Multiprocessor SPARC™ Architecture Simulator (MPSAS) Programmer’s Guide

THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300

Part Number: 800-6941-03

THIS PUBLICATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND /OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Please
Recycle

Adobe PostScript

Contents

Figures

Tables

Preface

This book, *Multiprocessor SPARC™ Architecture Simulator (MPSAS) Programmer's Guide*, is one of a four-manual documentation set for the microSPARC-II technology. The other three manuals are:

- *Multiprocessor SPARC Architecture Simulator (MPSAS) User's Guide*, which describes how to use MPSAS and its associated programs.
- *microSPARC-IIep Validation Catalog*, which describes a suite of validation tests for the microSPARC-IIep technology
- *microSPARC-IIep Megacell Reference*, which explains how to build microSPARC-IIep megacells

Organization of This Book

This book is intended for programmers who need to create or modify modules to simulate a SPARC-based computer system. It contains the following chapters and appendixes.

Chapter 1, *Overview*, describes MPSAS from the programmer's point of view. It introduces a number of concepts used in the rest of the manual.

Chapter 2, *Framework*, discusses the services provided by the framework.

Chapter 3, *Writing a Module*, discusses how to use the services provided by the framework to write a module.

Chapter 4, *Module Entry Points*, describes the entry points (routines) a module provides for its configuration, user interface, and simulation, as well as the framework routines they call.

Chapter 5, *Miscellaneous Framework Routines*, describes the framework routines used by modules.

Chapter 6, *Product Structure*, describes the source structure layout, its makefiles, and layers. It describes how to add message types and user interface commands to each layer.

Chapter 7, *Asynchronous Input (sigio)*, describes the asynchronous UNIX input facility (called `sigio`) and how modules can use it.

Chapter 8, *Access Classes*, describes how to write or modify an access class.

Appendix A, *Framework Manual Pages*, provides detailed descriptions of all framework routines that may be called by a module.

Appendix B, *Example Module Listing*, contains a source listing of the `example` module class.

Appendix C, *Services Provided by the Layers*, describes the message types and utility routines provided in layers above the framework layer.

At the end of the book is an index.

Prerequisite Knowledge

It is assumed that you are familiar with programming in the C language in the UNIX® environment and that you have a basic familiarity with computer architecture, in particular the SPARC architecture. For further information, see the list of documents in the following section.

Related Books and References

The following documents contain material that further explains or clarifies information presented in this guide.

The SPARC Architecture Manual/Version 8 by David Weaver, Prentice Hall; ISBN: 0138250014

The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall; ISBN: 0131103628

The Annotated C++ Reference Manual by Margaret Ellis and Bjarne Stroustrup

Typographic Conventions

TABLE P-1 describes the typographic conventions used in this book.

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, instructions, files, and directories; on-screen computer output; email addresses; URLs	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, section titles in cross-references, new words or terms, or emphasized words	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
<>	A bit number or colon-separated range of bit numbers within a field; bit 0 is the least significant bit.	`WB_VECTOR`<15:0>

Sun Documents

The SunDocs^{SM} program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals by using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

Sun Documentation Online

The `docs.sun.com` Web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com/`.

Disclaimer

The information in this manual is subject to change and will be revised from time to time. For up-to-date information, contact your Sun representative.

CHAPTER **1**

Overview

The MPSAS behavioral simulator is a software tool, written in C++ and SPARC assembly language, with which you can model SPARC-based computer systems at the instruction or transaction level. The system being simulated is independent of the computer on which the simulation is running. When run on a SPARCstation™, MPSAS typically simulates thousands of SPARC instructions per second.

In MPSAS, a computer system is modeled by a group of autonomous units called *modules*. Each module contains code and state to model the behavior of a hardware component. Typically, each module's state is private, although portions of it can be shared with other modules. Modules communicate with each other by passing messages and through shared state. A module updates its state based on messages it receives and as time elapses.

FIGURE 1-1 shows an abstract view of a basic computer system. The rectangles are modules; the lines between them represent their communication paths. The system depicted has two processors, each with a memory-management unit and a cache. These two processors share a serial port, disk controller, RAM, and ROM. The bus arbitrates between the requests of the two processors.

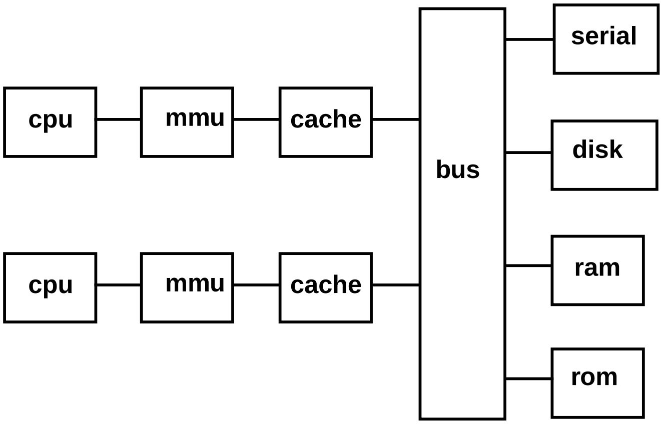

FIGURE 1-1 Abstract View of a Basic Computer System

MPSAS measures time in cycles. Modules use cycles to synchronize with each other (just as hardware uses a system clock for synchronization) and to divide their work into time-dependent pieces. A module can specify that a message is to be delivered some number of cycles in the future. This delay feature is useful in simulating the response time of a device or communication delays.

1.1 Simulation Levels

A module can simulate the behavior of its hardware at different levels of detail. As the level of detail increases, the following situations result:

- A module more accurately simulates all of the behaviors of its hardware.
- The performance of the simulator decreases.
- The effort to develop a module increases.

Modules of different levels of detail can be mixed in a system to provide the required level of accuracy and speed.

Two common levels of simulation detail for a module are called *instruction accurate* and *transaction accurate*.

1.1.1 Instruction-Accurate Module

An instruction-accurate module simulates its hardware at a low level of detail. It correctly simulates the functions of its hardware, but its timing and transactions might not be correct.

The structure of a system modeled by instruction-accurate modules can be quite different from the structure of the system hardware (due to the simulation's level of abstraction).

The order in which instructions are executed by the processors in the system might differ from the order in which they would be executed on actual hardware. Quite often these inaccuracies are acceptable.

1.1.2 Transaction-Accurate Module

A transaction-accurate module accurately simulates the transactions involving the module and the module's state. It is more detailed than an instruction-accurate module.

The structure of a system modeled by transaction-accurate modules is similar to the structure of the system hardware.

1.2 Modularization Criteria

In MPSAS, a computer system is modeled as a collection of modules. Thus, the functions of the system must be decomposed into a set of modules. Various trade-offs are involved in decomposing the system.

The functions of a module should be as cohesive as possible. Typically, there is a large degree of sharing of data and communications between them.

The degree of interdependence between modules should be minimized to decrease the overhead of message passing. Sharing of state can reduce the message-passing overhead.

Typically, a computer system has a hardware block diagram that shows the integrated circuits and how they are connected. Hardware designers follow a set of modularization criteria similar to those followed by the module programmer, so their block diagram is a reasonable starting point for decomposing the system into modules.

The blocks can be decomposed into several MPSAS modules, or some of the blocks can be combined into one module because hardware constraints differ from simulation constraints. Each bus in the hardware block diagram that has more than one reader should be modeled by a module.

The negative characteristics of a too coarsely decomposed system are:

- Reduction of runtime configurability
- Reduction of reuse of modules
- Increased complexity of debugging the system simulation—larger modules are more complex
- Reduced opportunities for parallel development by multiple programmers

The negative characteristics of a too finely decomposed system are:

- Increase in number of required modules states
- Degraded performance because of excess message passing

CHAPTER 2

Framework

In addition to the code implementing the modules, MPSAS contains code to implement the framework. The framework performs several key functions:

- Supports module classes and module instances
- Controls the configuration of the modules
- Implements the message-passing facility
- Implements the cycle paradigm
- Implements the concept of layers
- Provides support functions for the modules.

Think of the framework as surrounding the modules and providing the environment in which the modules exist. This idea is analogous to the UNIX kernel providing an environment in which applications exist. This book describes the environment created by the framework for modules; it does not discuss the internal data structures and detailed operations of the framework.

The operation of the framework proceeds in two distinct phases: configuration and simulation.

Soon after the simulator is started, the framework enters the configuration phase. The configuration phase parses a configuration file and then initializes the modules specified in it.

The simulation phase starts immediately after the configuration phase finishes. During the simulation phase, the simulation is said to be *running* if activities related to simulating the computer system are occurring (that is, the message-passing facility is active and cycles are advancing). The user can start and stop the simulation at will, and the modules and framework can stop the simulation as well (for example, at breakpoints).

2.1 Module Classes and Module Instances

The framework supports module classes and module instances. To the framework, a module class is a set of entry points (pointers to C routines). These entry points are called module entry points. A module instance is associated with a module class but it also includes a state. This document uses the term *module* by itself in many cases where the distinction between class and instance is unimportant or obvious.

A module class is analogous to a C data type; it specifies the behavior of an object. A module instance is analogous to a C variable; it is an instance of some C data type and has state associated with it.

The framework can support an unlimited number of module classes and module instances. Increasing the number of module instances in a simulation does not increase the framework's overhead (that is, the framework imposes no speed penalty for a large number of module instances).

The framework deals with module classes and module instances during the configuration phase. During the simulation phase, the framework only deals with the module instances.

The framework controls which module executes at any time. It calls a module entry point to have it perform some function related to that module's configuration or simulation. The module entry point can call routines in the framework (called framework routines) to help it complete its task, but eventually the module entry point returns control to the framework.

2.2 Module Configuration

The configuration of the modules in the system is divided into six steps:

1. Initializing the module classes
2. Initializing each module instance of each module class
3. Initializing each interface of each module instance
4. Create each shared object of each module instance
5. Looking up each shared object used by each module instance
6. Verifying the final configuration of each module instance

The framework performs each configuration step for all modules before it proceeds to the next step. For example, all module classes are initialized, and then all module instances of all module classes are initialized.

Each module must support the first step of configuration; the other steps are optional, but most modules support some of them.

Each module provides a routine to perform each step it supports. These routines are called *module configuration entry points*.

The framework calls the module configuration entry points to aid in the configuration of the system specified by the configuration file. Each entry point can have several parameters associated with it; some parameters are passed to the entry point when the framework calls it; others are obtained by the entry point calling an appropriate framework routine.

The module configuration entry points pass information to the framework by a combination of the entry point's return code and calls to appropriate framework routines.

FIGURE 2-1 shows how the framework and module configuration code interact during the configuration phase. The *modname* prefix in the module entry point names represents the name of some arbitrary module class. Note that only module classes that have instances defined in the configuration file are involved in the configuration process.

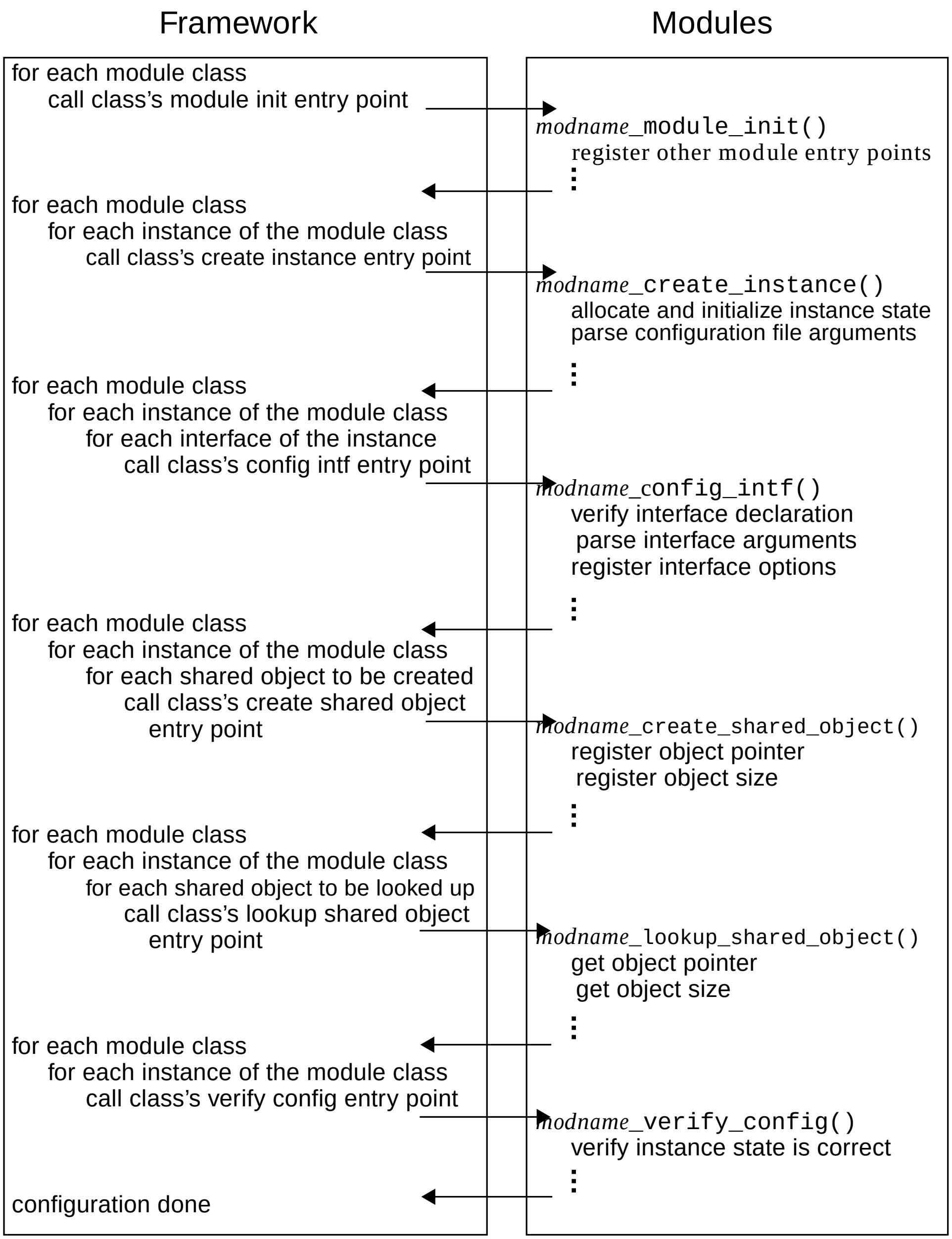
Framework
Modules
for each module class
call class's module init entry point
*modname*_module_init()
register other module entry points
for each module class
for each instance of the module class
call class's create instance entry point
*modname*_create_instance()
allocate and initialize instance state
parse configuration file arguments
for each module class
for each instance of the module class
for each interface of the instance
call class's config intf entry point
*modname*_config_intf()
verify interface declaration
parse interface arguments
register interface options
for each module class
for each instance of the module class
for each shared object to be created
call class's create shared object
entry point
*modname*_create_shared_object()
register object pointer
register object size
for each module class
for each instance of the module class
for each shared object to be looked up
call class's lookup shared object
entry point
*modname*_lookup_shared_object()
get object pointer
get object size
for each module class
for each instance of the module class
call class's verify config entry point
*modname*_verify_config()
verify instance state is correct
configuration done

FIGURE 2-1 Configuration Phase

2.3 Message Passing

Modules communicate by sending messages to each other much like two integrated circuits communicate, where the source chip drives its output pins and the destination chip samples its pins some time later. (Of course, the pins of the two chips must be connected by wires to enable the chips to communicate.)

Instead of pins, modules have interfaces. An interface provides a module with access to the framework message-passing facility like a pin provides the die of a chip access to the printed circuit board wires. Like a pin, each interface belongs to a single module instance.

There are some important differences between interfaces and pins.

- Interfaces can transmit messages which may contain many bits. Thus, each interface may represent a group of pins (for example, the bus interface of a chip).
- Pins can be connected in a configuration that includes multiple receivers (for example, a snoopy bus). Interfaces are typically connected in a one-to-one configuration. They can be connected in a many-to-one configuration, provided the *many* interfaces only send messages and the *one* interface only receives messages. Other interface configurations are not allowed.

A message sent by a module to an interface it owns is received by the module connected to the remote interface. One-to-one interfaces are bidirectional. The interface that originated the message is called the *source interface* or the *local interface*; the interface that receives the message is called the *destination interface* or the *remote interface*.

The framework allows each module instance to have any number of interfaces. The framework assigns a unique 32-bit opaque value (called the *interface handle*) to each interface during the configuration phase. The module can store this handle for later operations involving the interface.

During configuration, the module can examine the interface type and the interface arguments of each of its interfaces specified in the configuration file. From those two sources, it determines how each interface is to behave.

For example, an MMU module may require two types of interfaces: one to connect to the processor and one to connect to the memory system. In this case, a different type could be assigned to each interface, and the MMU would examine the interface type of each interface to identify it.

Topics related to message passing are:

- Channels
- Message composition
- Interface operations

2.3.1 Channels

The framework message-passing facility consists of three independent channels: positive-phase simulation, negative-phase simulation, and debug. Interfaces can support message passing on all channels.

Modules use the simulation channels to communicate information relevant to the simulation. For example, a processor may send a message on a simulation channel to fetch an instruction from the memory subsystem. The positive-phase simulation channel is used to send a message to an interface during the positive (first) half of a cycle. The negative-phase simulation channel is used to send a message to an interface during the negative (last) half of a cycle.

Modules use the debug channel to communicate information on behalf of the simulator user. For example, the user enters a command that causes the processor to send a message on the debug channel to fetch an instruction from the memory subsystem and then display it. Debug channel accesses should not change the simulation state unless the user specifically asks the simulator to change the simulation state. (For example, if a user requests an instruction in memory to be disassembled, the simulator should not mark the cache as referenced. If the user asks to write to an address in memory, the state of the cache may have to be changed.)

Simulation channel messages are only delivered by the framework when the simulation is running. However, debug channel messages are always delivered by the framework. So, users can stop the simulation and still have the modules communicate on their behalf.

Each simulation channel message has a cycle delay associated with it. Depending on the characteristics of the destination interface, the framework may deliver the message immediately (ignoring the delay) or may delay the transmission of the message by the specified number of cycles. All debug channel messages have zero delay.

The framework has a message queue for each channel. The simulation channel message queues accept messages with an arbitrary positive delay. A delay of zero is not allowed in certain situations (described later). The debug channel message queue accepts only messages with zero delay.

2.3.2 Message Composition

A message is composed of the following elements:

- Size — Specifies the size of the message data in bytes.
- Data pointer — Points to a dynamically allocated contiguous block of memory of “size” bytes if the message size is non-zero; otherwise the data pointer can be any value. In the first case, a message pass transfers the ownership of the data from

the sender to the receiver. The receiver may free the data or reuse it. The sender must not access the data even though it may still have a pointer to the data.

- Message type — Specifies how the message data is to be interpreted. Each message type has a unique opaque pointer associated with it. Message types must be registered with the framework. If the format of the data for a message type is described to the framework, the simulator user can observe the fields of the message when it is sent between interfaces.

2.3.3 Interface Operations

Three message-passing operations are associated with an interface: sending messages, receiving messages, and queuing messages. Each of these operations can occur over the simulation or debug channels.

FIGURE 2-2 shows a representation of how the three interface operations interact with the framework message-passing facility. The rectangles are module instances, the rounded rectangles are interfaces, and the ladderlike objects are queues. In the figure, interface *a* of module instance A is connected to interface *b* of module instance B.

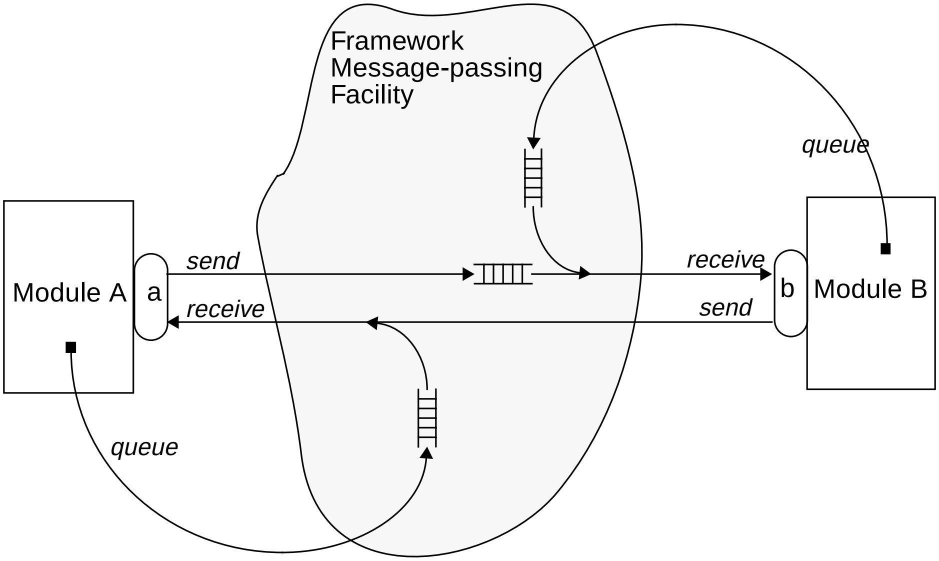

FIGURE 2-2 Interface Operations

FIGURE 2-2 shows only one of the three message channels; the diagram applies equally to all of them.

