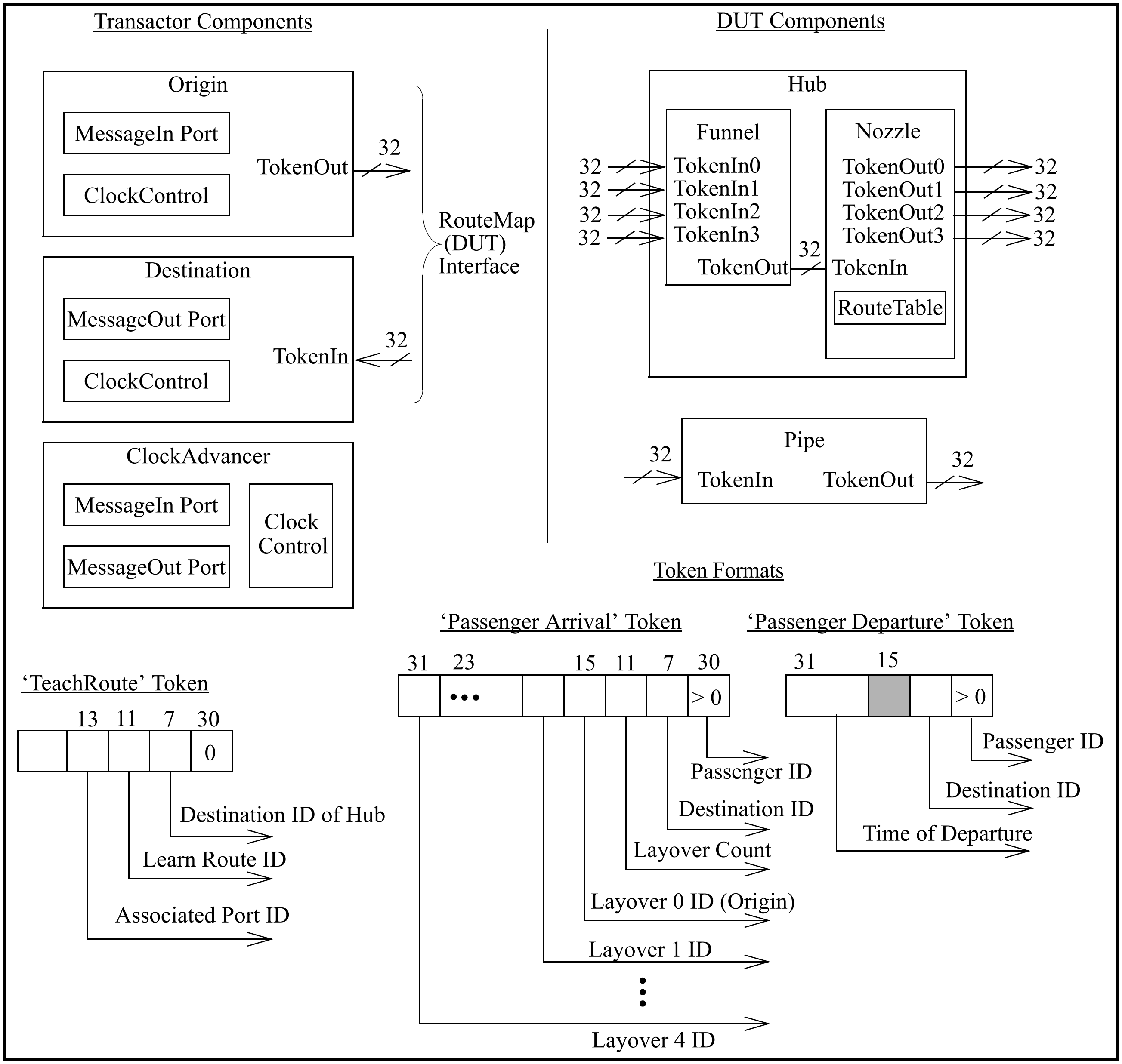


Figure A.5—DUT and transactor components

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 `uclock`s the `cclock` is active, so it can properly count `cclock`s 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 `Hub`s 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.