The send Operation

The send operation transfers a message from a source interface to a destination interface. Framework routines perform a send operation on the simulation or debug channels. The interface handle specifies the source interface. The message is received by the destination interface.

Note that an interface can be connected to itself. When a message is sent to such an interface, it is received by the same interface (which is both source and destination).

The receive Operation

Unlike some message-passing systems, a module does not call a routine to receive a message and become blocked until one arrives. Instead, a module registers a routine that is called by the framework when a message arrives. This routine is known as a *receive entry point*. The act of the framework calling an interface's receive entry point is called *delivering* the message.

A module can register a different receive entry point for each of its interfaces for each of the three message channels. The interface handle on which a message was received is passed to the receive entry point. One receive entry point can use this information to service multiple interfaces. A receive entry point can determine if it was called to receive a simulation or debug channel message, but it cannot determine if it received a positive or negative phase simulation channel message.

Interface Receive Modes

Mode specifies an interface's behavior for messages it receives from another interface; it does not affect how messages are sent by the interface. There are two interface receive modes: queued and immediate.

When a module sends a message and the destination interface is in queued mode, the message is put in a framework message queue and the sending module continues execution. The message waits in the message queue for a number of cycles equal to its delay value and then is delivered. In FIGURE 2-2 on page 11, interface *b* is in queued mode because a queue exists between the source interface *a* and the destination interface *b*.

When a module sends a message and the destination interface is in immediate mode, the message is delivered immediately (that is, the message bypasses the message queue). The sending module does not continue execution until the destination interface's receive entry point returns. The delay value of the message is ignored, but its value is passed to the receive entry point. In FIGURE 2-2, interface *a* is in immediate mode because no queue exists between the source interface *b* and the destination interface *a*.

Note that the sending module does not know if the destination interface is in immediate or queued mode.

Immediate interfaces are faster than queued interfaces because there is less overhead in a message transfer. However, they are typically only used in instruction-accurate modules since the framework does not honor the message delays of send operations.

Sending a message to an immediate interface amounts to nothing more than a function call with a standard set of parameters to the destination receive entry point (through a pointer).

With immediate mode interfaces, programmers must be careful to avoid uncontrolled recursive message passing. For example, assume a system contains a processor and a memory directly connected by a pair of interfaces, each in immediate mode. The processor sends request messages to the memory, and the memory sends back response messages (the processor is a master and the memory is a slave). If the processor sends a new request message before returning from a previous response message from the memory, the message passes may become uncontrolled recursive function calls. The processor's interface being in queued mode prevents this situation.

The queue Operation

The queue operation causes a message to be received by a local interface as though it was sent from the remote interface. The operation is used by a module to access the framework message queues and send a message to itself.

Framework routines perform a queue operation on the simulation or debug channels. The interface handle specifies the interface to queue the message to. The queued message is delivered to the specified interface after the specified cycle delay.

In FIGURE 2-2 on page 11, the queue operation is shown for interface *a* and interface *b*. The message does not go out of the interface; it only comes into the interface. Queued and immediate mode interfaces are treated identically by the queue operation.

2.4 Cycle Paradigm

MPSAS can simulate synchronous and asynchronous hardware.

To support synchronous hardware, the framework uses the concept of a cycle. Its closest analogy is the system clock of a computer system. Module instances can perform work each cycle (for example, a processor advancing its instruction pipeline).

To support asynchronous hardware, the framework allows modules to send messages with zero delay.

A cycle consists of six steps:

1. Delivering positive-phase messages whose delay has expired

2. Calling positive-phase module cycle entry points

3. Delivering zero-delay positive-phase messages sent in phase 2

4. Delivering negative-phase messages whose delay has expired

5. Calling negative-phase module cycle entry points

6. Delivering zero-delay negative-phase messages sent in phase 5

The first three steps are collectively known as the positive-phase of the cycle. The last three steps are collectively known as the negative-phase of the cycle. The positive-phase and negative-phase allow a module to synchronize its activities to both the rising and falling edges of a clock.

FIGURE 2-3 shows a timing diagram analogy for a cycle. The numbers 1 through 6 refer to the steps of a cycle.

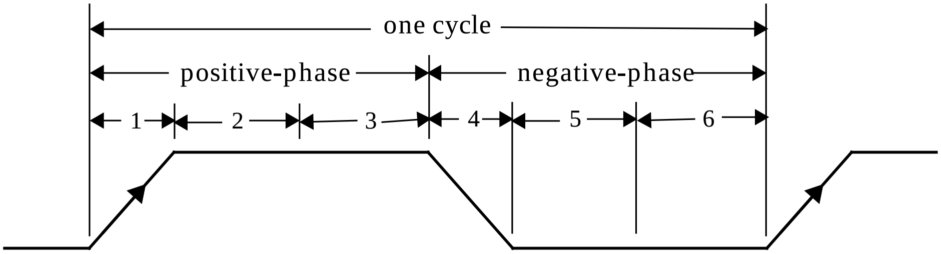

FIGURE 2-3 Cycle Phases

During steps 1 and 4, messages are delivered. These steps are analogous to synchronous logic's sampling of its inputs on the rising and falling edge of the clock, respectively. The nonvertical rising and falling edges in FIGURE 2-3 represent modules sending zero-delay messages in response to those they received in step 1 and 4, respectively.

During steps 2 and 5, module cycle entry points are called. They may send messages. This behavior is analogous to synchronous logic driving its outputs.

During steps 3 and 6, any zero-delay messages sent by the cycle entry points are delivered, allowing asynchronous logic to communicate before the next cycle starts.

If the steps of a cycle are being executed, the simulation is running; otherwise, it is stopped.

Messages sent to an interface in queued mode are delivered after the delay expires. A message is sent to a particular phase (positive or negative). A message sent with a delay of *n* cycles in cycle c will be delivered to the beginning of its associated phase of cycle c+*n* (that is, *n* cycles later).

Positive-phase, zero-delay messages sent in step 2 or 3 are delivered in step 3. Negative-phase, zero-delay messages sent in step 5 or 6 are delivered in step 6. Zero-delay messages sent to the phase opposite to the phase from which they are sent are not allowed; the attempt causes a fatal error.

Messages sent to an interface in immediate mode are delivered in the step in which the send operation is invoked (they bypass the framework message queues). Therefore, such messages can be received in any of the six steps (since messages can be sent in any of the six steps.)

Pseudocode for the steps of a cycle as executed by the framework is shown below.

```
              for each positive-phase simulationQ message
                     if message’s delay is 0
step 1                       deliver message
                     else
                             decrement message’s delay
step 2        call positive-phase module cycle entry points
           for each positive-phase simulationQ message added by
step 3           cycle entry points
                     if message’s delay is 0
                             deliver message

              for each negative-phase simulationQ message
                     if message’s delay is 0
step 4                       deliver message
                     else
                             decrement message’s delay
step 5        call negative-phase module cycle entry points
           for each negative-phase simulationQ message added by
step 6           cycle entry points
                     if message’s delay is 0
                             deliver message

              increment cycle count
```

During steps 1 and 4, each message in the positive-phase and negative-phase simulation channel queues is delivered to the corresponding receive entry point of its destination interface or its delay is decremented. Receive entry points can add messages to the simulation channel queues. Therefore, the messages being processed in the simulation channel queues may have been put there at two different times:

1. They may have already been in the queue before the loop was started.

2. They may have been added to the queue by a module’s receive entry point called by the deliver action of step 1 or 4.

During steps 2 and 5, the positive-phase and negative-phase cycle entry points (respectively) of each module instance (that registered one with the framework) are called. A cycle entry point can also add messages to the simulation channel queues.

During steps 3 and 6, all messages with zero delay that were added to the simulation queues by the cycle entry points are delivered. This is done because these messages must be delivered before the next cycle starts. Any messages with zero delay added to the simulation channel queues by the receive entry points called in step 3 and 6 are delivered before the phase finishes.

Finally, after step 6, the cycle count is incremented and the cycle is finished.

2.5 Layers

The MPSAS makefiles and directory structure divide the source files for the simulator into a hierarchy of layers. Each layer creates a new level of services. Those services are created from the source files in that layer and the services in all the layers below it. The bottom layer is the framework layer.

To the framework, the layers do not appear as a hierarchy; they appear as independent entities composed of:

- A layer name
- A list of user interface commands implemented by the layer
- A list of module classes simulated by the layer
- A layer initialization routine.

The layer name is used for display.

The framework user interface module uses the list of user interface commands to distribute the set of user interface commands across the layers. Layers can add commands to support the services created by that layer.

The framework uses the list of module classes to determine which module classes are available in a simulator executable.

The framework calls the layer initialization routine once, soon after the simulator is started. The layer initialization routine initializes the data structures used by that layer. Register message types can be registered with the framework in layer initialization routines. The alternative would be to register message types in a module's initialization routine. This alternative is not recommended because message types are typically used by more than one module.

2.6 Framework Services

The framework provides modules with several additional services in the following categories:

- Data types
- Interaction with the user
- Asychronous input

2.6.1 Data Types

MPSAS allows modules to use variables of any C data type (for example, int, short, long) in their state or in their messages. However, the framework provides a set of types based on the SPARC version 8 data types (see TABLE 2-1).

TABLE 2-1 Framework SPARC Data Types

Type	Size (in bits)	Description
`Byte`	8	unsigned byte
`s_Byte`	8	signed byte
`HWord`	16	unsigned halfword
`s_HWord`	16	signed halfword
`Word`	32	unsigned word
`s_Word`	32	signed word
`LWord`	64	unsigned long (double) word
`s_LWord`	64	signed long (double) word

The preceding types along with the built-in C data types `float` and `double` (floating-point single and floating-point double) constitute the entire set of SPARC version 8 data types except for floating-point quad.

Use the preceding types in module states and messages in preference to equivalent (but less precise) standard C types such as unsigned `char` and `int`. The sizes of the standard C types depend upon the compiler used, whereas the types in TABLE 2-1 are guaranteed to be the specified size.

The `LWord` and `s_LWord` data types are actually typedef'ed as `double` since the C compiler does not support 64-bit integers. You should never use the standard C arithmetic and logical operators (for example., `+`, `==`, `>`) with the `LWord` types because those operators will consider the `LWord` types as floating-point values rather than as 64-bit integer values. You can use the C assignment operator (that is, `=`) to assign one `LWord` to another. The framework provides a set of macros and routines to perform arithmetic, logical, and conversion operations on the `LWord` types (see the *Math64* manual page).

TABLE 2-2 shows some general-purpose data types the framework defines and standard data types it uses in a nonstandard way.

TABLE 2-2 Framework Non-SPARC Data Types

Type	Size (in bits)	Description
`Bool`	8	Boolean (`0` = `FALSE`, `1` = `TRUE`)
`caddr_t`	32	opaque pointer

2.6.2 Interaction with the User

Modules interact with the simulator user in several ways:

- Module commands
- Accesses
- State dump and restore
- Memory load

Module Commands

During the configuration phase, a module informs the framework of the name of each module command it supports and the corresponding module command entry point. This entry point is invoked by the framework when the simulator user executes the command. This mechanism allows modules to extend the set of user interface commands.

Accesses

The framework provides commands that allow the user to print module variables, set them, dump them out to trace files, and use them in expressions. However, the framework itself has no built-in knowledge about the variables defined by modules. Modules use a mechanism called *accesses* to register information about the module's state with the framework, allowing the framework to manipulate variables as required for these purposes.

A package called an *access class* must exist for each type of variable to be described by accesses. The framework provides a rich set of access classes that directly supports the SPARC version 8 data types, as well as some other constructs familiar to C programmers. The framework supports strings, bit fields, and arrays. A grouping mechanism collects accesses and treats them as one. New access classes can be created if necessary.

State Dump and Restore

A module can dump its state to a file or restore its state from a file. These operations are performed in response to commands from the user.

The state dump and restore facility allows the user to store the state of a system, quit from the simulator, start the simulator again, and then restore the saved state of the system.

Memory Load

The memory load facility initializes modules that simulate memory (for example, RAM and ROM). The contents of a user-specified file are loaded into the module’s memory.

2.6.3 Asynchronous Input

The framework provides a facility (called *sigio*) to allow modules to independently accept asynchronous input from UNIX file descriptors. The framework reads data from the file descriptor when it is available and sends it to its associated module in a message. Because of the method the framework uses to provide this facility, modules must not write data to the file descriptor directly. Instead, they send a message containing the data, and the framework writes the data to the file descriptor.

CHAPTER **3**

Writing a Module

This chapter explains some of the issues involved in writing a module. It provides guidance in the following areas:

- Module instance state
- Use of accesses
- Use of interfaces
- Performance tips

3.1 Module Instance State

All instances of a module class share its routines, but each instance requires its own independent module state. You meet this requirement by collecting the members of the state into a C structure; then, instruct the module class to dynamically allocate and initialize the structure for each module instance. Typically, the state structure is called *modname*`_state`, where *modname* is the name of the module class.

A pointer to the state structure is registered with the framework for each module instance. When the framework calls an entry point in a module, it passes back to it the state pointer for that module instance. The instance casts the opaque pointer to a *modname*`_state` structure pointer and accesses its state via the state pointer. Typically, the state pointer variable is called `msp` (module state pointer).

The module state can be any legal C data structure. However, some practical limitations do exist. The limitations occur because any members of the state that may change during the simulation must be able to be dumped to and restored from a file (see Section 4.2, *User Interface Entry Points*).

The example module state structure is listed below. Appendix B, *Example Module Listing*, contains the source code listing of an example module called example. This module simulates a simple count-down timer containing a small memory. Portions of the source listing are included throughout this document as examples.

```
#define MAX_CORE_SIZE   0x4000
struct example_state {
    char           *inst_name;  /* module instance name */
    caddr_t         intr_intf;  /* interrupt interface handle */
    LWord           reg_addr;   /* address of timer register */
    Byte           *core_ptr;   /* pointer into core array */
#define EXAMPLE_DUMP_PT count
    Word            count;      /* timer register */
    Byte            core[MAX_CORE_SIZE]; /* memory array */
};
```

The inst_name member is a pointer to the module instance's name. It is used by the module instance to identify itself to the simulator user. Typically, modules have this member in their state.

The intf_intf member is an opaque pointer that contains the interrupt interface handle. The handle identifies an interface to the framework for interface operations such as send and queue. The handle is also passed to the receive entry point of each interface. An interface's handle is stored in the state if the module must perform a send or queue operation on that interface and the interface's handle is not available from a previous receive operation.

The other members of the example state structure are described later in this book.

3.1.1 Static and External Variables

Any static or external variable declared in a module class is shared by all instances of that module. Any change an instance makes to the value of one of these types of variables affects all other instances. Use these types of variables with care because their use may violate the independence of each instance.

Typically, static and external variables store read-only data structures, such as tables, for all instances of a module class to share.

3.2 Use of Accesses

An *access* is an entity that presents to the framework—using a standard interface—everything it needs to manipulate a module variable (such as message fields). At configuration time, a module instance defines accesses for each of its variables to which the user is to have access. The routines used by the module instance routine to do this are introduced in this chapter and described in detail in the Access Create (for macros that create accesses) and Access Control (for routines that control aspects of the behavior of any access) manual pages in Appendix A.

Every access is an instance of some *access class*. If you are familiar with object-oriented languages such as C++, think of an access as being an object and an access class as being a class in the object-oriented sense. You will find that some of the terminology, such as "constructor," was borrowed from that paradigm.

The framework contains a number of access classes for variables of type `Word`, `HWord`, `char*`, and so on. You can write your own access classes, although doing so is not a trivial task for someone who has never written one before. It is probably best to understand the kinds of data that can be described with access classes and stick to them as much as possible. Of course, if the user need not have access to certain data, then there is no need to describe that data using accesses—and hence no concern for whether existing access classes can describe them.

The following is a simplified form of the code used by the `cpu` module's create instance routine to describe two pieces of its state:

- `irl`, the interrupt request level, which is a `Byte`
- `tbr`, the trap base register, which is a `Word` divided into two bit fields, `tba` and `tt`.

```
#include "types.h"
#include "expr.h"
#include "state_access.h"
 ...
        /* this code gets executed by cpu_create_instance() */
        ACCESS *access;

        BYTE("irl", &msp->irl);
        access = RO(WORD("tbr", &msp->tbr.w));
        MEMBER_BITF(access, "tba", TBR_TBA_MASK);
        MEMBER_BITF(access, "tt", TBR_TT_MASK );
        access_compact_print(access);
         ...
```

The `irl` is handled by a simple use of the `BYTE` macro, which describes variables of type `Byte`. It is passed the name by which the user will refer to the variable, and a pointer to the variable's data in memory. That is all the framework needs to know to allow the user to manipulate the `irl` value, as seen here.

```
cpu1: print irl
0x00

cpu1: set irl=3

cpu1: when irl changes {print -v cpu1.irl; stop}
```

Creating the `tbr` variable is somewhat more involved. First, we use the `WORD` macro to describe the entire variable as a `Word`, just as we used the `BYTE` macro for `irl`. Those macros return a pointer to the `ACCESS` data structure they create; we didn't need the pointer with `irl`, but we need it with `tbr`. We pass the pointer to the `RO` macro, making the access read-only—that is, an attempt by the user to change its value with the `set` command will fail.

The `RO` macro also returns the pointer to the access, which we save for further calls. The next two lines use `MEMBER_BITF` to create two `Bitfield` accesses, `tba` and `tt`, as "members"—variables within a variable—of `tbr`. Because `tbr` has members, printing it results in its members being printed. Ordinarily, members are each printed on a separate line; the call to `access_compact_print` causes the members of `tbr` to be printed on the same line. The result is a variable that behaves as follows.

```
cpu1: print tbr
tba=0x0 tt=0x0

cpu1: set tbr.tba=1

cpu1: set tbr.tt=2

cpu1: print tbr
tba=0x1 tt=0x2

cpu1: expr tbr
0x1020  4128
```

Notice that members of an access are referred to by the notation *access*`.`*member.*

The following macros create accesses: FLOAT, DOUBLE, BOOL, BYTE, S_BYTE, CHAR, HWORD, S_HWORD, WORD, S_WORD, LWORD, S_LWORD, WORD_ADDR, LWORD_ADDR, STR, BBITF, S_BBITF, BITF, S_BITF, LBITF, S_LBITF, MEMBER_BITF, MEMBER_LBITF, GROUP, and ARRAY. The *Access Create* manual page discusses these macros in more detail, but two—GROUP and ARRAY—merit special mention here.

- GROUP — Any access can have members. Ordinarily, having members only affects the way an access is printed, in the manner demonstrated above. But the GROUP macro creates an access that is nothing more than the collection of its members. That is, no data are associated with the group itself; when the group is dumped, all of its members are dumped. A group may not be assigned a value. Use a group to create a collection of data that the user will likely want to display or dump as a whole.

  The access_add_member function adds members to a group or to any other access. For the common cases involving module state variables, state_members provides another, more convenient, way to do the same thing. As shown in the example, MEMBER_BITF and MEMBER_LBITF automatically make the accesses they create members of another access, since that is usually the way bit fields are used.
- ARRAY — This macro creates an array, using some other access as a template for an element of the array. For example, a command to print or set an array prints or sets every element of the array; or a command can specify an individual member to be affected by supplying an integer index expression as a parameter.

A number of routines modify the behavior of an access.

- access_hide — Prevents an access from showing up in the output of the list command. For example, a developer might use access_hide with a special access created for debugging that is not for use by the average user.
- access_invalid_print — Overrides the error message that is normally displayed when the user prints a variable that is currently invalid.
- access_address_generator — Describes a variable whose address may vary during the simulation; you supply a function that generates the current address of the variable whenever the framework needs it.
- access_change_func — Specifies a function that is consulted each time the user tries to set a particular variable. This “change routine” approves or rejects the set and can even modify the value that is ultimately assigned.
- access_arg — Stores an arbitrary pointer in an access for use by any routines you associate with the access.
- access_custom_set, access_custom_print, access_custom_dump, and access_custom_restore — Override specific functions that would ordinarily be handled by the accesses’s class; you need not write a new access class.

3.3 Use of Interfaces

Interfaces provide the mechanism for modules to communicate with other modules. A module may need to communicate with many other modules and may support many types of transactions. To decide how to partition a module's communications needs among its interfaces, use criteria similar to those discussed in Section 1.2, *Modularization Criteria*.

A module can support multiple interface types. For example, an MMU module may support two interface types: virtual and physical. The virtual interface connects to a processor, and the physical interface connects to the memory subsystem.

A module can support multiple instances of each interface type. For example, a memory module can support multiple-port access by supporting an interface type that allows access to the memory and then supporting one instance of this interface type for each port.

More than one type of transaction can be multiplexed on an interface by having a field in the message data specify the message's transaction type. Also, different message formats can be multiplexed on an interface by use of multiple message types on it.

3.3.1 Simulating a Bus

A hardware bus consists of multiple writers and multiple readers connected together. Access to the bus is controlled by some arbitration logic.

In MPSAS, a bus is simulated with its own module. A bus module has an interface for each device connected to the bus and performs arbitration and routing among the bus writers (also called *masters*).

FIGURE 3-1 shows a bus module with several masters (m) and slave (s) interfaces connected to it. The master interface is connected to the writers; the slave interface is connected to the readers.

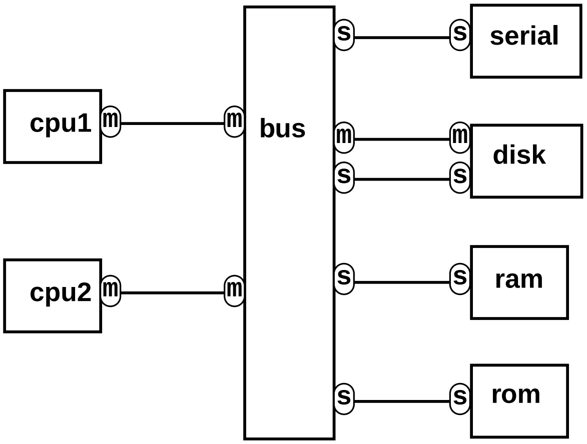

FIGURE 3-1 Bus Module

The bus arbitrates among the various messages sent to it by the masters. Masters send requests to the bus at the beginning of a cycle, and the cycle entry point of the bus uses an arbitration algorithm to select which master is granted access to the bus. That master's message is forwarded to the addressed slave.

3.3.2 Interface State Pointer

A module can associate a state structure with an interface. A pointer to the structure (called the *interface state pointer*) is registered with the framework.

The interface state pointer can be read or written by the module at any time and with any value. The `get_interface_state` framework macro returns the interface state pointer. The `set_interface_state` framework macro sets the interface state pointer at any time. The `register_intf_state` framework routine sets the interface state pointer during the configuration phase.

The data pointed to by the interface state pointer is considered to be part of the module instance state; it just has a special method to access it.

3.4 Performance Tips

The performance of the simulator is greatly affected by the level of detail in the modules of that system. As the level of detail is increased, the simulator performance decreases since it is performing more work to simulate the same activity.

However, you can use the techniques described below to maximize performance at a given level of detail.

3.4.1 Preallocated Memory

As much as possible, a module should avoid using the dynamic memory allocation routines (that is, `malloc` and `calloc`) in code executed in the simulation phase. When a module needs to allocate memory for the data of a new message, it is a natural to use dynamic memory allocation routines. However, instead of dynamically allocating the memory during the simulation phase, for performance allocate the memory in the configuration phase and store a pointer to the memory in the module instance's state. The module returns the preallocated memory when done with it.

Performance sidelight: When a developer modified a processor module to preallocate memory for each message it sent to the memory subsystem (that is, each time it a fetched an instruction or performed a load / store operation), performance of the entire simulator increased by approximately 15%.

Use the preallocation technique only when you know the maximum memory requirements of a module. Usually, you know this information because the hardware being modeled has finite resources.

Using preallocated memory slightly complicates the dumping and restoring of the module instance's state. The framework provides routines that greatly aid in this effort. For more information, see Section 4.2, *User Interface Entry Points*.

3.4.2 Common Message Formats

Many modules support multiple interfaces. Potentially, each one of these interfaces may speak a different protocol and may require a different message format. However, the interfaces are usually not independent. That is, a message received on one interface may result in a message being sent by the module on another interface.

If the message data formats between such interdependent interfaces are similar enough, the message received on one interface can then later be sent on another interface. This technique saves the overhead of allocating / deallocating memory for a message (even preallocated memory has some overhead) and copying information in the message. Usually, only a few fields in the message need to be updated before the message is sent.

3.4.3 Cycle Entry Points

A cycle entry point is invoked every cycle of the simulation. Thus, the performance of the simulator is significantly affected by the overhead imposed by its cycle entry points.

Avoid cycle entry points in a module whenever possible. If a module must have a cycle entry point, make it as efficient as possible. For example, many cycle entry points really only need to be called when triggered by an event (such as receiving a message). The cycle entry point should test the trigger as efficiently as possible (for example, testing one Boolean variable in the module's state).

CHAPTER **4**

Module Entry Points

A module's entry points are divided into three categories: configuration, user interface, and simulation.

Configuration entry points configure the module class, instances of the module class, and its interfaces. They are called by the framework during the simulator's configuration phase.

User interface entry points support requests made on behalf of the user. They are called by the framework during the simulator's simulation phase.

Simulation entry points simulate the behavior associated with the module. They are composed of the interface receive entry points and the cycle entry point. The simulation entry points are called by the framework during the simulator's simulation phase.

For details on how these entry points fit into the various phases, review Chapter 2, *Framework*.

4.1 Configuration Entry Points

A module class provides the six entry points that the framework calls to handle the following six steps of configuration for that module:

1. Initializing the module class
2. Initializing each module instance of the module class
3. Initializing each interface of each module instance
4. Registering objects that can be shared between modules
5. Looking up objects that can be shared between modules
6. Verifying the final configuration of each module instance

4.1.1 Module Class Initialization Entry Point

A module class's initialization entry point (also called the *module init entry point*) is called just once, before any of the other module entry points. For a description of how the framework determines the address of a module init entry point given its class name, refer to Chapter 6, *Product Structure*.

A module init entry point is defined as follows:

```
int modname_module_init()
```

By convention, the name of every module entry point called by the framework is prefixed by the name of the module class (referred to as *modname* above). For example, the name of the `cpu` module's module init entry point is `cpu_module_init`.

The module init entry point initializes the module class, typically:

- Calls framework routines to register the address of the module's other entry points
- Initializes global variables
- Sets the module class `extra` pointer

Registering Module Entry Points

The following framework routines register with the framework the address of the corresponding module configuration entry points (shown in parentheses; they are described later):

- `register_create_instance` (create instance)
- `register_config_interface` (config interface)
- `register_verify_config` (verify config)
- `register_shared_object_create` (shared object create)
- `register_shared_object_lookup` (shared object lookup)

All of the module configuration entry points are optional. All of the routines except `shared_object_create` and `shared_object_lookup` are used by most modules. These framework registration routines can only be called once per module init entry point.

The following framework routines register with the framework the address of the corresponding module entry points (shown in parentheses; they are described later):

- `register_cycle` (cycle)
- `register_pos_cycle` (positive-phase cycle)
- `register_neg_cycle` (negative-phase cycle)
- `register_dump` (dump)
- `register_restore` (restore)
- `register_load_file` (load file)
- `register_module_command` (module command)

The framework calls the module entry points during the simulation phase. All of the entry points are optional, although most modules register dump and restore entry points. These framework registration routines can be called only once per module init entry point, except for the `register_module_command` routine, which is called once for each module command.

Initializing Global Variables

The module init entry point is a convenient place to initialize global variables since it is called only once no matter how many instances of that module are specified in the configuration file.

Module Class Extra Pointer

Each module class has an opaque pointer, called `extra`, associated with it. A module class sets the pointer to a value that can be retrieved by the other module configuration entry points. Most modules do not use the `extra` pointer. It is used primarily when the other module configuration entry points are shared by more than one module init entry point (and therefore by more than one module class).

The `extra` pointer is stored in the framework, but its value is not used by the framework.

The module init entry point sets its `extra` field with the `set_module_extra` routine. The value of the `extra` field is fetched with the `get_module_extra` routine and can be called by any module configuration entry point.

Return Value

The module init entry point returns `0` if it detects no errors; otherwise, it returns nonzero, causing the simulator to exit with an appropriate message.

Example

Below is a listing of the module init entry point of the `example` module (see Appendix B, *Example Module Listing*, for the complete source listing for the module).

```
int
example_module_init()
{
        register_create_instance(example_create_instance);
        register_config_interface(example_config_intf);
        register_verify_config(example_verify_config);
        register_shared_object_create(example_create_shared_obj);
        register_shared_object_lookup(example_lookup_shared_obj);
        register_dump(example_dump);
        register_restore(example_restore);
        register_cycle(example_cycle);
        register_module_command("core", example_core_cmd,
            core_shorthelp, core_longhelp);
 }
```

The `example` module registers the create instance, config interface, verify config, shared object create and lookup, dump, restore, and cycle entry points, and one command entry point.

The declaration for `core_shorthelp` and `core_longhelp` is not shown in the preceding listing; they are both static arrays of characters.

4.1.2 Create Module Instance Entry Point

A module's create module instance entry point (also called the *create instance entry point*) is called once for each instance of that module specified in the configuration file.

A create instance entry point is defined as follows:

```
caddr_t modname_create_instance(args)
char *args;
```

The create instance entry point creates an instance of its module class:

- Allocates and initializes the module instance state
- Parses the configuration file instance arguments
- Registers a cycle entry point

Module Instance State

The create instance entry point allocates and initializes a copy of the module instance state (typically, a structure called *modname*`_state`).

The framework allows a module instance to have any number of interfaces. If a module needs to maintain some state for each interface, the module programmer might find it convenient to allocate space for the state in the create instance entry point. The get_num_interfaces framework routine returns the number of interfaces connected to the calling module instance. The get_num_interfaces_by_type framework routine returns the number of interfaces connected to the calling module instance of a particular type.

The get_config_mod_instance_name framework routine returns a pointer to the module instance's name. A module uses the name only for identifying itself to the user. Most messages generated by modules start with the module instance name followed by a colon. A copy of the instance name pointer is stored in the instance state structure with the name inst_name. A module must not modify the name.

Parsing Arguments

Each module instance declared in the configuration file can have a character string of arguments associated with it. These arguments configure the behavior of a module instance, allowing different instances of the same module class to behave differently.

The create instance args parameter points to the configuration file argument string. If there are no arguments, it points to the empty string ("").

The interpretation of the argument string is up to the module; it is not interpreted by the framework. You can specify keywords in upper case and parameters in lower case (for example, PIPELINE enabled).

You can specify multiple arguments for the configuration file. The framework concatenates them all into a single string separated by newlines.

The get_argc_argv and look_for_keyword routines are helpful in parsing the argument string. The get_argc_argv routine parses a null-terminated string and returns an array of arguments (like the arguments passed to a C main routine). The look_for_keyword routine searches an array of arguments for a particular keyword and, optionally, certain parameters.

Registering Cycle Entry Point

You can call the register_cycle or register_pos_cycle framework routines from the create instance entry point to register a positive-phase cycle entry point with the framework. You can call the register_neg_cycle framework routine to register a negative-phase cycle entry point. These entry points override the cycle entry points that might have been registered by the module init entry point of a module.

Registering the cycle entry points in the create instance entry point instead of in the module init entry point allows different module instances of the same module class to have different cycle entry points (or, perhaps, none at all).

Return Value

The create instance entry point returns an opaque pointer. The framework considers this the state pointer of the module instance being configured. If the instance has no state, it just returns `0` (`NULL`).

Reporting Errors

The create instance entry point reports errors by calling the `fatal_nodump` framework routine.

The other module configuration entry points do not call the `fatal_nodump` routine to report errors because they have a return code that indicates whether an error occurred. All return values of the create instance entry point are valid.

Example

Below is a listing of the create instance entry point of the `example` module.

```
static caddr_t
example_create_instance(args)
        char           *args;
{
        struct example_state *msp;
        int             argc;
        char          **argv;
        int             index;

        /* Allocate a chunk of memory big enough for my state. */
        if ((msp = (struct example_state *) calloc(1,
            sizeof(struct example_state))) == NULL) {
                fatal_nodump("Unable to malloc.\n");
        }

        /* Store pointer to instance name in my state. */
        msp->inst_name = get_config_mod_instance_name();

        /* Convert args to an argc/argv data structure. */
        argc = get_argc_argv(&argv, args);

        /* Get address (64-bit value) of my counter register. */
        if ((index = look_for_keyword(argc, argv, “REG_ADDR”, 1,
            msp->inst_name)) <= 0) {
                fatal_nodump("%s: error in config file\n",
                        msp->inst_name);
        }

        msp->reg_addr = strto64(argv[index], (char **)NULL, 0);

        /* Create access to count register. */
        WORD("count", &msp->count);

        /* Return an opaque pointer to my state. */
        return (caddr_t)msp;
}
```

The `example_create_instance` framework routine allocates and clears a copy of the module’s state structure (see Appendix B, *Example Module Listing* for the definition of the state structure).

`get_config_mod_instance_name` retrieves the instance name. The pointer is stored in the state for error messages later on.

Argument vector `argv` is obtained from the `args` string by the `get_argc_argv` framework routine.

The look_for_keyword framework routine is called to search for the REG_ADDR keyword with one parameter. If the keyword is not found or the parameter is missing, then fatal_nodump displays an error message and quits from the simulator.

Otherwise, the parameter is converted from an ASCII string to a 64-bit integer with the strto64 framework routine and assigned to the reg_addr member of the state structure.

An access called count is created with the WORD framework macro to allow the simulator user to read and write the count value. The count is the current value of the 32-bit count-down timer.

Finally, the state is cast to an opaque pointer and returned to the framework.

4.1.3 Configure Interface Entry Point

A module's configure interface entry point (also called the *config intf entry point*) is called once for each interface of each instance of that module specified in the configuration file.

A config intf entry point is defined as follows:

```
int modname_config_intf(state, intf)
caddr_t state, intf;
```

The config intf entry point configures an interface:

- Verifies the interface declaration as it appears in the configuration file
- Calls framework routines to register interface options

The *state* parameter is an opaque pointer to the module instance state of the instance that owns the interface. The state parameter is a common parameter to module entry points; it is the value returned to the framework by the module's create instance entry point.

The *intf* parameter (referred to as the *interface handle*) is an opaque pointer that identifies the interface to the framework. The module saves the value of the interface handle in its state if it needs to send a request message (that is, *not* a message in response to a request) to that interface in the simulation phase. The framework maintains information about each interface. The module can access this information through a set of framework macros and routines.

Verifying the Interface Declaration

The config intf entry point verifies that the interface declaration, as it appears in the configuration file, is acceptable to the module. It can examine the interface type, its connected/unconnected status, and argument string to perform this verification.

The `get_interface_type` framework macro returns a pointer to the interface type string. The config `intf` entry point must ensure that the type is supported by the module.

The `is_interface_connected` framework macro returns `1` if the interface is connected to another interface and `0` if it is unconnected. Sending a message to an unconnected interface causes a fatal error. Messages are received on an unconnected interface only if the module queues a message on it. This behavior can be used by a module to delay an action.

The `get_interface_args` framework macro returns a pointer to the interface argument string. If there are no interface arguments, the macro returns a pointer to the empty string (“”).

The interpretation of the argument string is up to the module; it is not interpreted by the framework. You can specify keywords in upper case and parameters in lower case (for example, DEBUG on).

You can specify multiple interface arguments for the configuration file. The framework concatenates them all into a single string separated by newlines.

The `get_argc_argv` and `look_for_keyword` routines are helpful in parsing the argument string.

Registering Interface Options

The config intf entry point calls framework routines to register interface options with the framework.

- `register_sim_intf_receive` and `register_pos_sim_intf_receive` specify the address of the positive-phase simulation channel receive entry point.
- `register_neg_sim_intf_receive` specifies the address of the negative-phase simulation channel receive entry point.
- `register_dbg_intf_receive` specifies the addresses of the debug channel receive entry point. A receive entry point is optional, although if a message is sent to an interface on a channel that has not registered a receive entry point, a fatal error occurs.

The four preceding framework registration routines can only be called once per interface.

- `register_sim_intf_mode` and `register_dbg_intf_mode` specify the receive mode of an interface (immediate or queued) for the simulation (positive-phase and negative-phase) and debug channels, respectively. A mode must be registered for a channel if, and only if, a receive entry point has been registered for that channel.

- The `register_intf_state` framework routine sets the interface state (stored in the framework's interface structure) to an opaque 32-bit value. Typically, the opaque value is a pointer to a structure that is associated with the interface. The value of the interface state pointer is retrieved with the `get_interface_state` framework macro at any time.

Return Value

The config `intf` entry point returns `0` if it detects no errors; otherwise, it returns nonzero, and the framework causes the simulator to exit.

Example

Below is a listing of the config intf entry point for the `example` module.

```
static int
example_config_intf(state, intf)
    caddr_t     state;
    caddr_t     intf;
{
    struct example_state *msp = (struct example_state *)state;
    char       *type = get_interface_type(intf);
    int         connected = is_interface_connected(intf);
    int         okay_to_be_unconnected = 0;
    static char duplicate_interface_type[] = "%s: interface\
\"%s\":\n\tOnly one \"%s\" interface type allowed.\n";
    static char unknown_interface[] = “%s: interface\
\”%s\”:\n\tUnknown interface type \”%s\”.\n”;
    static char unconnected_interface[] = “%s: interface\
\”%s\”:\n\tInterface type \”%s\” cannot be unconnected.\n”;

    if (strcmp(type, SLAVE_INTF_TYPE_NAME) == 0) {
            register_sim_intf_mode(IMMEDIATE_MODE);
            register_sim_intf_receive(example_slave_sim_rcv);
            okay_to_be_unconnected = 1;
    } else if (strcmp(type, INTR_INTF_TYPE_NAME) == 0) {
            if (msp->intr_intf) {
                    fwprintf(duplicate_interface_type,
                        get_interface_mod_inst_name(intf),
                        get_interface_name(intf), type);
                    return 1;
            }
            msp->intr_intf = intf;
    } else {
            fwprintf(unknown_interface,
                get_interface_mod_inst_name(intf),
                get_interface_name(intf), type);
            return 1;
    }
    if (!connected && !okay_to_be_unconnected) {
            fwprintf(unconnected_interface,
                get_interface_mod_inst_name(intf),
                get_interface_name(intf), type);
            return 1;
    }
    return 0;
}
```

The `example_config_intf` routine casts the opaque `state` parameter to an `example_state` struct pointer called `msp`.

The get_interface_type framework macro extracts the type string from the interface handle.

The is_interface_connected macro extracts the connected flag from the interface handle.

The type string is compared with the interface type names (the macro definitions are not shown) supported by the module. If the type does not match, an error message is displayed. Note the use of the get_interface_mod_inst_name and get_interface_name framework macros to get information about the interface for the error message.

If the interface type is slave, then the register_sim_intf_mode and register_sim_intf_receive framework routines register the slave receive entry point with the framework. The example module only sends a message to a slave interface if it receives a request message from that interface. Therefore, the example module does not need to store its slave interface handles. Also, it does not require the slave interface to be connected because if it receives a message from a slave interface, it must be connected.

If the interface type is interrupt, then the example module saves the interface handle in the module state for later use. First, it tests whether the handle in the state has already been assigned. If it has, the example module displays an error message because it allows only one interrupt interface. Note that the slave interface never performed this check since any number of slave interfaces are allowed and their interface handles need not be stored.

The interrupt interface does not have a receive entry point or mode assigned to it because the example module does not receive messages on this interface—it is write-only.

Finally, the config intf entry point tests whether the interface is illegally unconnected. If so, an error message is displayed. If not, the config intf entry point returns to the framework.

4.1.4 Create Shared Object Entry Point

The create shared object entry point is called once per declaration of each shared object in each module instance in a module class.

A create shared object entry point is defined as follows:

```
int modname_create_shared_obj(state,obj_name)
caddr_t state;
char *obj_name;
```

The create shared object entry point configures a shared object:

- Verifies the shared object declaration as it appears in the configuration file
- Calls framework routines to register shared object options

The *state* parameter is an opaque pointer to the module instance state of the instance that owns the shared object. The *obj_name* parameter is the name of the object from the configuration file that is being configured for sharing.

Verifying Shared Object Declaration

The name of the object is checked against a specific list of objects in the module that are capable of being shared.

Registering the Shared Object

The create shared object entry point calls framework routines to register information about the shared object.

- `register_object_ptr` specifies the address of the object being shared.
- `register_object_size` specifies the size of the object being shared.

Return Value

The create shared object entry point returns `0` if it detects no errors; otherwise, it returns nonzero, and the framework causes the simulator to exit.