A.2.3.4 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 offloaded 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 // }
```

A.2.3.5 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 5.4.5.3).

A.2.4 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.

A.2.4.1 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.

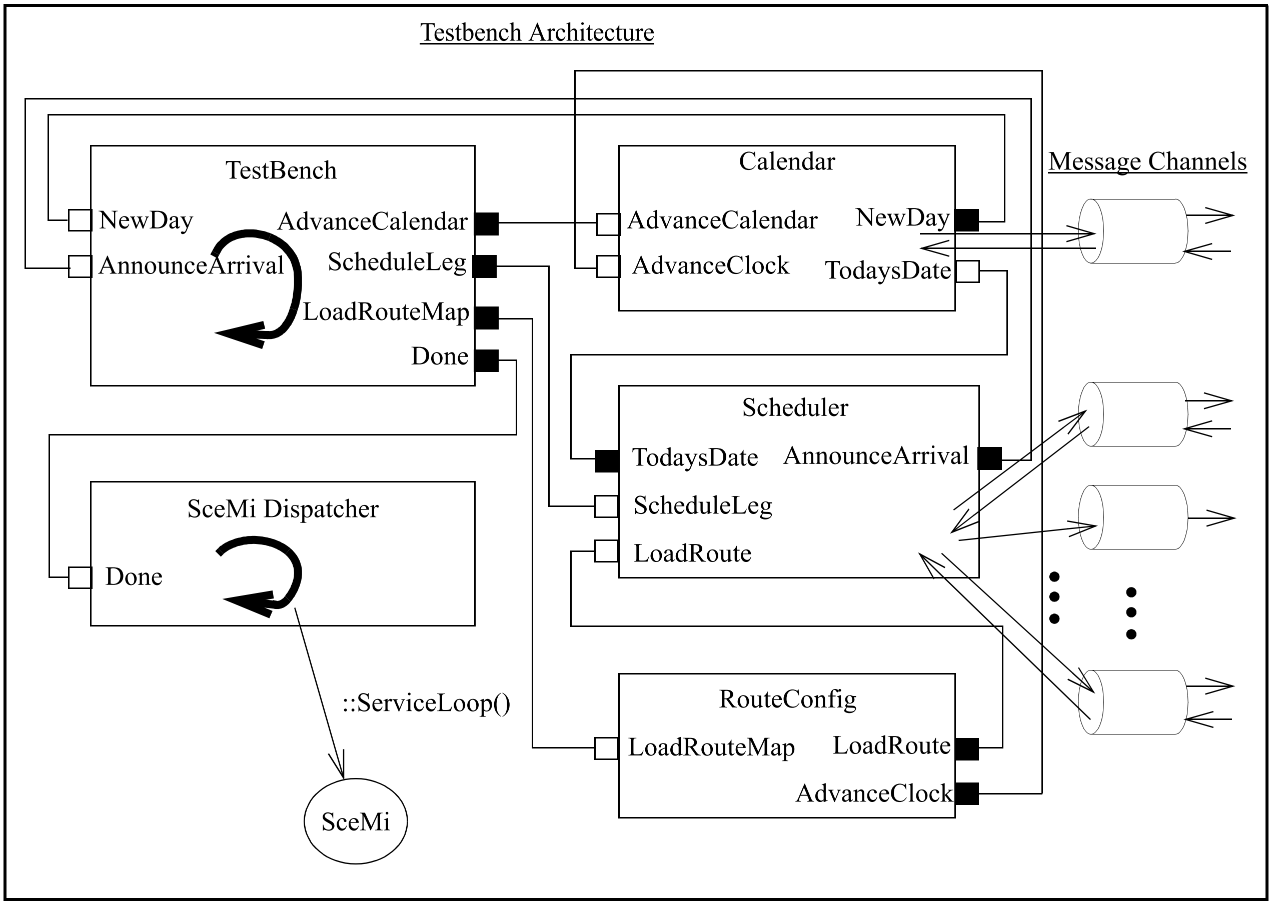


Figure A.6—Interconnect of SystemC models

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 parametrized 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*.

A.2.4.2 The `sc_main()` routine and error handler

The following listing shows the `sc_main()` routine which is the top-level entrypoint 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++ entrypoint 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 an `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).

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 `Test-bench` 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).

A.2.4.4 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;
    };
};
```