Example

Below is a listing of the create shared object entry point for the `example` module.

```
static int
example_create_shared_obj(state, obj_name)
    caddr_t state;
    char *obj_name;
{
    struct example_state *msp = (struct example_state *) state;

    if(!strcmp(obj_name, “count”)) {
            register_object_ptr(&msp->count);
            register_object_size(sizeof(Word));
            return 0;
    }
    else {
       fwprintf(“%s:example_create_shared_obj:%s Object unknown\n”);
            return 1;
    }
 }
```

The `example_create_shared_obj` routine casts the opaque `state` parameter to an `example_state` struct pointer called `msp`.

The `obj_name` parameter is compared with the object names the module supports. If the parameter does not match, then an error message is printed. If the parameter matches, then the pointer to object in the module state is returned to the framework.

4.1.5 Lookup Shared Object Entry Point

The lookup shared object entry point is called once per lookup of each shared object in each module instance in a module class.

A lookup shared object entry point is defined as follows:

```
int modname_lookup_shared_obj(state,obj_name)
caddr_t state;
char *obj_name;
```

The lookup shared object entry point configures a shared object:

- Verifies the shared object lookup as it appears in the configuration file
- Calls framework routines to get information about the shared object

The *state* parameter is an opaque pointer to the module instance state of the instance that intends to share the object. The *obj_name* parameter is the name of the object to which the pointer to the shared object is being assigned.

Verifying the Shared Object Lookup

The name of the object is checked against a specific list of objects in the module that are capable of being shared.

Configuring the Shared Object

The lookup shared object entry point calls framework routines to get information about the shared object.

- `get_object_ptr` returns the address of the object being shared.
- `get_object_size` returns the size in bytes of the object being shared.

Return Value

The lookup shared object entry point returns `0` if it detects no errors; otherwise, it returns nonzero, and the framework causes the simulator to exit.

Example

Below is a listing of the lookup shared object entry point for the `example` module.

```
static int
example_lookup_shared_obj(state, obj_name)
    caddr_t state;
    char *obj_name;
{
    struct example_state *msp = (struct example_state *) state;
    union u_example_psr sh_psr;
    ACCESS* access;
    void *ptr = get_object_ptr();

    if(!strcmp(obj_name, “example_psr”)) {
        msp->sh_example_psr = (union u_example_psr *) ptr;
        msp->sh_example_psr_size = get_object_size();
        /* Shared PSR */
        access = WORD(“sh_example_psr”,
                     &msp->sh_example_psr->w);
        access_compact_print(access);
        MEMBER_BITF(access, “et”, WORD_MASK(sh_psr, et));
        return 0;
    }
    else {
    fwprintf(“%s:example_lookup_shared_obj:%s Object unknown\n”,
msp->inst_name, obj_name);
      return 1;
    }
}
```

The state pointer is cast to variable called `msp`.

The pointer to the shared object is retrieved by `get_object_ptr` and is assigned to a void pointer. This pointer is then cast to `u_example_psr` and assigned to the variable in the state.

The size, in bytes, of the object is retrieved by the `get_object_size` routine.

4.1.6 Verify Configuration Entry Point

A module’s verify configuration entry point (also called the *verify config entry point*) is called once for each instance of that module specified in the configuration file.

A verify config entry point is defined as follows:

```
int modname_verify_config(state)
caddr_t state;
```

The verify config entry point examines the state passed into it, determines if it contains a legal configuration of the instance, and checks the combinations and number of interfaces of the instance. The module's other configuration phase entry points must store this information in the state so it can be checked in the verify config entry point.

Return Value

The verify config entry point returns `0` if it detects no errors; otherwise, it returns nonzero, and the framework causes the simulator to exit.

Example

Below is a listing of the verify config entry point of the `example` module.

```
static int
example_verify_config(state)
    caddr_t         state;  /* per instance state pointer */
{
    struct example_state *msp = (struct example_state *)state;
    static char missing_interface[] = "example: module instance
\"%s\":\n\tInterface type \"%s\" not present.\n";

    if (msp->intr_intf == NULL) {
            fwprintf(missing_interface, msp->inst_name,
                INTR_INTF_TYPE_NAME);
            return 1;
    }

    return 0;
}
```

The `example_verify_config` routine casts the opaque `state` parameter to an `example_state` struct pointer called `msp`.

It makes sure that the instance has an interrupt interface by examining the interrupt interface handle it sets in its config intf entry point. If the interrupt interface is missing, `example_verify_config` returns `1`; otherwise, the state is valid and `example_verify_config` returns `0`.

4.2 User Interface Entry Points

During the simulation phase, the framework calls module user interface entry points to handle the following user-initiated actions:

- Dump module instance state to a file
- Restore module instance state to a file
- Execute a module command
- Load simulated memory from a file

A module's user interface entry points are registered with the framework during the configuration phase.

4.2.1 Dump State Entry Point

A module's dump state entry point (also called the *dump entry point*) is called when the user requests that the state of that module instance be dumped. If a module has state that changes on the basis of message traffic on the simulation channel—as most modules do—the module should register a dump entry point.

The dump entry point works in conjunction with the restore entry point described in *Restore State Entry Point* on page 52. The code in the dump and restore entry points both agree on the format of the dumped state so that the restore entry point can rebuild the state of the module instance as it was when it was dumped by the dump entry point.

A dump entry point is defined as follows:

```
Bool modname_dump(state, stream)
caddr_t state;
FILE    *stream;
```

By convention, the name of the dump entry point is the module name (*modname*) followed by an underscore and the word `dump`.

The *state* parameter to the dump entry point is the opaque state pointer of the instance to be dumped.

The module dumps the state to the stream specified by the *stream* parameter, which is opened and closed by the framework.

The members of the instance state structure that must be dumped are those that are associated with the simulation channel and that change during simulation (for example, writable registers).

You must take special care when dumping pieces of module state that are implemented using pointers. Pointers in the state structure could point to memory that is part of the state structure (for example,. a pointer into an array in the state structure) or could point to memory that is outside the state structure (for example, statically or dynamically allocated memory). In both of these cases the pointers must be dumped to the stream in an address-independent form because pointers are not reusable in a different invocation of the simulator. For example, if a state structure member contains a pointer to a string, the actual string must be dumped. If a state member is a pointer into another state member that is an array, this pointer could be converted into an integer index and this index value dumped.

The framework provides the following routines that assist in dumping pieces of state that were implemented with pointers.

- `dump_buffer` writes a dynamically allocated block of memory to a stream.
- `dump_string` writes a null-terminated string to a stream.
- `dump_array_ptr` converts a pointer into an array into its equivalent integer index and then writes it to a stream.
- `dump_message` writes a message to a stream.
- `dump_intf` writes an interface handle to a stream.

Return Value

The dump entry point returns a `Bool` type, which is `true` if the dump is successful and `false` otherwise.

Example

The following state structure is used throughout this section to illustrate the various user-interface entry points.

```
#define MAX_CORE_SIZE   0x4000
struct example_state {
    char           *inst_name;  /* module instance name */
    caddr_t         intr_intf;  /* interrupt interface handle */
    LWord           reg_addr;   /* address of timer register */
    Byte           *core_ptr;   /* pointer into core array */
#define EXAMPLE_DUMP_PT count
    Word            count;      /* timer register */
    Byte            core[MAX_CORE_SIZE]; /* memory array */
};
```

Note – The core_ptr and core members are not used in any other routines of the example module outside this section and are only of use for this section.

The example module's state structure is divided into two sections: a section that can be dumped directly to the stream without conversion and a section that cannot.

The structure members in the group below the EXAMPLE_DUMP_PT definition are members that can be dumped directly to the stream and need no conversion.

The members in the group above the EXAMPLE_DUMP_PT definition are members that either need conversion before being dumped or do not need to be dumped at all. The inst_name, intr_intf, and reg_addr members should not be dumped; they are initialized during the configuration phase and do not change in the simulation phase. If these were dumped and then restored, the values restored would invalidate values initialized during the configuration phase and could cause serious problems. Core_ptr points into the core array, so it must be converted to a form that can be written to the stream.

The following example illustrates a module dump entry point.

```
/*
 * This routine is called by the framework when the user requests
 * the state of an instance of the example module class to be
 * dumped to a file.
 */
static Bool
example_dump(state, fp)
        caddr_t         state;
        FILE           *fp;
{
        struct example_state *msp = (struct example_state *) state;
        char           *src = (char *) &msp->EXAMPLE_DUMP_PT;
        int             size;  /* # bytes we dump */

        /*
         * Calculate the size of the portion of state below
         * the dump point.
         */
        size = sizeof(struct example_state) -
            (int) (src - (char *) msp);

        /*
         * Write out portion of my state below the dump point.
         */
        if (fwrite(src, size, 1, fp) != 1) {
                fwprintf_unbuf("%s: fwrite failed\n",
                    msp->inst_name);
                fwperror("");
                return FALSE;
        }
        /*
         * Write out members of my state above dump point
         * that change with the simulation.
         */
        if (dump_array_ptr(fp, msp->core, sizeof(Byte),
            msp->core_ptr)) {
                fwprintf_unbuf("%s: dump_array_ptr failed\n",
                    msp->inst_name);
                fwperror("");
                return FALSE;
        }

        return TRUE;
}
```

In the `example_dump` routine, the number of bytes of the portion of the state structure that can be dumped without conversion is calculated and saved in the `size` variable. Next, this section is written to the stream. The rest of the code writes the portion of the state structure that needs conversion. The call to `dump_array_ptr` converts the pointer `core_ptr` into its equivalent index into the `core` array and writes this value to the stream.

If the `fwrite` or the `dump_array_ptr` routines fail, an error message is displayed and the dump entry point returns `false`. The error messages are displayed with the `fwprintf_unbuf` routine, followed by `fwperror` to display the system error. The `fwprintf_unbuf` routine is used because `fwperror` is unbuffered and order between the messages should be preserved.

4.2.2 Restore State Entry Point

A module‘s restore state entry point (also called the *restore entry point*) is called when the user requests the state of that module to be restored. If a module registers a dump entry point, it should register a restore entry point.

The restore entry point works in conjunction with the dump entry point described in *Dump State Entry Point* on page 48.

A restore entry point is defined as follows:

```
Bool modname_restore(state, stream)
caddr_t state;
FILE    *stream;
```

By convention, the name of the restore entry point is the module name (*modname*) followed by an underscore and the word `restore`.

The *state* parameter to the dump entry point is the opaque state pointer of the instance to be dumped.

The module restores the module instance state from the stream specified by the *stream* parameter, which is opened and closed by the framework. The stream specified by the parameter *stream* should be considered read-only.

The restore entry point clears the module instance’s state and then restores the module instance’s state from a file that was previously written by the module’s dump entry point.

The clearing of the module instance’s state members is usually accomplished by overwriting the existing state of the module. In some cases, state members need special attention. For example, state members that have pointers to dynamically allocated memory should free the memory before overwriting the pointers.

You can restore pointers that were dumped by the framework dump routines by using the following matching routines.

- `restore_buffer` reads a block of memory from a stream and dynamically allocates space for it.
- `restore_string` reads a null-terminated string from a stream `malloc`'s space and reads a string from a file.
- `restore_array_ptr` reads an index from a file and converts it to its equivalent pointer into an array.
- `restore_message` reads a message from a stream.
- `restore_intf` reads an interface handle from a stream.

Return Value

The module restore entry point returns a `Bool` type. It returns `true` if the restore is successful and `false` otherwise.

Example

The following example illustrates the `example` module's restore entry point.

```
/*
 * This routine is called by the framework when the user requests
 * the state of an instance of the example module class to be
 * restored from a file.
 */
static Bool
example_restore(state, fp)
        caddr_t         state;
        FILE           *fp;
{
       struct example_state *msp = (struct example_state *) state;
        char           *src = (char *) &msp->EXAMPLE_DUMP_PT;
        int             size;  /* # bytes we dump */

       /*
        * Calculate the size of the portion of state below
        * the dump point.
        */
        size = sizeof(struct example_state) -
            (int) (src - (char *) msp);

       /*
        * Read in portion of my state below the dump point.
        */
        if (fread(src, size, 1, fp) != 1) {
             fwprintf_unbuf("%s: fread failed\n",msp->inst_name);
                fwperror("");
                return FALSE;
        }

       /*
        * Read in members of my state above dump point that
        * change with the simulation.
        */
        if (restore_array_ptr(fp, msp->core, sizeof(Byte),
            &msp->core_ptr)) {
                fwprintf_unbuf("%s: restore_array_ptr failed\n",
                    msp->inst_name);
                fwperror("");
                return FALSE;
        }
```

In the example_restore routine, the size of the portion of the state structure that can be dumped without conversion is calculated and saved in the size variable.

Next, this section is read from the stream. The call to restore_array_ptr reads an index and converts it to its equivalent pointer, core_ptr, into the core array.

If the fread or restore_array_ptr routines fail, an error message is displayed and the restore entry point returns false. The error messages are displayed with the fwprintf_unbuf routine followed by fwperror to display the system error. The fwprintf_unbuf routine is used because fwperror is unbuffered and order between the messages should be preserved.

4.2.3 Module Command Entry Point

A module command entry point is registered with the framework during the configuration phase for each user interface command the module adds. When the user issues the command, the framework calls the appropriate entry point.

A command entry point is defined as follows:

```
Bool modname_cmd_cmd(state, cmd, args)
caddr_t  state;
char    *cmd;
char    *args;
```

By convention, the name of the module command entry point starts with the name of the module (*modname*) followed by the name of the command (*cmd*) and the letters cmd (separated by underscores).

The *state* parameter to the command entry point is the opaque state pointer of the module instance requested to execute the command.

The *cmd* parameter points to a null-terminated string that contains the command name.

The *args* parameter points to a null-terminated string that contains the remainder of the command line typed by the user (not including the newline).

The framework provides routines to help with the processing of commands—parsing strings, parsing and evaluating expressions, and manipulating access variables.

- ui_parsew returns the next word (sequence of nonwhite characters) in the string. The routine is useful when parsing strings.
- ui_parse_delimiter finds a delimiter in a string and returns the string following it. The routine is useful when parsing strings.
- xpr_parse parses an expression from a string. The routine is useful when a command argument is an expression.
- expr_boolean ensures that an expression's value can be interpreted as a Boolean.

- expr_access ensures that an expression represents an access
- expr_get_LWord and expr_get_Word parse and evaluate expressions and return LWord and Word, respectively.
- expr_eval_boolean evaluates an expression that results in an integer as a Boolean value.
- expr_show displays an expression given the expression tree.
- expr_show_value displays the value of an expression.
- expr_equiv compares two expressions to see if they represent the same expression.
- expr_free frees an expression tree.
- capture_printf and init_capture_info can be used with expr_show and expr_show_value to capture their output in a string.
- Macros EXPR_IS_INTEGER and EXPR_IS_ACCESS determine whether an expression is of integer type or represents an access, respectively.
- The following macros provide convenient ways of obtaining the value of an expression once it has been evaluated: BYTE_VALUE, HWORD_VALUE, WORD_VALUE, LWORD_VALUE, FLOAT_VALUE, DOUBLE_VALUE, STRING_VALUE, and VALUE_HI_WORD.
- access_valid provides a convenient, standard way of ensuring that an access expression is valid.
- access_isa and access_class_isa determine whether an access or access class is related to some other access class.

A module command entry point might need to send messages to other modules to get the information necessary to process a command. This communication is done over the debug channel. An example of this is an MMU module command that requires the MMU to translate a virtual address into a physical address—MMU might have to get information from a RAM device to make the translation. Refer to Section 2.4, *Cycle Paradigm* for a discussion on sending and receiving messages over the debug channel.

The user interface provides variables that contain command return codes. Module command entry points can optionally set these variables to return a numeric result to the user.

The user interface variables that hold these return codes are cmd_result, for integer results, and cmd_result_double, for floating-point results. To set one of these user interface variables, you simply set the global variable of the same name. The framework provides a macro SET_CMD_RESULT_AS_WORD that can set the cmd_result variable to a Word value.

Commands that can run for more than a second or so (for example, in loops) should periodically check the fw_terminate_cmd global variable. This is a Bool variable that the framework sets to false every time a command is started. If the user types a Control-C while the command is running, the framework sets fw_terminate_cmd to true. Commands can detect this situation and terminate the command.

Return Value

The module command entry point returns one of two values to the framework:

- UI_CMD_IS_DONE — Returned if the command processing has completely finished before this entry point returns.
- UI_CMD_IS_NOT_DONE — Returned when the module has not finished processing the command at the time the entry point returns to the framework. In this case, the framework will not start executing another command until this one finishes. Use UI_CMD_IS_NOT_DONE when a command entry point sends messages to other modules and needs to wait until all the messages propagate.

  For example, the cpu module's read command entry point sends a message on the debug channel to the memory subsystem to access memory. The cpu module's read command entry point returns UI_CMD_IS_NOT_DONE to the framework. Later, when the cpu module gets a response back from the memory subsystem, it completes the read command; the cpu's debug receive entry point notifies the framework that the read command has finished by sending a message to the framework's command_done interface.

Example

Following is the example module's core command entry point. The example displays an entry in the core array. If an index is specified, it displays that entry, otherwise, it displays the byte pointed to by the core_ptr variable.

```
/*
 * This routine is called each time the user invokes the "core"
 * command.
 */
static int
example_core_cmd(state, cmd, args)
        caddr_t     state;
        char       *cmd;    /* Actual command string */
        char       *args;   /* Any arguments to the command */
{
     struct example_state *msp = (struct example_state *) state;
        char       *rest_of_line;
        char       *index_str;
        Byte       *cptr;
        int         index_num;

        if (*args == ’\0’) {   /* if nothing specified */
                /* display the core_ptr entry */
                cptr = msp->core_ptr;
                if (cptr == NULL) {
                        fwprintf("%s: core_ptr is NULL\n",
                            msp->inst_name);
                        /* Set cmd_result variable to -1. */
                        cmd_result = make64(-1, -1);
                        return UI_CMD_IS_DONE;
                }
                index_num = (msp->core_ptr - &msp->core[0])
                    / sizeof(Byte);
        } else {
                /*
                 * Parse core argument. Make sure only one arg
                 * on line.
                 */
                rest_of_line = ui_parsew(args, &index_str);
                if (*rest_of_line != ’\0’) {
                        fwprintf("Usage: core [<index>]\n");
                        return UI_CMD_IS_DONE;
                }
```

- more -

```
                /*
                 * Get expression parser to evaluate index
                 * expression.
                 */
               if (expr_get_Word(index_str, EXPR_OPT_MSG,
                    _, &index_num) == FALSE) {
                        return UI_CMD_IS_DONE;
                }

              if (index_num < 0 || index_num > MAX_CORE_SIZE-1) {
                        fwprintf(
                         "%s: invalid index specified \"%s\".\n",
                            msp->inst_name, index_str);
                        return UI_CMD_IS_DONE;
                }

                cptr = &msp->core[index_num];
        }

        fwprintf("%s: core[%d] =  0x%x\n", msp->inst_name,
            index_num, *cptr);
       /*
        * Set cmd_result variable to value of the element
        * displayed.
        */
        SET_CMD_RESULT_AS_WORD(*cptr);

        return UI_CMD_IS_DONE;
}
```

Note – The framework provides other mechanisms by which you can display contents of variables (that is, accesses). This example just illustrates how it can be done by a command entry point.

The `example_core_cmd` routine looks at the argument string to determine which array element to print. If the user did not supply arguments, the `core_ptr` field in the state structure is used as a pointer to the array element to print and the index is calculated. If an argument is supplied, it is evaluated as an expression and the index is set to the result.

Once the index and pointer are calculated, the value is printed and the `cmd_result` user-interface variable is set to the displayed value. This function returns `UI_CMD_IS_DONE` because all command processing has been completed at the end of the routine.

4.2.4 Load Memory Entry Point

Modules that simulate memory can register a module load memory entry point (also called a *load file entry point*). This entry point is called by the framework when the user issues a `load` or `load_section` command that refers to this module.

A load file entry point is defined as follows:

```
Bool modname_load_file(state, stream, addr, size)
caddr_t   state;
FILE     *stream;
LWord     addr;
int       size;
```

By convention, the name of the load file entry point is the module name (*modname*) followed by _load_file.

The *state* parameter to the command entry point is the opaque state pointer of the instance specified in the `load` command.

The *stream* parameter specifies the stream from which the module reads the image of memory; the stream is opened and closed by the framework.

The *addr* parameter is the destination address, in the simulated address space, of the image to be loaded.

The *size* parameter is the size, in bytes, of the memory image to be loaded.

Typically, the contents of the load file entry point consist of reading *size* bytes of data from *stream* into the module instance's simulated memory at address *addr*.

Return Value

The load file entry point returns a `Bool` type. It returns `true` on successful loading of the file, and `false` otherwise.

Example

The example below illustrates the `example` module‘s load entry point, which loads a file's contents into its `core` array.

```
/*
 * This routine is called when the user asks that a file be loaded
 * into an instance of the example module’s memory.
 */
static Bool
example_load_file(state, fp, addr, size)
    caddr_t   state;
    FILE     *fp;     /* stream to load memory from */
    LWord     addr;   /* address to load memory into */
    int       size;   /* # bytes to load into my memory */
{
    struct example_state *msp = (struct example_state *) state;

    /*
     * Check for valid address range.
     * Valid addresses range from 0 to MAX_CORE_SIZE-1
     */
    if (UCMP64(add_64_32(addr, size), >,
        make64(0, MAX_CORE_SIZE-1))) {
            fwprintf(
              "%s: invalid address 0x%x%08x specified.\n",
                msp->inst_name, HI_W(addr), LO_W(addr));
            return FALSE;
    }

    if (fread(&msp->core[LO_W(addr)], size, 1, fp) != 1) {
            fwprintf_unbuf("%s: load file failed\n",
                msp->inst_name);
            fwperror("");
            return FALSE;
    }

    return TRUE;
}
```

In the example, the address range for `addr` is checked first. If it is valid, then the actual data is read from the file into the `core` array.

4.3 Simulation Entry Points

During the configuration phase, each module can register entry points that are invoked during the simulation phase to inform the module about the progression of the simulation. Four types of module simulation entry points can be registered:

- Start simulation entry point
- Interface message receive entry point
- Positive-phase and negative-phase cycle entry points
- Simulation dieing entry point

4.3.1 Start Simulation Entry Point

A module class's start simulation entry point (also called the *start entry point*) is called just before the first cycle of the simulation is started.

A start entry point is defined as follows:

```
Bool modname_start(state)
caddr_t state;
```

The start entry point can perform any initialization related to the module instance's state and can send messages on the simulation channels (since the simulation phase has started).

The start entry point returns `true` if it detects no errors; otherwise, it returns `false`.

Many modules do not need a start entry point.

4.3.2 Receive Entry Point

A module registers a receive entry point for an interface at configuration time if it expects to receive messages on the interface.

Receive entry points are called by the framework to deliver a message to the module.

A receive entry point is defined as follows:

```
void modname_intftype_channel_rcv(state, intf, data, type, size, delay)
caddr_t state, intf, data, type;
int size, delay;
```

By convention, the name of a receive entry point is a concatenation of the following, separated by underscores: the module class name (*modname*), the interface type (*intftype*), the channel (*channel*)—dbg, sim, or dbg_or_sim for both—and the letters rcv.

The *state* parameter provides access to the module instance state registered during the configuration phase. This parameter is not interpreted by the framework and is typically cast into a pointer to this module's state structure.

The *intf* parameter is the interface handle for the interface on which the module received the message. It can be used to return the message to the sender or, if the receive entry point is used for multiple interfaces, to determine which interface received the message. This interface handle is also needed for certain framework services (for example, get_interface_state).

The *data* parameter is an opaque pointer to the message that is being delivered. The module code typically casts this into a pointer to a message structure to interpret the message.

The *size* parameter is the size, in bytes, of the message pointed to by data. If the value of size is 0, data might be a null pointer.

The *type* parameter specifies the message type, registered at configuration time, of the message being delivered.

The *delay* parameter specifies any remaining delay associated with a message. The delay is nonnegative for the simulation channel and –1 for the debug channel. If the receive routine is for the simulation channel, the delay is always 0 (since the delay has already been introduced by the framework before the message was delivered). If the interface is an immediate mode interface, the value of delay is the delay specified by the sender of the message. In that case, the module can simulate the delay with a queue operation, ignore it, or pass it on.

The purpose of a receive entry point is to allow the module to react to a message being sent on an interface. Each module reacts differently, depending on the device it is simulating. The receive entry point might need to send messages or queue messages in order to react to the message.

The framework provides the following macros to send messages.

- send_sim_channel and send_pos_sim_channel send a message on the positive-phase simulation channel.
- send_neg_sim_channel sends a message on the negative-phase simulation channel.
- send_dbg_channel sends a message on the debug channel.
- send_either_channel and send_pos_either_channel send a message on either the positive-phase simulation or debug channel.

- send_neg_either_channel sends a message on either the negative-phase simulation or debug channel.

For the details of these routines, see the *Send* manual page.

The preceding send macros also have functions with the same name but _func appended. You can use them if a module requires a pointer to a send routine.

The framework provides four functions to queue messages on an interface:

- queue_on_dbg_channel
- queue_on_sim_channel
- queue_on_pos_sim_channel
- queue_on_neg_sim_channel

These routines queue messages on the debug and simulation channels, respectively. For the details of these functions, see the *Queue* manual page.

The module can use functions get_interface_state and set_interface_state to get or set the interface state pointer for the receiving interface.

You can use the following macros to dynamically change the receive entry points for an interface:

- modify_sim_intf_receive
- modify_pos_sim_intf_receive
- modify_neg_sim_intf_receive
- modify_dbg_intf_receive

The module can obtain the pointer to the current receive entry point by calling the following routines:

- get_sim_intf_receive
- get_pos_sim_intf_receive
- get_neg_sim_intf_receive
- get_dbg_intf_receive

Example

Following is a listing of the simulation receive routine from the example module.

```
static void
example_slave_sim_rcv(state, intf, data, type, size, delay)
    caddr_t state;/* opaque ptr to this module’s state struct */
    caddr_t intf; /* interface message came in on. */
    caddr_t data; /* opaque ptr to the data pkt */
    caddr_t type; /* type id of the data pkt */
    int     size; /* sizeof the data packet */
    int     delay;/* always 0 for queued interfaces */
{
    struct example_state *msp = (struct example_state *) state;
    struct gen_bus_pkt *gbp = (struct gen_bus_pkt *) data;
    Word tmp;

    if (EQ64(gbp->paddr, msp->reg_addr)
         && gbp->size == sizeof(Word)) {
        switch (gbp->type) {
         case GEN_BUS_RD:
              GBP_DATA_WORD(gbp) = msp->count;
              break;
        case GEN_BUS_WR:
              msp->count = GBP_DATA_WORD(gbp);
              break;
        case GEN_BUS_RW:
              tmp = msp->count;
               msp->count = GBP_DATA_WORD(gbp);
              GBP_DATA_WORD(gbp) = tmp;
              break;
        default:
            fatal("%s: unknown gen_bus_pkt type of 0x%x.\n",
                msp->inst_name, gbp->type);
        }
        gbp->status = GEN_BUS_OK;
    } else {
        gbp->status = GEN_BUS_FAULT;
    }
    send_sim_channel(intf, (caddr_t)gbp, size, type, delay);
}
```

In the above example, the parameters state and data are cast into useful structure pointers. This example module interprets the incoming data parameter as a pointer to a gen_bus_pkt structure.

A response to the message is built based on the type field of the gen_bus_pkt. If the message is a read request, count is copied into the data field of the gen_bus_pkt. If the message is a write request, the data field from the gen_bus_pkt is copied into count. If the message is a read-modify-write request, the data field from the gen_bus_pkt and count are exchanged.

Next, the status field of the gen_bus_pkt is set, and the message is sent back to the sender. Notice that the intf parameter is used to return the message to the sender. Because the module does not originate requests and only responds to requests, this module can just use the intf parameter to return the message to the sender instead of storing the interface handle during the configuration phase.

4.3.3 Cycle Entry Points

Each module can register a positive-phase and negative-phase cycle entry point during the configuration phase. These entry points are called by the framework once each cycle during the appropriate phase. For more details on cycles see Section 2.4, *Cycle Paradigm*.

A cycle entry point is defined as follows:

```
void modname_cycle(state)
caddr_t state;
```

By convention, the name of a cycle entry point is the module class name (*modname*) followed by an underscore and the word cycle.

The *state* parameter provides access to the module instance state registered during the configuration phase. This parameter is not interpreted by the framework and is typically cast into a pointer to this module's state structure.

Cycle entry points can use the same framework functions described with the receive entry points earlier.

Example

Following is the cycle entry point for the example module.

```
static void
example_cycle(state)
      caddr_t         state;
{
      struct example_state *msp = (struct example_state *) state;
        struct gen_int_pkt *gip;

      if (--msp->count == 0) {
           gip = new_gen_int_pkt();

           gip->action = INTERRUPT_SET;

           send_sim_channel(msp->intr_intf, (caddr_t)gip,
               sizeof(*gip),gen_int_msgtype, 0);
      }
}
```

This routine first casts the incoming parameter `state` to a useful structure pointer for this module. The `example` module maintains a count of its state. That count is decremented each cycle, and when the count reaches zero, an interrupt message is sent on its interrupt interface.

4.3.4 Simulation Dieing Entry Point

A module instance's simulation dieing entry point (also called the *dieing entry point*) is called just before the simulation exits to clean up a module before the simulation exits (for example, to remove temporary files).

A dieing entry point is defined as follows:

void *modname*_dieing(*state*)
caddr_t state;

The dieing entry point must be registered in the module's create instance entry point. Most modules do not need a dieing entry point.

CHAPTER 5

Miscellaneous Framework Routines

This chapter describes some framework routines used by modules. The use of these routines is not confined to any one of the three types of module entry points (configuration, user interface interaction, or simulation) or accesses, so they are considered “miscellaneous.”

5.1 Display Routines

The framework provides a family of routines to display messages for the simulation user. They are similar in function to the standard C display routines that write characters to `stdout` and `stderr`.

TABLE 5-1 shows each framework routine and its corresponding standard C display routine.

TABLE 5-1 Display Routines

Framework Display Routine	Standard C Display Routine
`fwprintf`	`printf`
`fwprintf_unbuf`	`fprintf(stderr, ...)`
`fwflush`	`fflush(stdout)`
`fwputchar`	`putchar`
`fwputs`	`puts`
`fwperror`	`perror`

The framework display routines offer two advantages over the standard C display routines:

1. All output generated by the framework display routines is sent to the display and also to the simulator log file.
2. Portions of the output generated by the standard C display routines can be lost; framework display routines do not have this problem.

Another advantage of using the framework display routines is that the modules are isolated from the actual display devices; they never have to specify `stdout` or `stderr`. The framework could be enhanced to support a graphical user interface (GUI) without the need to change the modules.

5.2 Error Exit Routines

The framework provides four `fatal` routines associated with exiting from the simulator when an error is detected.

- The `fatal` routine should be used whenever a module detects a nonrecoverable programming error (for example, an unexpected null pointer). The `fatal` routine causes the simulator to:

  a. Display a message provided by the caller (`printf`-style calling conventions).
  b. Display the stack backtrace of the simulator.
  c. Exit from the simulator cleanly.
  d. Dump core.

- `fatal_sim` is similar to the `fatal` routine for simulation channel requests. For debug channel requests, it just displays a message and returns.
- `fatal_nodump` is the same as the `fatal` routine except that it does not dump core. Use this routine whenever a module detects a nonrecoverable user error and no other mechanism exists to inform the framework (for example, a `malloc` command fails during a module user interface command).
- `fatal_push` specifies the address of a function to call when step 3 of the `fatal` or `fatal_nodump` routines is reached. Most modules do not need this step.

Do not call the standard C `abort` and `exit` routines; if they are called, the simulator will not exit cleanly.

5.3 Halt Routines

The framework provides two routines to halt the simulation. A halted simulation cannot be restarted. A module should halt the simulation when it discovers an error situation but wants to allow the simulator user to inspect the state of the simulation when the error occurred.

- `halt` displays a message provided by the caller (`printf`-style calling conventions) and then halts the simulation.
- `halt_simulation` simply halts the simulation.

5.4 Simulation Control Routines

The framework provides routines to start, stop, and test the running status of the simulation.

- `stop_simulation` stops the simulation before the next cycle starts.
- `start_simulation` starts the simulation running.
- `sim_running` returns a value that indicates if the simulation is currently running or not.

5.5 64-Bit Integer Routines

The framework provides routines and macros to support 64-bit integers (the `LWord` and `s_LWord` framework data types). There are many routines and macros, and they are not listed here individually. Please refer to the *Math64* manual page for complete details.

There are routines and macros to support most of the standard C integer arithmetic and logical operators, as well as routines and macros to support conversion to and from other C data types.

5.6 Interface Manipulation Routines

The framework provides routines that allow a module instance to create and connect interfaces independently of the contents of the configuration file.

- `create_unconnected_interface` creates an unconnected interface.
- `connect_interfaces` connects two interfaces owned by a module instance.

5.7 Symbol Table Access Routines

The framework provides two routines to access the symbol table information that gets loaded into the framework by the `load`, `load_section`, and `symtab` commands.

- `toSymbolic` converts an address into its equivalent symbol.
- `toAddr` accepts a symbol name and converts it to its equivalent address.

5.8 SPARC Assembly / Disassembly Routines

The framework provides a routine to access the SPARC one-line assembler and a routine to access the SPARC one-instruction disassembler.

- The `assemble` pointer points to the one-line assembler routine. It assembles the contents of a string into its equivalent SPARC 32-bit integer instruction.
- The `print_disassembly` pointer points to the disassembler routine. It disassembles a SPARC 32-bit integer instruction and displays it.

5.9 String Routines

The framework provides three routines to manipulate strings.

- strdup_fatal duplicates a string, using the standard C strdup routine. If the strdup fails, strdup_fatal calls the fatal routine with an appropriate error message.
- strmcpy and strmcat perform the same operations as the standard C strcpy and strcat routines, respectively, except that they concatenate an arbitrary number of strings.

5.10 ASCII Conversion Routines

The framework provides several routines to convert between some framework data types and ASCII representations.

- dtoa converts a double into its ASCII equivalent.
- ctoa converts a character into its octal ASCII equivalent.
- ctoa_hex converts a character into its hexadecimal ASCII equivalent.
- atoc accepts an ASCII string and converts it to its character equivalent.

5.11 Global Variables

The framework declares several global variables that can be accessed by modules.

- The cyclecount variable is equal to the number of cycles executed by the simulator since it was started. It is incremented by the framework each time a cycle finishes. Its value must never be written by modules. It can be used by modules for synchronization (for example, to enable a processor 1000 cycles into the simulation) or for display purposes.
- The instrcount variable is equal to the number of instructions executed by all of the processors since the simulation was started. The processor modules must increment this variable each time a non-annulled instruction is successfully or unsuccessfully executed. This value is used by the framework for performance measurements.
- The progname variable is set to the name of the simulator as invoked on the command line. Any path components of the name are removed. The progname variable must never be written by modules.
- The opt_simpleprint variable is a flag that indicates if the simpleprint option is enabled or disabled (by the user). When the simpleprint option is enabled, modules should try to produce terse output messages that can be easily parsed by a program.

CHAPTER 6

Product Structure

MPSAS comes with all of the source code for the framework, modules, and supporting routines. This source code and makefiles to create archives and executables from the source are arranged in a directory hierarchy, which is described in this chapter. This chapter also explains how to add your own modules and make other common changes.

6.1 MPSAS Root Directory

When you installed MPSAS using `extract_unbundled`, you specified a directory in which to install MPSAS, or you let the directory default to `/usr/share`. In the directory, the installation process created directory `sparctools`, and within that, directory `mpsas`. This directory, called the *mpsas root directory*, contains all of the source files, organized into various subdirectories.

6.2 Layers

MPSAS source files are conceptually organized into a hierarchy of layers, as shown in FIGURE 6-1. Generally, each layer can use modules, variables, functions, and other features defined in the layers below it, but cannot use things defined in the layers above it. The software in each layer is built separately from that in other layers.

Organizing the source into layers makes it possible to share code wherever possible and still have alternatives for the same functionality. In particular, the organization allows MPSAS to simulate multiple architectures that share some modules and have differing versions of other modules. The modules in common are simply placed in lower layers of the hierarchy. In effect, each layer provides a foundation of modules, message types, commands, variables, and so on that can be built upon in higher layers.

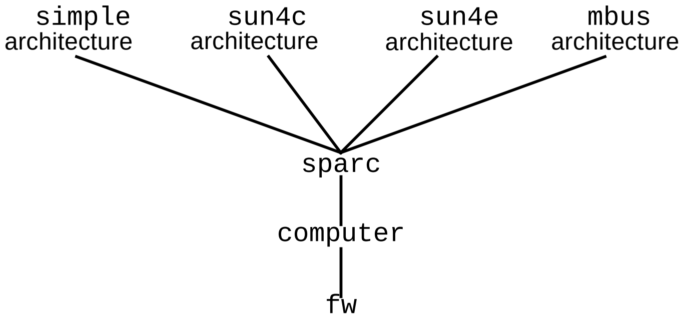

FIGURE 6-1 MPSAS Layers

The fw (framework) layer, on the bottom of the hierarchy, contains the user interface, the software that processes the configuration file, I/O-handling software, the routines that allow processes to communicate with one another, the software that calls all of the modules' cycle routines, and so on. The ui and sigio modules are in the fw layer.

The computer layer contains modules, message types, and other definitions that are useful in simulating a large class of computers. The ram, rom, serial, and simdisk modules are in the computer layer.

The sparc layer contains modules, message types, and other definitions that are useful in simulating a large class of SPARC-based computers. The cpu, fpu, and trap modules are in the sparc layer.

In the tree-shaped configuration formed by the various layers, the leaves represent architectures. An architecture layer is a layer in which an mpsas executable is built—that is, all of the modules required in order to simulate a complete system are present in an architecture layer or the layers below it. The product comes with four architecture layers; these architectures are described in *Multiprocessor SPARC Architecture Simulator (MPSAS) User's Guide*. The simple architecture contains no additional modules, since the minimal architecture it describes uses only modules defined in lower layers. But the other, more realistic architectures each add an MMU and modules for other architecture-dependent features.

Note that an mpsas executable contains only one choice from each vertical level of the tree; for example, the sun4e architecture consists of the four layers sun4e, sparc, computer, and fw. It would not be possible with the current arrangement of layers to create an mpsas executable containing both the sun4e and mbus layers, for example.

You can add to the layers that come with the product. You could, for example, add a new architecture called my_arch at the same level as the existing architectures. It could make use of all of the modules defined in the fw, computer, and sparc layers, but none of the modules contained in other architectures. Or, you could create a layer on top of the sun4c architecture; this might be appropriate if your design was very close to the sun4c architecture, since the new architecture could make use of any modules contained in the sun4c architecture. You could even create simulators for systems based on a different line of processors. All of these alternatives are depicted in FIGURE 6-2, with the fictitious layers in italics.

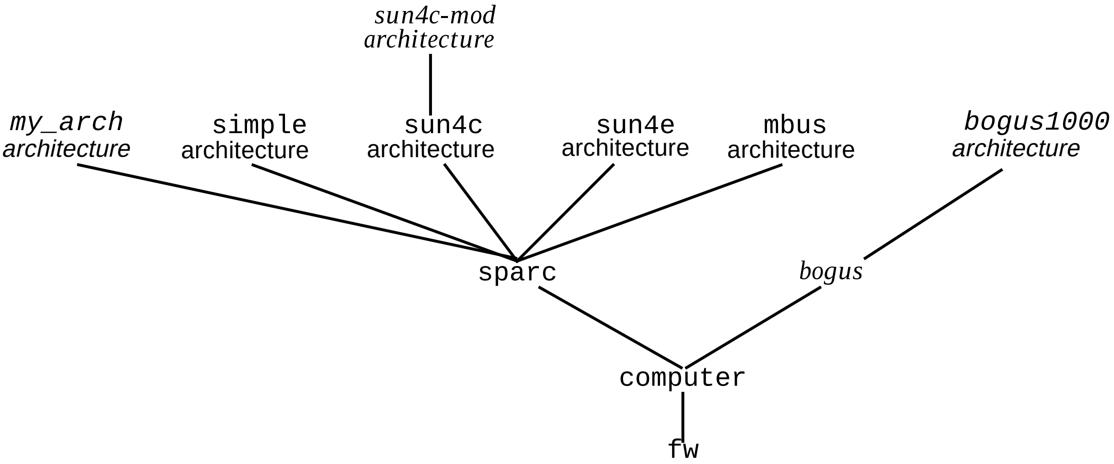

FIGURE 6-2 MPSAS Layers with Additional Fictitious Layers

6.3 Directory Structure

In the mpsas root directory you should find the following entities (directories are marked with a trailing slash; executables are marked with a trailing asterisk):

- makeall* — A shell script that builds all of the mpsas executables from source
- Makefile.common — A file included by most other makefiles, encapsulating most of the logic about how the system is built
- fw/ — A directory containing the fw layer
- computer/ — A directory containing the computer layer
- sparc/ — A directory containing the sparc layer
- simple/ — A directory containing the simple architecture layer
- sun4c/ — A directory containing the sun4c architecture layer

- sun4e/ — A directory containing the sun4e architecture layer
- mbus/ — A directory containing the mbus architecture layer
- stand/ — A directory containing sample programs for running on the simulator, as well as many routines of use in writing your own programs to run on the simulator (an appendix to the *User's Guide* describes this directory)
- util/ — A directory containing utility programs that work with the simulator

Many of these directories contain subdirectories to further organize the source code:

- fw/include/ — Public header files defined by the fw layer
- computer/include/ — Public header files defined by the computer layer
- sparc/include/ — Public header files defined by the sparc layer
- sparc/cpu/ — Source for the cpu module
- sparc/fpu/ — Source for the fpu module

6.4 Inside a Layer

A layer is always implemented as a directory (whose name is the name of the layer) in the mpsas root directory. Each layer directory contains certain files that support the layering concept:

- Makefile — Input file to the make program
- Makefile.inc — File included by makefiles in this layer and every layer above this one; these files actually contain the information that defines the hierarchy of layers
- layer.c — A C source file providing a special data structure describing the layer, as well as any layer initialization code

In addition, the layer directory will contain C source files and header files for the modules and commands defined in that layer. These files might be organized into subdirectories. Any header files that might be included by layers above this one are placed in an include subdirectory.