A.2.4.5 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;
    }
};
```

A.2.4.5.1 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.

A.2.4.5.2 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.
    }
}
```

A.2.4.6 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.

A.2.4.6.1 `::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);
    }
}
```

A.2.4.6.2 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 `replyCall-`

back() 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) ); }
```

A.2.4.6.3 ::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.
}
```

A.2.4.7 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.

A.2.4.7.1 `::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.

A.2.4.7.2 `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.

Appendix B

(informative)

Multi-clock hardware side interface example

Figure B.1 shows the top level structure of a simple multi-clock, multi-transactor example.

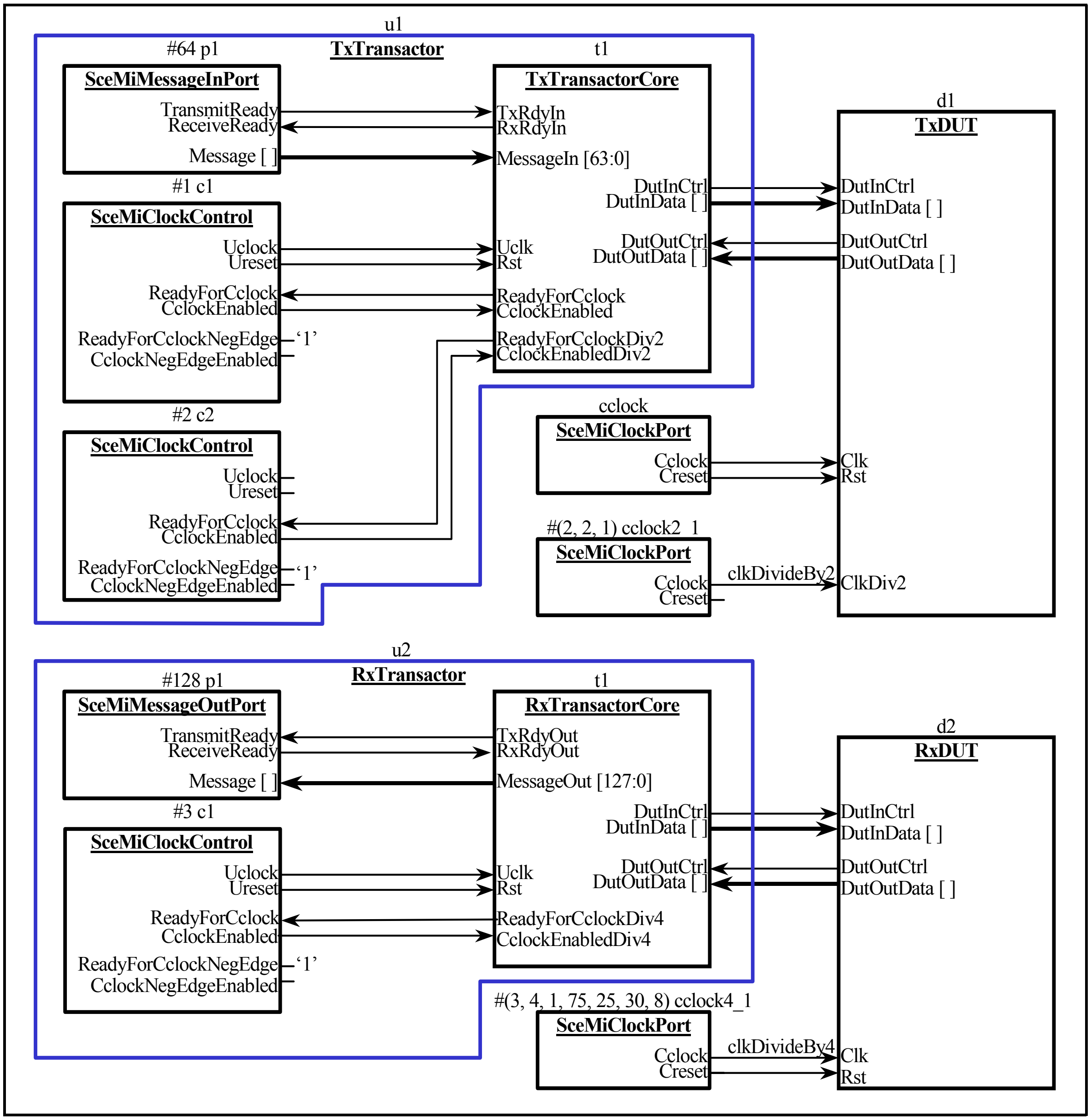


Figure B.1—Multi-clock, multi-transactor example

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 C

(informative)

VHDL SceMiMacros package

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

(informative)

Applying the SCE-MI to event-based systems

The SCE-MI is composed of three primary pieces, all of which are necessary to create a complete communications system between a DUT and a software testbench. In addition, the three pieces affect different 'users' of the standards. These three pieces are

a) *The infrastructure* - This contains the basic communications protocol. It is implemented by an execution engine provider.
b) *The software side API* - This enables to connection the testbench on the software side and is the ultimate end-user of the specification.
c) *Support macros for transactors* - This enables the software communications to be received on the hardware side and makes the information available to the transactors. It also contains macros for controlling the execution engine. These affect the transactor author.

In SCE-MI Version 1.1.0, the only type of execution engines that are directly supported are traditional emulators and rapid prototyping systems which have similar clocking and control requirements. Other execution types will be supported in future releases of this document.

To support this standard using other execution engine types, consider making the following modifications.