When make is run in a layer directory, it creates (at least) archive lib*layer*.a in that directory. This archive contains the object files for everything in this layer that might be used by a layer above this one; in particular, it contains all modules and commands defined by this layer. The object files constituting that archive will not be present as individual object files. There may be some source files for which the object does not go into the archive.

A zero-length file called our_lib_part is created during the make. This file's timestamp is used by make to determine whether a particular directory's source code that is destined for an archive needs to be recompiled.

In a layer representing an architecture, you will also find the file vers.c, which defines a string printed out when mpsas initializes, and by the version command

In an architecture directory, make creates several files in addition to the archive mentioned earlier. File mpsas is the simulator executable. Also created are files fingerprint.c, built_layers.c, and various object files not placed in the archive. fingerprint.c contains another string printed out by the version command. built_layers.c contains a table through which mpsas can access the data structures defined in the layer.c files in the various layers.

6.5 Makefile Usage

This section explains how to use make in layer directories, given the special makefiles they use.

6.5.1 Targets

The following makefile targets are available in any layer directory or subdirectory:

- all (the default target) — Makes everything in this directory and any subdirectories
- this — Makes everything in this directory (list any targets to be built in this directory as dependencies)
- clean — Removes all derived files (files generated by the make) in this directory and in any subdirectories
- this_clean — Removes all derived files in this directory
- lint — Runs lint on the C source files in this directory, leaving the output in lint.out. lint is run with -u and -q to suppress messages about non-Sun portability and about externals. Note that this way of running lint does not check the arguments of calls to functions defined in other layers.

To fully understand which files are affected by executing make, be sure to read *Multiple Sets of Derived Files* on page 81.

6.5.2 Faster Makes

Ordinarily, running make on mpsas in an architecture layer directory will cause make to visit each layer below the architecture to make sure that those layers are up-to-date. This procedure can take significant time, and if you are only making changes to files in your architecture directory, those makes will never actually make anything.

By setting macro X to the null string on the make command line, you can cause make to ignore the fact this layer's derived files are dependent upon the archives in other layers; therefore, make will not visit those layers. You might even want to create an alias to do such a make:

```
alias mk "make X="
```

Clearly, the link aspect of the build is unchanged—the archives of those other layers are still searched as before. If those archives do not exist, the build will fail. If they exist but are not up-to-date, make will not know this. In using X= you are relieving make of the responsibility to verify that those layer archives exist and are up-to-date.

6.5.3 Compiler Flags

make allows you to control the options with which the compiler and other programs used by make are run. You exercise this control by means of macro definitions on the make command line or by environment variables. Four of these macros—ASFLAGS, CFLAGS, CPPFLAGS, and LDFLAGS—are set by the mpsas makefiles and therefore should *not* be set by the user. However, the mpsas makefiles do allow you to add your own flags to those used by the mpsas makefiles. For example, developers will typically want to do all compiles and links with switch -g so that the resulting executable can be debugged. To add flag, put the following line in your .login file:

```
setenv CFLAGS0 -g
```

The following environment variables can be used to control flags in this way:

- ASFLAGS0 — Additional flags for as (used with .s and .S files)
- CFLAGS0 — Additional flags for cc (used with .c files and for linking)
- CPPFLAGS0 — Additional flags for cpp (used with .c files and .S files)
- LDFLAGS0 — Additional flags for ld (used for linking)

These flags are propagated to every submake and makes of other layers. CFLAGS, etc., are typically defined in /usr/include/make/default.mk and are described in the manual page for make.

Because the .KEEP_STATE feature of make is used by mpsas, changing these environment variables will cause many things to be remade.

6.5.4 Multiple Sets of Derived Files

In some cases, you might want multiple copies of an architecture's simulator in the architecture directory at the same time. For example, you might ordinarily compile the simulator with flags that allow debugging, but at some point decide that you want to test the performance of the optimized simulator. As another example, you might want a version of the simulator compiled with flags that allow profiling, or even a version compiled with a different compiler.

Ordinarily, recompiling with different switches will destroy all of the derived files as they currently exist, forcing `make` to regenerate them if you later rebuild with the old switches. However, the makefiles of `mpsas` allow the specification of a name suffix that is used in all archive names as well as in the `mpsas` executable name, allowing multiple copies to coexist in the same directory. To use this feature, set macro `NAME` on the command line to an appropriate suffix, as in the following `make` command, to produce an optimized version of `mpsas` (executed in the directory of an architecture layer):

```
make NAME=opt CFLAGS0=-O
```

In this case, after making archive `lib`*layer*`-opt.a` in each layer, `make` creates executable `mpsas-opt` in the architecture directory.

When you specify a name suffix in this manner, `make` looks for that suffix in *all* archive names—not just the archive in the current directory. The makefiles define macro `LIB` to be the name of the archive to be built in this layer, qualified by the `NAME` macro if it was used. Hence, every layer will list `${LIB}` as a dependency of its `this` target.

If you add other makefile targets to a layer and you want different versions of that target depending upon `NAME`, use macro `OBJ_NAME` in the target's name, as is done here for the `mpsas` executable:

```
this: mpsas${OBJ_NAME}
```

`OBJ_NAME` will either be nothing (if `NAME` was not used) or a hyphen followed by the value of the `NAME` macro.

Note – If you used `NAME` when executing `make` in a layer directory, making the `clean` target will only remove those files if you again specify `NAME`.

6.6 Adding to an Existing Layer

You can extend an existing layer in several ways. You might simply want to add a new C source file to be compiled and linked into the executable. Perhaps you have created a whole new module definition that you would like to add to the simulator so that you can configure it into your system. Or you might simply have created a new message type for use in the layer, or a command that the layer adds to the repertoire.

Probably the best approach to modifying a makefile is to look in existing mpsas makefiles for a case similar to yours and do the analogous thing. This section gives you a good idea of how to do the more common modifications.

6.6.1 Adding a File to the Layer Archive

Macro LIB_OBJS lists all of the object files to be included in the layer's archive. Adding the name of the object file to this list automatically causes it to be made and added to the archive. make knows how to build the object for the common kinds of source code, so you probably won't have to tell make how to compile or assemble your source file.

Macro CSRCS defines the C source files in the current directory. All of the files listed in it will have lint run on them by a make lint.

One of these two macros is usually defined in terms of the other. For example, in the following makefile text, CSRCS is defined to be the list of .c files corresponding to the .o files in LIB_OBJS, plus mmugen.c.

```
# Object files to go into the library.
LIB_OBJS  =  mmu.o sys.o layer.o

# C source files to be linted.
CSRCS =  ${LIB_OBJS:%.o=%.c} mmugen.c
```

With CSRCS defined this way, only LIB_OBJS needs to be modified for each new C source file. However, if a non-C object file were to be added to the layer's archive, the definition of CSRCS would need to change.

6.6.2 Adding a Module

To add a new module to a layer, first add any source files as described in the previous section. Then, edit file layer.c and add an entry to the array of module_info structures it defines. The entry should contain the name of the module class (a string) and the address of the class's module initialization routine. For example, the following layer.c file defines two modules, sys4c and mmu.

```
#include "types.h"
#include "layer.h"

extern int sys_module_init();
extern int mmu_module_init();

static struct module_info sun4c_mod_info[] = {
        {"sys4c",  sys_module_init},
        {"mmu",    mmu_module_init},
        NULL
};

struct layer sun4c_layer =
        {"Sun4c", NULL, sun4c_mod_info, NULL};
```

The array is terminated by a NULL. Also, notice the extern declarations for the module initialization routines.

The layer structure is generally the only variable defined in a layer.c file that is visible from outside the file. It contains the layer name, a pointer to a command list (in this case, NULL because the layer defines no commands), a pointer to the module_info table, and as the layer initialization entry point (in this case, NULL because the layer requires no special initialization).

6.6.3 Adding a Message Type

Message types are defined in the layer initialization entry point in the layer.c file. For example, the following layer.c file defines a gen_int_pkt message type.

```
#include "types.h"
#include "layer.h"
#include "msgtype.h"
#include "expr.h"
#include "msg_access.h"
#include "gen_int_pkt.h"

static void
computer_init(layer)
        struct layer    *layer;
{
        struct gen_int_pkt *gip = 0;
        gen_int_msgtype = add_msgtype("gen_int_pkt");
        WORD("irl",       &gip->irl      );
        WORD("action",    &gip->action   );
        WORD("irl_valid", &gip->irl_valid);
        WORD("extra",     &gip->extra    );
}

struct layer computer_layer =
        {"Computer", NULL, NULL, computer_init};
```

The `computer_init` routine of this example calls framework routine `add_msgtype` to define the message type, then creates accesses to describe the fields of such a message. For more information on the use of accesses, see Section 3.2, *Use of Accesses*.

6.6.4 Adding a Layer Command

If a layer adds commands (other than module commands) to the simulator's repertoire, it does so through the `layer.c` file, as in this example.

```
#include "types.h"
#include "ui_cmds.h"
#include "layer.h"

extern struct ui_cmd_entry ui_sparc_cmds[];

struct layer sparc_layer =
       {"SPARC", ui_sparc_cmds, NULL, NULL};
```

Typically (as in this case), the actual table and functions describing the commands will be in some other file; only the address of the table is needed in the layer structure.

The table itself is an array of ui_cmd_entry structures, terminated by a NULL. A ui_cmd_entry looks like this.

```
struct ui_cmd_entry {
        char *cmd;            /* command name */
        char *short_helpstr; /* string that explains syntax */
        char *long_helpstr;  /* string that explains command */
        int  (*func)();      /* routine that performs command */
};
```

When a command is typed, the framework looks through the layer structures—starting with the fw layer and working up to the architecture layer—for a table entry whose command name matches the one used in the command. If the framework finds such an entry, it calls the entry point designated by the entry's func field, passing it the rest of the command line (after the command name and any white space) and the "message context" as arguments. The function can use all of the same framework facilities to do its work as module commands, described in Chapter 3, *Writing a Module*. The function returns UI_CMD_IS_DONE or UI_CMD_IS_NOT_DONE, just as does a module command.

The function takes two parameters:

- A string (char*) containing the arguments to the command. For example, if the command was cmd -v a b c, then the first argument to the function would be -v a b c.
- The message context (struct event_cmd_msg*) in which the command was issued. Your function should treat this value as if it were opaque, never dereferencing it to access the structure's members. This context information is needed by certain functions which deal with expressions, such as expr_parse.

Here is an example of a file that sets up two fictitious layer commands, show_widgets and reset_widgets, and a table describing them to be referenced in layer.c:

```
#include "types.h"
#include "ui_cmds.h"
#include "module.h"

static int
show_widgets_cmd(rest_of_line, msg_context)
        char *rest_of_line;
        struct event_cmd_msg *msg_context;
{
        ...
        return UI_CMD_IS_DONE;
}

static int
reset_widgets_cmd(rest_of_line, msg_context)
        char *rest_of_line;
        struct event_cmd_msg *msg_context;
{
        ...
        return UI_CMD_IS_DONE;
}

struct ui_cmd_entry ui_sparc_cmds[] = {
    {
        "show_widgets",
        "show_widgets - print widget information",
        "show_widgets\n\
        This command prints the total number of widgets since\n\
        the last reset_widgets command was issued.",
        show_widgets_cmd
    },
    {
        "reset_widgets",
        "reset_widgets - reset widget information",
        "reset_widgets\n\
        This command resets the widget totals to zero.",
        reset_widgets_cmd
    },
    NULL /* terminate the list of layer commands */
};
```

6.7 Adding a New Layer

Probably the best way to create a new layer is to start with a similar layer and then modify files as necessary to form the new layer linkages and other necessities.

To add a new layer:

1. **Create a new directory for the layer in the same directory as the other layers.**

2. **Copy into it the relevant contents of a similar layer.**

   For example, if you are creating a new architecture layer, you might want to copy the following files from the `sun4c` architecture into the new directory:

   ```
   Makefile Makefile.inc layer.c vers.c
   ```

   If the layer is not an architecture layer, there will be no `vers.c` file to copy.

3. **Modify `Makefile` to list the correct object files in `LIB_OBJS` and the correct C source files in `CSRCS`.**

4. **Architecture makefiles also set `CFLAGS` to define the name of the architecture; change this setting as well.**

5. **Change targets `this` and `this_clean` if the list of products from this directory is not the same as it was for the `sun4c` architecture.**

6. **`Makefile.inc` contains instances of the layer's name (e.g., `sun4c`); change those to the new layer's name.**

7. A line includes the `Makefile.inc` of the layer below this one in the hierarchy described earlier.

   For the `sun4c` version of `Makefile.inc` that line looks like this:

   ```
   include ${ROOT}/sparc/Makefile.inc
   ```

   Change the layer name (here, `sparc`) **to that of the layer which is to be directly below your new layer in the hierarchy.**

   That change establishes the position of your new layer in the tree.

8. **Change `layer.c` as described in the previous section to reflect the modules, commands, message types, etc., of the new layer. Wherever the name of the old architecture appears, change it to the name of the new architecture.**

9. **If the layer represents an architecture, change the architecture name wherever it appears in `vers.c`.**

6.8 How the Framework Knows About the Layers

If something goes wrong in setting up a new layer, it may help to understand what's going on with the makefiles and the layer.c files.

When you run make in a layer directory, the various Makefile.inc files included automatically create macro LAYERS with the names of all the layers in it, in order from the layer corresponding to the current directory down to the fw layer. This list of layers is used to generate flags for the preprocessor, telling it to look in the include directories in those layers for header files, as well as in the current directory.

If the layer directory corresponds to an architecture (remember that this simply means you are creating a version of mpsas in this layer), the makefiles also use the list of layers to create file built_layers.c. As an example, here is the built_layers.c for the sun4c architecture:

```
/* THIS FILE IS GENERATED AUTOMATICALLY */
extern struct layer sun4c_layer, sparc_layer, computer_layer, fw_layer;
struct layer *layers[] = {&sun4c_layer, &sparc_layer, &computer_layer, &fw_layer, 0};
```

This file creates a null-terminated array of pointers to layer structures, where the layer structures themselves are defined in the layer.c files of the various layers. The framework uses this layers array to find out what modules are available in the various layers, and so on. Since the names of the layer structures—for example, sun4c_layer—are generated simply by appending to the name of the layer, it is important that the names in the Makefile.inc and layer.c files for a new layer be set up correctly. Also, layer.o must be in the LIB_OBJS list of every layer's makefile.

Built_layers.o is explicitly listed as a dependency for the mpsas executable target and is explicitly linked into the executable. The list of layers is also used to construct macro LDLIBS, which is a list of the archives to be searched in making executables. The references in built_layers.o to the various layer structures cause all of the layer.o files to be brought into the executable from those archives.

6.9 Makeall

You can use the makeall script in the mpsas root directory to build multiple architectures. This script is useful when you have made changes to a layer that is used by several architectures or when you want to build the entire product from source.

The mpsas root directory must be your current working directory when you run the script. Simply type makeall followed by a list of targets to be built. If you do not specify any targets, the target that appears first in the makefile is built; for layer directories, this target is all.

The script then asks which architectures you want to build. You can enter a list of architecture layer names, separated by spaces, or just press Return to have all (sun4c, sun4e, mbus, and simple) architectures built. The build in each directory is done with X= for efficiency, but it does build the layer directories starting with fw and working up the tree so that any dependencies are up-to-date.

By default, makeall arranges for the C compiler to be invoked with switch -O to produce optimized code. You can override this behavior by setting environment variable CFLAGS0 (makeall sets CFLAGS0 to -O if it is not already set).

The script builds the entire mpsas product but allows only a subset of the architecture layers to be built in order to save time and space. It tries to make the specified targets in all of the mpsas directories it knows about (minus those corresponding to any architectures which you omitted), including those that do not correspond to layers (such as stand).

CHAPTER 7

Asynchronous Input (`sigio`)

The `sigio` facility provides a service that allows modules to independently accept asynchronous input from UNIX file descriptors. If a module needs to perform synchronous (blocking) input to a file descriptor or only needs to output to a file descriptor, it should not use the `sigio` facility; instead, it should use the standard UNIX input/output facilities.

The `sigio` facility consists of the `sigio` module class and a set of framework routines. The `sigio` module uses the UNIX `SIGIO` signal to determine when input is available for a file descriptor.

When input data is available for a file descriptor, the `sigio` module reads the data and sends it in a debug channel message to the module associated with the file descriptor. When a module needs to output data to its file descriptor, the module sends the data to the `sigio` module in a debug channel message. A module must never perform input or output operations directly to a file descriptor that is being used by the `sigio` facility.

7.1 Preparing to Use the `sigio` Facility

A module performs the following steps in preparation for using the `sigio` facility:

1. Ensure that one of the module's interfaces is connected to the instance of the `sigio` module class.

2. Open each file descriptor.

3. Register the file descriptor(s) and interface with `sigio`.

7.1.1 Interface Connected to sigio

The module's interface connected to sigio must register a debug channel receive entry point and be set to QUEUED_MODE. The simulation channel is unused, so no receive entry point for it should be registered.

A module can use one interface connected to sigio to support any number of file descriptors. A module can have more than one interface connected to sigio, but this is not required.

7.1.2 Opening File Descriptors

Each file descriptor used with the sigio facility must be opened with read permission. If a module needs to send output to a file descriptor, it must also be opened with write permission.

7.1.3 Registering File Descriptors

The module's verify config entry point informs the sigio facility of the file descriptors on which sigio is to perform input and output on behalf of the module.

- sigio_set_input_mapping maps a file descriptor to the module's interface connected to sigio for input.
- sigio_set_output_mapping maps a file descriptor to the module's interface connected to sigio for output.

7.2 Communications

A module sends a message to the sigio module to write data to a file descriptor used with the sigio facility. The sigio module sends a message containing data to a module when input is available on the module's file descriptor.

7.2.1 sigio Message Type

The format of the messages sent to and from the sigio module is shown below.

```
struct sigio_msg {
        int fd;          /* File descriptor         */
        short data_size; /* Number of bytes of data */
        char data[1];    /* data being transferred */
};
```

The fd field specifies the file descriptor being read / written. It is used to potentially multiplex more than one file descriptor over a single interface. For incoming data, the sigio module sets fd to the file descriptor that it read. For outgoing data, a module sets fd to the file descriptor it wants to write.

The data_size field specifies the number of bytes to be read / written.

The data field is a variable-length array of characters of data_size bytes. It contains the data read by, or to be written by, sigio.

7.3 sigio Modes

When a file descriptor is registered with the sigio facility for input or output with the sigio_set_input_mapping and sigio_set_output_mapping routines, one of two modes is specified: raw or block.

For output to a file descriptor, raw mode and block mode are functionally equivalent, although block mode output has less overhead. The sigio facility writes data_size bytes from the data array to fd.

7.3.1 Raw Mode Input

If a file descriptor is in raw mode, then sigio sends one message for each character received from the file descriptor.

7.3.2 Block Mode Input

If a file descriptor is in block mode, then sigio reads characters into an internal buffer and sends one message containing the buffer contents when a newline is encountered in the input or when 256 characters have been read.

When a message is sent because a newline was detected, the newline is not placed in the data field of the message. When a message is sent because the buffer is full, only 255 characters from the buffer are placed in the message; the 256th character is sent in the next message. In both cases, the data field of the message is null-terminated so it can be parsed as a string. The terminating null character is included in the data_size field count so that the string length of the data field is one less than the data_size field's value.

Block mode input can only be used to read ASCII data because sigio interprets the input stream. Raw mode input can be used to read binary or ASCII data.

7.4 Flow Control

The framework debug channel message queue has a finite number of messages it can contain (approximately 1,000). If the modules in the system exceed this limit by sending or queueing too many debug messages in a cycle, a fatal error occurs.

Typically, overflowing the debug channel message queue is not a concern of modules because they place very few messages in the debug queue each cycle. However, it is a concern for the sigio module. Remember that the sigio module sends one message per character received if a file descriptor is in raw mode. It is quite possible for the sigio module to overflow the message queue in this case.

To prevent the overflow of the message queue, the sigio module examines the debug queue before it sends a message to a module. If the queue is full, the sending of the message is delayed until the framework has a chance to empty the debug queue.

This overflow prevention implements a crude form of flow control between the sigio module and external devices sending data to it via a file descriptor. If sigio processes the input more slowly than the device creates it, eventually the sending device will block when it sends.

CHAPTER **8**

Access Classes

Chapter 3, *Writing a Module,* showed how the module programmer can tell the framework about module state, thus giving the user the ability to print, set, and dump state variables and to use them in expressions. Each variable is associated with a particular access class, and that access class knows how to print, set, etc., variables of its class. The framework has built into it a number of access classes that cover the common data types, and you can write your own access class to add support for a new type of variable.

Writing an access class is not a trivial task, especially a class for a complicated type. Furthermore, there are limitations to the expressiveness of the access class mechanism. Therefore, it is always wise to consider, when defining a module’s state or a message packet, whether your definition can be readily described by existing access classes. Only consider writing a new access class as a last resort.

If you have some information that you want users to be able to display but none of the existing access classes seem to fit, consider whether you want users to be able to make changes to the information, trace it, and use it in expressions. If you find that you really only care that users can print the information, then you can write a module command to display the information.

In this chapter, we first look briefly at the data structure representing an access class, then look in more detail at what happens when an access is defined and when an expression involving that access is parsed and evaluated. Next, we use an example access class as an introduction to the various entry points constituting an access class. Finally, we tie up some loose ends: how to derive one access class from another, how to write an access class that makes use of access parameters, and how to set up macros that allow accesses to be created with relative ease.

8.1 The ACCESS_CLASS Structure

An access class is represented by a variable (which by convention has a name that begins with AC_) of type ACCESS_CLASS. The ACCESS_CLASS data structure consists largely of pointers to routines (generally referred to as *access class entry points*, or in this document, simply as *entry points*) that the framework calls when it wants to do things to a variable of that class. The structure looks like this.

```
/* The ACCESS_CLASS structure, which defines a class of data. */
typedef struct access_class_ ACCESS_CLASS;
struct access_class_ {
    ACCESS_CLASS*  base;               /* class this one is derived from, or NULL */
    char*          name;               /* e.g. "Word" */
    DATA_TYPE      eval_type;          /* data type evaluator puts in value field */
    unsigned       expr_size;          /* size of an EXPR for this access class */
    void   (*destroy)      (/* ACCESS* */);
    Bool   (*ck_syntax)    (/* ACCESS* */);
    Bool   (*set)          (/* ACCESS*,   ACCESS* value_list */);
    Bool   (*print)        (/* ACCESS*,   void (*  print_p)(), void* arg */);
    Bool   (*dump          (/* ACCESS*,   void (*   dump_p)(), void* arg */);
    Bool   (*restore)      (/* ACCESS*,   void (*restore_p)(), void* arg */);
    void   (*set_base)     (/* ACCESS*,   char* base */);
    Bool   (*as_unsigned)  (/* ACCESS* */);
    Bool   (*as_signed)    (/* ACCESS* */);
    Bool   (*as_float_pt)  (/* ACCESS* */);
    Bool   (*as_string)    (/* ACCESS* */);
    void   (*cleanup)      (/* ACCESS* */);
};
```

The name field is the name of the access class, for example, Word. This is distinct from the name of an access; for example, pc and psr might be the names of two accesses belonging to the Word access class. An access is always an instance of some access class. All the knowledge about how to manipulate a Word is represented in the ACCESS_CLASS data structure for the Word access class, whereas information about the pc variable (for example, where the data is located) is in the pc access. The data structure for an access is of type ACCESS; ACCESS is defined later.

The other fields of the ACCESS_CLASS structure are discussed in upcoming sections as new concepts are introduced. For now, just notice that each of the entry points—destroy, ck_syntax, set, ..., cleanup—takes as its first argument a pointer to the particular access it is to manipulate. Accesses are object oriented in the sense that the framework does not have built-in knowledge of how to print, set, dump, etc., any variables; rather, the framework does the work that is common to all variables regardless of their class and then calls upon the corresponding entry point from the access's access class to do the part of the job that differs from class to class.

8.2 Expressions and Accesses

To write access classes, you need to understand how accesses are used in expressions. This section shows the data structures built when an access is defined and when an expression involving that access is parsed.

8.2.1 Defining an Access

FIGURE 8-1 depicts the arrangement of key data structures after a module instance named `cpu1`, with variable `msp` pointing to a `cpu_state` structure containing its state, defines an access named "pc" by executing the statement

```
    WORD("pc", &msp->pc);
```

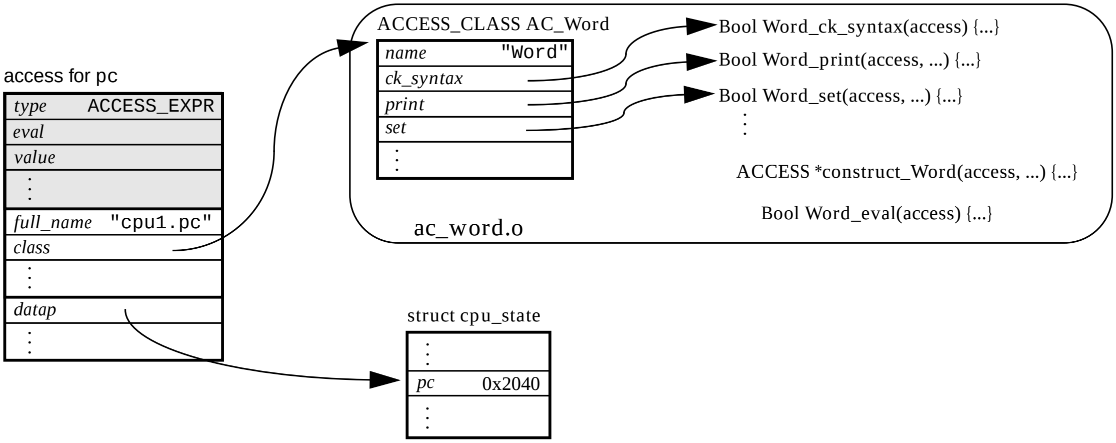
access for pc
type ACCESS_EXPR
eval
value
full_name "cpu1.pc"
class
datap
ACCESS_CLASS AC_Word
name "Word"
ck_syntax
print
set
Bool Word_ck_syntax(access) {...}
Bool Word_print(access, ...) {...}
Bool Word_set(access, ...) {...}
ACCESS *construct_Word(access, ...) {...}
Bool Word_eval(access) {...}
ac_word.o
struct cpu_state
pc 0x2040

FIGURE 8-1 Data Structures After an Access Is Defined

Note – Only those fields that are of interest here are shown.

File `ac_word.o` contains all of the routines to deal with variables of type `Word`, along with an `ACCESS_CLASS` structure called `AC_Word`. The call to the `WORD` macro created an instance of an `ACCESS_common` data structure (labeled “access for pc” in the figure) that contains a pointer (`datap`) to the module instance’s `pc` variable.

The macro first invoked a routine to create a new access with the appropriate name, class, and data type as follows:

```
access_for_pc = access_new_ck("pc", &AC_Word, UNSIGNED);
```

This routine does all of the class-independent initialization of an access. But part of the initialization of an access is class-dependent, so the macro then called the constructor for this access class:

```
construct_Word(access_for_pc, &msp->pc);
```

An access class might have multiple constructors, each taking different arguments to create an access of the same class. For example, there are three constructors for the `Bitfield` class, depending upon whether the pointer and mask correspond to a `Byte`, a `Word`, or an `LWord`; as part of creating an access of class `Bitfield`, you call the constructor appropriate to the particular bit field being described.

The constructor for the `Word` access class simply checks that `datap` is `Word`-aligned and saves it in the `ACCESS_common` structure.

The nature of the `ACCESS_common` data structure merits some explaining. First, the `ACCESS` structure, which is used to describe all accesses, is an extension of a structure, called `EXPR`, that is heavily used in the interface to the expression-handling facilities. That is, an access expression is considered a particular type of expression, so an `ACCESS` consists of an `EXPR`—which contains information common to all expressions—followed by information relevant only to access expressions. The definitions for `EXPR`, `ACCESS`, and `ACCESS_common` look something like this.

```
typedef struct {
        fields needed by framework to describe any expression
} EXPR;

typedef struct {
        EXPR    e;
        additional fields needed by framework to describe an access expression
} ACCESS;

typedef struct {
        ACCESS    a;
        additional fields needed for an access expression of certain common classes;
        this information is not for the framework, but for the access class entry points
} ACCESS_common;
```

The ACCESS structure contains information common to all accesses—information used by the framework—but each access class can extend this structure again to include information specific to that access class. In the case of AC_Word, the extended data structure is called ACCESS_common (because the same structure is used for variables of type Byte, HWord, and others).

The EXPR structure within an access is actually not used (hence the shading in FIGURE 8-1). However, when an access is used in an expression, a copy is made of the access's data structure and in this copy the EXPR part *is* used.

8.2.2 EXPR and ACCESS Data Structures

The definition of an EXPR structure looks something like this.

```
/* An EXPR is a node in an expression tree returned by expr_parse(). */
typedef struct expr_ EXPR;
struct expr_ {
        EXPR*           next;           /* for list of expression trees; NULL=end */
        EXPR_TYPE       type;           /* type of this EXPR */
        DATA_TYPE       value_type;     /* type of resulting value */
        U_VALUE         value;          /* result of the evaluation */
        Bool            (*eval)();      /* function to evaluate this EXPR */
          ...
};
```

Only those fields that you might legitimately need to use are listed. As with all framework data structures, your code should not depend upon the order of these fields.

next links EXPR trees together into a list.

type tells what type of EXPR this is. Depending upon an EXPR's type, it might be possible to cast a pointer to the EXPR into a pointer to some other structure that begins with an EXPR, allowing access to more information. For example, a type of ACCESS_EXPR means that the EXPR is really an ACCESS.

The value field is a union of members to hold an expression's value in various formats; the value_type field tells which member should be used.

The eval field is a pointer to the function that should be called to evaluate the expression.

The definitions for EXPR_TYPE, DATA_TYPE, and U_VALUE follow.

```
/* EXPR_TYPE is an enumeration of the various types of EXPRs. */
typedef enum {
    UNARY_OP, BINARY_OP, QUEST_OP, IN_OP, TO_OP, CHANGES_OP, ASSEMBLE,
    ACCESS_EXPR, VALUE
} EXPR_TYPE;

/* DATA_TYPE is an enumeration of data types understood by the expression parser. */
typedef enum {UNSIGNED=1, SIGNED, FLOAT_PT, STRING} DATA_TYPE;

/* U_VALUE is a union for the value field of an EXPR. */
typedef union {
        LWord    l;   /* 64-bit integer */
        double   d;   /* double */
        char*    s;   /* string */
} U_VALUE;
```

The following is essentially the definition of an ACCESS:

```
#define EXPR_MAX_PARAMS 6
typedef struct {
        EXPR            e;              /* begin with an EXPR because we are one */
         char*            name;               /* e.g. "pc" */
         char*            full_name;          /* e.g. "cpu1.pc" or "bus_pkt.asi" */
         ACCESS_CLASS*    class;              /* class this is an instance of */
         void*            arg;                /* access-specific arg, for whatever */
         unsigned char    params_used;        /* mask of parameters supplied */
         EXPR*            params[EXPR_MAX_PARAMS]; /* the parameters */
          ...
} ACCESS;
```

Note that an ACCESS begins with an EXPR, so that in effect an ACCESS is a specific type of EXPR.

name is the name under which the access was created; full_name is the fully qualified name of the access, with the module instance name or message type name prepended.

class is a pointer to the access class for this access.

arg is a field reserved for use by the module programmer in customizing an access (discussed later).

Some accesses allow parameters, as in ram1.bytes(0x1000,5). When an expression is being parsed, each of these parameters is parsed into its own expression tree. A pointer to each parameter’s tree is placed into the corresponding element of the params array in the access expression, and the corresponding bit is set in params_used to indicate the presence of that parameter.

8.2.3 Expression Parsing

Given this access definition for `pc`, suppose the user types the command

```
when pc == 0x2180
```

The `when` command calls framework routine `expr_parse` to parse the expression text into a tree of `EXPR` structures representing the expression. This tree is kept around; at the end of each cycle, it is evaluated—that is, the value of the expression at that time is ascertained.

FIGURE 8-2 depicts key data structures after the user interface parses the expression used in this command.

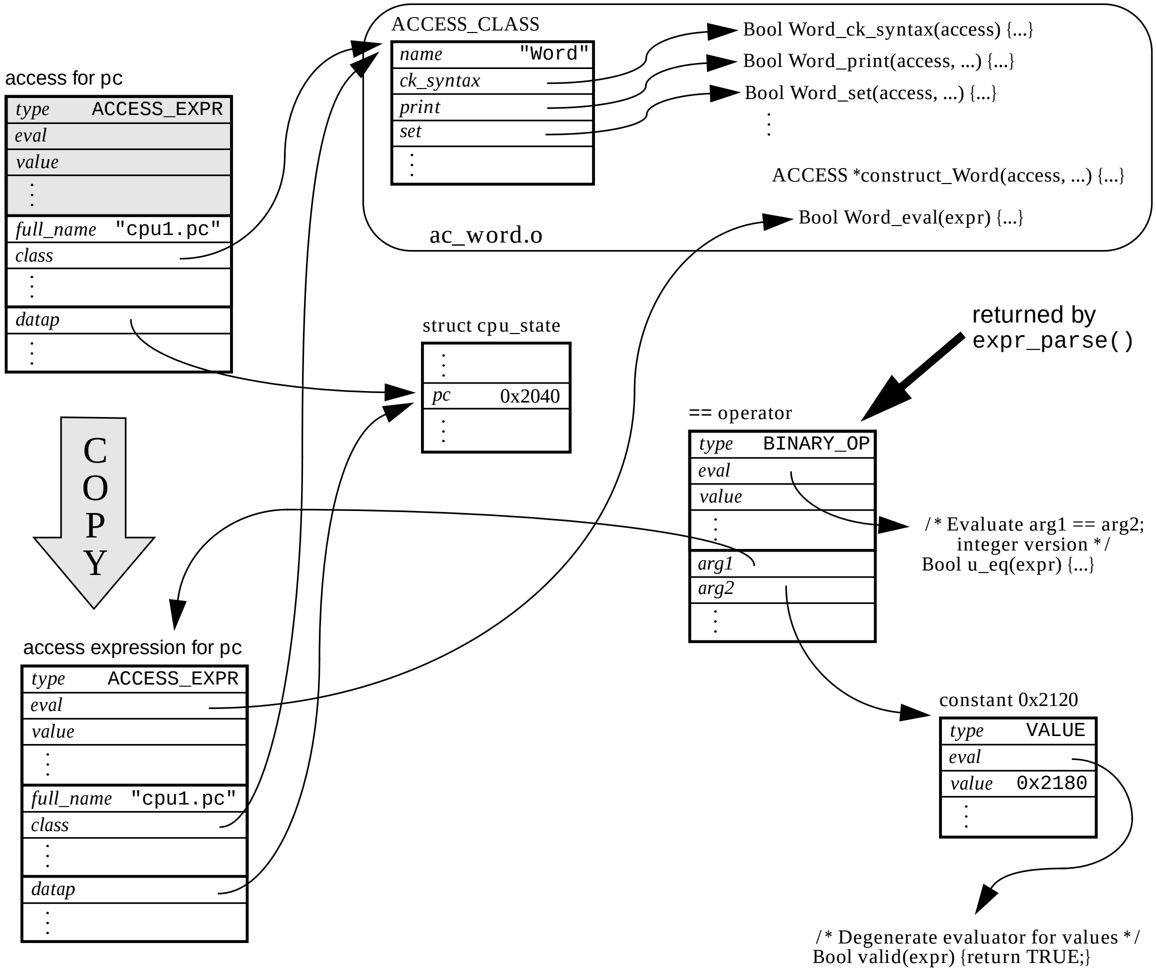

FIGURE 8-2 Data Structures After an Expression Is Parsed

Function expr_parse has returned a pointer to a tree of three EXPR structures. The top structure corresponds to the == operator. Its type is BINARY_OP (binary operator), and its arg1 and arg2 point to a copy of the access for cpu1.pc and an EXPR structure containing the value 0x2180.

Each EXPR structure in the tree has associated with it an evaluator function (pointed to by eval)—the function which should be called to evaluate that node in the tree. The VALUE node's evaluator function simply returns TRUE, indicating that the value already in its value field is valid. The == operator's evaluator is a routine that evaluates its two arguments, compares them as integers, and puts 1 in its value field if they are both valid and have the same value, else 0.

Notice that the access's eval field points to a function defined by the access class. This pointer was set during the parse. After making a copy of the ACCESS (and filling in any parameters to the variable provided in the expression—in this case none), the parser calls the access class's ck_syntax entry point on the copy. This routine determines whether this is a valid use of the access (for example, Word_ck_syntax complains if any parameters were supplied) and then sets the eval pointer to the address of a suitable evaluator function.

The resulting expression tree remains in memory until freed by means of the framework routine expr_free.

8.2.4 Expression Evaluation

The following call evaluates the expression just discussed (where expr is the EXPR pointer returned by expr_parse):

```
expr->eval(expr)
```

This call returns a Bool. If it returns FALSE, the expression is invalid; if it returns TRUE, the expression is valid and its value is in expr->value. The value field is a union of LWord, double, and char* types, so that every data type enumerated in type DATA_TYPE—UNSIGNED, SIGNED, FLOAT_PT, and STRING—can be represented. Expr->value_type is the data type of the expression's value field.

The evaluation proceeds as follows. The call expr->eval(expr) is actually made to the u_eq function, the == evaluator for integers (SIGNED or UNSIGNED). This routine then calls the evaluator for arg1:

```
expr->arg1->eval(expr->arg1)
```

That call is actually made to the Word_eval function, which gets the Word pointed to by datap and puts it in the least significant Word of value (using the LWord union member), and returns TRUE. Since the value field is initialized to 0 when the expression tree is created, this routine is sufficient to set the entire value of the LWord when it is being used as an UNSIGNED.

The u_eq routine sees that arg1 is valid and evaluates arg2 in the same way:

```
expr->arg2->eval(expr->arg2)
```

That call is actually made to the valid function, which returns TRUE to indicate that its result—which is already in the value field—is valid. The u_eq routine sees that arg2 is also valid and compares expr->arg1->value to expr->arg2->value as LWords and sets expr->value to 1 if they are equal, else 0. Finally, u_eq finishes by returning TRUE.

8.2.5 Access Expressions

One might reasonably wonder why the ACCESS data structure needs to be copied when an expression tree is created. That is, why can't the expression tree point directly to the ACCESS data structure built by the WORD macro?

The primary reason is that access expressions can involve parameters. For example, the ram module creates a bytes variable for each of its module instances, and bytes can take several parameters. If the user enters the following commands, we need to keep around two expression trees involving the one access, each with different parameters:

```
when ram1.bytes(0x4038,32) changes
when ram1.bytes(0x3140) > 3
```

Another way to solve the problem would have been to put the parameters in a different data structure, but expression evaluation is more efficient with just one structure.

To be precise, copied ACCESS data structures resulting from the use of accesses in expressions are called *access expressions*. Structurally, of course, they are identical to accesses, except that more of the fields are used. Where the distinction is obvious or unimportant, this manual often refers to an access expression as an access.

8.2.6 Multiple Evaluators

One might also ask why the ACCESS_CLASS structure does not contain a pointer to the evaluator function, since it contains pointers to so many other functions defined for the class. After all, if it did, then the framework could take care of setting the eval pointer of the access expression.

The answer is that there can be more than one evaluator function for an access class. In access classes that use parameters, it is often possible for the ck_syntax entry point to look at the way parameters are used and choose an evaluator peculiar to that use of parameters. This approach can save considerable expression evaluation time over the alternative of having one evaluator that always examines the way parameters are used and then executes the appropriate code. Remember that the ck_syntax entry point is only called when the expression is parsed, whereas the evaluator function is called each time the resulting expression tree is evaluated—therefore, it makes sense to do as much of the work in the ck_syntax entry point as you can.

8.3 An Example Access Class

In this section, a simple access class illustrates how to write the various parts of an access class. This access class is for variables of type Word. It is not the implementation for AC_Word, since AC_Word is created from a special template file that allows several access classes to share code, thereby obscuring what is happening. Functionally, however, the two are identical. This new implementation is called Word2, and the ACCESS_CLASS structure is AC_Word2.

File ac_Word2.h defines the extended ACCESS data structure that Word2 accesses use.