```
module SceMiMessageInPort(clk, reset, ReceiveReady, TransmitReady,
                          Message);
parameter PortWidth = 1;
input clk;
input reset; /* can be any user signal to reset the module */
input ReceiveReady;
output TransmitReady;
output [PortWidth-1:0] Message;

module SceMiMessageOutPort(clk, reset, TransmitReady, ReceiveReady,
                           Message);
parameter PortWidth = 1;
input clk;
input reset;
input TransmitReady;
output ReceiveReady;
input [PortWidth-1:0] Messsage;
```

In these two routines, an additional clock signal is passed into the routine. This replaces the current clocking mechanism, which includes the controlled and uncontrolled clock. Making this change means certain other macros become unnecessary, such as `SceMiClockPort()` and `SceMiClockControl()`. In addition, the parameter file, which currently contains the linking information between the transactor models and the clocks, and aids in system reset, serves no useful purpose and can thus be ignored.

If these modifications are incorporated, the essence of the interface is intact and any software testbenches *should* be portable between different execution engine types. However, the transactor models are not intact; these are specific to the actual implementation.

Appendix E

(informative)

Sample Header files for the SCE-MI

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.

C++

```
//
// Copyright © 2003-2005 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 1
#define SCEMI_MINOR_VERSION 1
#define SCEMI_PATCH_VERSION 0
#define SCEMI_VERSION_STRING "1.1.0"

/* 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
```

ANSI-C

```
/*
 * scemi.h
 *
 * Copyright © 2003-2005 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 1
   #define SCEMI_MINOR_VERSION 1
   #define SCEMI_PATCH_VERSION 0
   #define SCEMI_VERSION_STRING "1.1.0"

   /* 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 F

(informative)

Bibliography

[B1] The IEEE Standard Dictionary of Electrical and Electronics Terms, Sixth Edition.

[B2] *SystemC, Version 2.0 User's Guide*, www.systemc.org.