Note – Files ac_Word2.h and ac_Word2.c are not included with MPSAS. They are merely teaching tools for this chapter rather than code actually used in the simulator as delivered.

```
/*
 * ac_Word2.h -- Header file for access class used in programmer’s guide.
 */

/* Declare the extended ACCESS structure for Word2 accesses. */
typedef struct {
        ACCESS  a;      /* stuff common to all accesses */
        Word*   where;  /* address or offset of data given to constructor */
        Word*   datap;  /* where, or where + base if set_base() called */
} ACCESS_Word2;
```

This data structure can be safely cast to either an ACCESS* or an EXPR*, since it begins with an ACCESS and an ACCESS begins with an EXPR. The larger structures can be thought of as having been derived from the smaller structures, inheriting all of the members of the parent structure and adding their own.

It is a good idea to put this structure definition in a file separate from the access class implementation itself; if you later decide to derive another access class from this one, the new class will need the definition for the old class’s structure.

The datap field is the one actually used to retrieve the data. The purpose of the where field is discussed later.

8.3.1 Access Constructors

A constructor for an access class is a routine called to initialize a new access of that class. A constructor typically takes arguments that are stored in the class-specific part of the extended ACCESS structure. There can be more than one constructor for a class if there are different contexts in which such an access might be used. For any particular access, only one constructor should be called to initialize the access. Following is the beginning of the file containing the Word2 access class, showing its only constructor.

```
/*
 * ac_Word2.c -- Access class used in programmer’s guide; for Word variables.
 */

#include "types.h"
#include "expr.h"
#include "access_class.h"
#include "ac_Word2.h"

/*
 * This is the constructor for Word2 accesses; it gets called whenever a
 * new Word2 access is defined. It does NOT get called when an access is
 * copied by the parser.
 */
extern ACCESS*
construct_Word2(access, where)
        ACCESS*         access; /* ptr to the access, from access_new() */
      Word*        where;  /* ptr/offset to the Word to which to provide access */
{
        ACCESS_Word2*   it = (ACCESS_Word2*)access;

        /* Make sure the data pointer is on an appropriate boundary. */
        if ((unsigned long)where & (sizeof(Word)-1))
                fatal( "%s %s access data pointer misaligned\n",
                       access->class->name, access->name );

        it->where = it->datap = where;
        return access;
}
```

This constructor accepts a pointer to the Word variable as a parameter. All this constructor does is ensure that the data pointer is Word-aligned and assign it to the where and datap fields. In some cases, this value is actually an offset, but for now think of it as a pointer to the data.

Notice that the constructor receives an ACCESS* and casts it to an ACCESS_Word2*, assigning the value to a variable called it. The it variable is used much like the this keyword of C++ (the latter name was avoided in order to simplify a potential future port to C++) to point to the data structure corresponding to an instance of the class.

The constructors for a class are generally its only externally defined functions. All other functions of the class are called by the framework through the ACCESS_CLASS structure or through pointers set by such functions, so they might be static.

Notice that the constructor used the framework routine fatal to display its error. Constructors (except those for framework access classes, which can be used with the var command) can only be called during the simulator's initialization, and any error represents an error on the part of the programmer.

8.3.2 The ck_syntax Entry Point

The ck_syntax entry point is called during the parsing of an expression involving an access of this class.

```
/*
 * ck_syntax() checks the syntax of a Word2 access. Called by the expression
 * parser, it is responsible for syntax-checking (number and types of
 * parameters); the return is TRUE if all is OK. It also sets the eval ptr.
 */
static Bool
Word2_ck_syntax(access)
        ACCESS*         access;
{
        ACCESS_Word2*   it = (ACCESS_Word2*)access;

        /* Make sure there are no parameters. */
        if (access->params_used) {
                fwprintf("usage: %s   (no parameters)\n", access->full_name);
                return FALSE;
        }

        access->e.eval = Word2_eval; /* set up evaluator function */
        return TRUE;
}
```

This routine checks that the parameters are all being used correctly; in this case, that means ensuring that none were used at all. The routine simply checks that the mask of supplied parameters, params_used, is 0.

The ck_syntax entry point checks on structures created in response to the user's typing, so errors are to be expected. Hence, an error results in a message to the user and a return value of FALSE to indicate failure.

If the parameters are all right, the ck_syntax entry point chooses an evaluator routine (the Word2 class only has one evaluator function) and puts its address in the eval field of the access expression's EXPR part. Finally, the routine returns TRUE to indicate to the parser that the access expression is a proper one.

Remember that all of this is happening to a *copy* of the access. Only the constructor touches the original access; the other routines work with an access expression involving that access.

8.3.3 Evaluator Functions

When it comes time to evaluate the expression (remember that the expression, once parsed, can be evaluated repeatedly), the eval function is called, using the pointer set by the ck_syntax entry point. The evaluator function's job is to place in expr->value the current value of the variable represented by the access expression. Here is the evaluator for Word2 accesses.

```
/*
 * eval() evaluates a Word2 access.
 */
static Bool
Word2_eval(expr)
        EXPR*           expr;
{
        WORD_VALUE(expr) = *((ACCESS_Word2*)expr)->datap;
        return TRUE;
}
```

In this case, evaluation simply dereferences the data pointer and places the Word thus obtained in the appropriate part of the EXPR structure. The WORD_VALUE macro accesses the least significant Word of the value field (taken as an LWord). Then, the evaluator function returns TRUE to signify that the result is valid. There is no way that an access of this class can ever be invalid, but if an evaluator function decides that the variable represented by the access expression is invalid (as it might, for example, if the variable depended on a pointer that was currently NULL), the evaluator function signifies this by returning FALSE.

You might be wondering why this routine does not begin with a line like

```
ACCESS_Word2* it = (ACCESS_Word2*)access;
```

as the constructor did, and then just dereference it->datap. It could, but with most compilers that solution would actually result in additional overhead that the cast avoids, and evaluators are the only routines in access classes for which performance is particularly important.

8.3.4 The set Entry Point

The set entry point is called when the user uses the set command to set an access expression involving an access of that class, such as

```
set cpu1.pc = 0x3128
```

Some access expressions can be set to multiple values. For example, you can set five consecutive words of physical memory with a command like

```
set ram1.words(0xfe00) = 3, 4, cpu1.pc+1, -9, 3.1416
```

So, the arguments to the set entry point of an access class are:

- A pointer to the access expression being set
- A pointer to the first of a linked list of expression trees

The expression trees have already been evaluated by the time the access's set entry point is called, so the top node of each tree contains the value of interest. There is never any need to walk the tree or even to look at the node type: the value_type and value fields are all that matter. Also, the values are guaranteed to be valid; if any are invalid, the set entry point is not called.

The set entry point must take the list of value expressions and assign them—whatever that means to the access class—to the variable described by the access expression. Here is the set entry point for the Word2 access class.

```
#define MANTISSA_MASK ((Word)~0 >> 1)
static Bool
Word2_set(access, value_list)
        ACCESS*         access;
        EXPR*           value_list;
{
        ACCESS_Word2*   it  = (ACCESS_Word2*)access;
        Word            hi  = VALUE_HI_WORD(value_list);
        Word            lo  = WORD_VALUE(value_list);
        Word            val;

        /* Make sure the types are compatible and the data fits. */
        switch (value_list->value_type) {
        case UNSIGNED:
                if (hi)
                        return ac_err_too_big(access, value_list);
                val = lo;
                break;
        case SIGNED:
                /* Make sure all bits of hi word match lo word’s sign bit. */
                if (hi != ((s_Word)lo >> (BITS_PER_WORD-1)))
                        return ac_err_too_big(access, value_list);
                val = lo;
                break;
        case FLOAT_PT: /* assumes floats are IEEE single-precision float pt */
                *(float*)&val = DOUBLE_VALUE(value_list);
                break;
        default:
                return ac_err_set_type(access, value_list);
        }

        /* See if the "change" routine approves. */
        if (access->change && !access->change(access, it->datap, &val))
                return FALSE;

        /* Set the thing. */
        *it->datap = val;

        /* Complain if there was more than one value in the list. */
        if (value_list->next)
               ac_warn_set_extra(access);

        return TRUE;
}
```

This routine first looks at the data type of the first value in the list. If it is not UNSIGNED, SIGNED, or FLOAT_PT, then it is not a valid value to assign to this type of access. Currently, only STRING values are excluded, but writing it this way prepares for the possibility of additional data types being created in the future. If one were to use a string, as in

```
set cpu1.pc = "zort"
```

framework routine ac_err_set_type would display an appropriate message and return FALSE, which this routine would then return. The return of FALSE indicates that the set failed.

If the value is an UNSIGNED integer, the routine checks to make sure that it will fit into a Word and then saves it in val. Remember that an integer expression evaluates to an LWord; therefore, such size checks are necessary to make sure that values will fit into the variable the access is describing. If the value does not fit, framework routine ac_err_too_big displays an appropriate message and returns FALSE, which this routine then returns.

A similar check is done for SIGNED integers. Here, though, the check is that the upper Word of the value is just an extension of the sign bit of the lower Word.

No check is performed for floating-point numbers. Note that the value could change, since the value is of type double and we are converting it to a float so it will fit in a Word, but the semantics of IEEE floating-point assignment ensure that a reasonable value will be assigned in every case.

Having put the value in val, the routine next checks (using routine access_change_func) whether a change routine has been defined for the access and if so, calls it with pointers to the old and new values. If the change routine returns FALSE, the set is aborted. The change routine is responsible for printing out an appropriate error message if it rejects the new value.

If the need for change routines is unclear, bear in mind that a set entry point applies to all accesses of a given class. It is the change routine that enforces range restrictions, etc., on a particular access of that class.

Notice that the change routine is not given any information about the type of the original value. It should not need that information. From that original value, the bit pattern in val was created, and either that bit pattern is valid or it isn’t. Where it came from is irrelevant.

If the change routine approves the new value (by returning TRUE), the next step is to actually change the Word pointed to by datap. Remember that the change routine is allowed to change the new value, so it is important that the new value whose address was passed to the change routine be used in the assignment.

Next, this routine checks for additional values. Since a Word can only hold one value, this code calls framework routine ac_warn_set_extra to display a standard warning if the user tries to set the variable to more than one value:

```
set cpu1.pc = 1, 2
```

The value_list argument is actually a linked list of expression trees; value_list->next is the next tree in the list, or NULL.

Finally, the routine returns TRUE to indicate that the set succeeded.

8.3.5 The Print Entry Point

An access class's print entry point is called whenever the user uses the print command to print an access variable of that class. Its job is to print an appropriate human-readable representation of the variable's current value, *without* a trailing newline. Here is the print entry point for the Word2 class.

```
/*
 * print() prints a Word2 in hex or floating point.
 */
static Bool
Word2_print(access, print_p, arg)
        ACCESS*         access;
        void            (*print_p)();
        void*           arg;
{
        ACCESS_Word2*   it = (ACCESS_Word2*)access;
        char            buf[80];

        if (access->e.value_type == FLOAT_PT)
                print_p(arg, dtoa(buf, DOUBLE_VALUE(it)));
        else
                print_p(arg, "0x%08x", WORD_VALUE(it));
        return TRUE;
}
```

Argument print_p is a pointer to a function that should be used for printing. arg is an argument that must be passed to print_p. The rest of the arguments to print_p are the same as those to printf—a format string followed by values to be used with the conversion specifiers of the format string.

Remember that an access class can handle more than one data type. The Word2 class handles accesses of type UNSIGNED, SIGNED, and FLOAT_PT, although the code that specifies this has not yet been shown. In a class that handles more than one data type, the print entry point will probably need to do different things depending upon the access's data type. Here SIGNED and UNSIGNED are handled by the same code, but FLOAT_PT must be handled separately. Assuming you have set up the class correctly, the framework will not create an access of the wrong type for a given class, so there is no check for the STRING type here as there was with the value expression of the set entry point. The routine returns TRUE to indicate that it was successful.

Notice that the value being printed is not obtained by dereferencing datap, but rather is the value field of the EXPR part of the access's data structure (macros DOUBLE_VALUE and WORD_VALUE use the value field). This works because the framework always evaluates an access expression before calling the print entry point (or set, dump, or restore routines). If the access expression is invalid, the print (or whichever) entry point will not be called.

Notice that for floating-point accesses, the value field is being used as a double. This usage might seem incorrect, since the evaluator function only set the lower Word of the value field—clearly, the Word the access describes is a float, not a double. But you will see later that the value field is always converted to the form expected by expression operators, which for floating-point values is double and for integers is LWord. It might look as though the print entry point is behaving inconsistently, because for integers it only uses the lower Word, but of course the lower Word would not change in converting from a Word to an LWord, and the print entry point is only concerned with printing the lower Word.

Another reason evaluation is done prior to calling these entry points is that if the access expression involves parameters, those parameters need to have been evaluated by the time these routines are called. It is the evaluator function that evaluates any parameters used in the access expression.

8.3.6 The dump and restore Entry Points

The framework calls an access class's dump entry point when the dump command is executed. The dump entry point should dump a compact representation (generally something close to its internal representation) of an access expression, using a routine supplied for that purpose. The restore entry point restores it. Here are the dump and restore routines for the Word2 class.

```
/*
 * dump() dumps a Word2 out to a file.
 */
static Bool
Word2_dump(access, dump_p, arg)
        ACCESS*         access;
        Bool            (*dump_p)();
        void*           arg;
{
        ACCESS_Word2*   it = (ACCESS_Word2*)access;
        return dump_p(arg, it->datap, sizeof(Word));
}

/*
 * restore() restores a previously dumped Word2 from a file.
 */
static Bool
Word2_restore(access, restore_p, arg)
        ACCESS*         access;
        Bool            (*restore_p)();
        void*           arg;
{
        ACCESS_Word2*   it = (ACCESS_Word2*)access;
        return restore_p(arg, it->datap, sizeof(Word));
}
```

Arguments dump_p and restore_p are pointers to functions that do the actual dumping and restoring. arg is an argument that must be passed to those functions. The other two arguments to each of those functions are a pointer to the data area and the length of the data area in bytes. Those functions return TRUE if they succeed, else FALSE. Word2_dump and Word2_restore simply pass the return code on.

In the current release of MPSAS, the restore function is not used. For compatibility with future releases, however, it is included in the ACCESS_CLASS structure.

8.3.7 The set_base Routine

If an access is used to describe a field of a message type or if the access has an address generator function associated with it (the association having been created by a call to access_address_generator), then the data described by the access might be in a different place each time an expression involving that access is evaluated. The framework uses an access class's set_base entry point during each evaluation to tell the access expression where the data is now located.

It is not required that an access class have a set_base entry point. If an access class does not have a set_base entry point, then it is a fatal error to try to create an access of that class to describe a message field or to associate with such an access an address generator function. If you are writing a general-purpose access class, however, it is probably possible to write a set_base entry point and worth the effort. Here is the set_base entry point for the Word2 class.

```
/*
 * set_base() changes the base pointer for the access's datum.
 */
static void
Word2_set_base(access, base)
        ACCESS*         access;
        char*           base;
{
        ACCESS_Word2*   it = (ACCESS_Word2*)access;
        it->datap  =  (Word*)(base + (unsigned long)it->where);
}
```

Recall that the constructor set both the where and datap fields of the extended access data structure to the address of the Word. If the access is not one that moves around, these fields remain the same forever. But if the set_base entry point is called, datap is set to the sum of the new base address passed in through Word2_set_base and the value passed to the Word2 constructor as the location of the datum. The constructor stored this location in the where field. When set_base is used, where is assumed to be not the address of the data, but rather an offset to it from some base address, which is the value provided in the call to set_base.

For example, consider the following code, taken from the computer layer's initialization entry point. It is part of the code that uses accesses to describe the gen_int_pkt message type.

```
        struct gen_int_pkt *gip = 0;
          ...
        gen_int_msgtype = add_msgtype("gen_int_pkt");
        WORD("irl",       &gip->irl      );
        WORD("action",    &gip->action   );
        WORD("irl_valid", &gip->irl_valid);
        WORD("extra",     &gip->extra    );
          ...
```

Because gip is initialized to 0, expressions such as &gip->irl evaluate to the offsets of various fields from the beginning of the packet. When an expression involving one of the fields defined here is parsed by expr_parse, the framework makes the following adjustments in the ACCESS structure to ensure that the access is relocated to the appropriate field in the message being transmitted at the time the expression is evaluated:

1. It sets base_eval to eval, the pointer to the evaluator function as set by the ck_syntax entry point.
2. It sets eval to point to a special routine within the framework.

The latter routine is called when the expression is evaluated; it does three things.

1. It makes sure the message being transmitted is of the same message type as the one used in the expression; if not, it returns FALSE to indicate that the expression is invalid.
2. It calls the access's set_base entry point with the address of the current message packet.
3. It calls the access's real evaluator through field base_eval.

In other words, when expr_parse sees that the access referenced in an expression is a message field, it arranges for a wrapper to be called instead of the normal evaluator routine. The layer moves the access to the right spot and then calls the normal evaluator.

FIGURE 8-3 depicts the relationship between the where and datap fields in a Word2 access when used in this way. This usage of the two fields is typical of framework access classes.

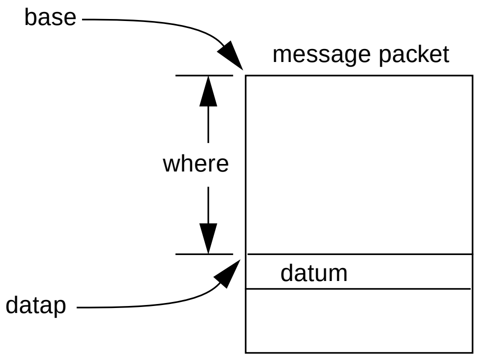

FIGURE 8-3 Handling of Message Fields by Word2 Access Class

8.3.8 The Access Class Variable

The last part of the access class is an instance of the ACCESS_CLASS data structure—the one that represents that access class. Here it is for the Word2 class.

```
 /*
 * Here is the actual ACCESS_CLASS data structure for Word2 accesses.
 */
ACCESS_CLASS AC_Word2 = { _, "Word2", UNSIGNED, sizeof(ACCESS_Word2),
        _,              Word2_ck_syntax,  Word2_set,            Word2_print,
                       Word2_dump,     Word2_restore,    Word2_set_base,
        _,              ac_sex_Word,      ac_float_to_double,   CANNOT_PROMOTE,
        _
};
```

Note – Underscore (_) is a macro defined to be 0. By convention it is not used in place of 0 used as a count, but rather where 0 means “I don’t want/have one of those.”

The first underscore is in the base field, indicating that this class is not derived from any other class. This field is discussed in more detail in a later section. The next field is name, a string that will be displayed in certain messages and list command output. The eval_type field, here UNSIGNED, is the data type returned by the class’s evaluator functions. The framework uses this field to determine whether it needs to do any additional work to get the data type called for in an expression.

The fourth field, expr_size, is the size of the extended access data structure for this class, in bytes. This field is used by routine access_new to allocate the access structure when an access of this class is created.

The next field is destroy and the destructor entry point for the class. This field is optional, and in fact very few classes will need destructors. A destructor is called when an access of that class is deleted (using access_delete). A destructor receives as an argument a pointer to the access being deleted and has a void return. The destructor’s job is to release any resources allocated by the access.

Note – The destructor deallocates resources held by the access itself, not resources held by access expressions referring to that access.

The next few fields are pointers to the ck_syntax, set, print, dump, restore, and set_base entry points, which have already been discussed.

The next four fields are used in conjunction with the eval_type field. They are pointers to functions that convert from whatever is returned by the evaluator functions to the various expression data types (as enumerated in DATA_TYPE), in the formats required by the expression evaluator. Such a conversion is called a *promotion*. Here, as_unsigned is an underscore, which means that no conversion is required (eval_type is UNSIGNED). The fact that as_signed is given the value ac_sex_Word means that if the value of the access is to be used as a signed integer, ac_sex_Word (which sign-extends the Word to an LWord) should be called on the access rather than the evaluator specified by the ck_syntax entry point. If the value is to be interpreted as a floating-point number, then the framework should call ac_float_to_double (which converts the float value to double). The entry for as_string is CANNOT_PROMOTE, which is a special value which indicates that the promotion is impossible. Therefore, if one were to try to use an access of this class as a string, an error would result.

Four promoter functions are built into the framework:

- ac_sex_Byte — Sign-extends a Byte to 64 bits
- ac_sex_HWord — Sign-extends an HWord to 64 bits
- ac_sex_Word — Sign-extends a Word to 64 bits
- ac_float_to_double — Converts a float to double

If you need to write your own promoter function, it should look like this:

```
Bool
promoter_func_name(access)
    ACCESS *access;
{
        if (!access->promote_eval(access)) /* evaluate the access */
             return FALSE; /* if it’s invalid, so are we */
        promote value field, in place
        return TRUE if result is valid, else FALSE
}
```

The promoter function is called *instead of* the routine specified by the ck_syntax entry point (that is, access->e.eval points to the promoter function); therefore, the first thing the promoter function must do is evaluate the access normally. A pointer to the routine to do this evaluation has been saved by the framework in access->promote_eval. It, like the promoter function itself, follows the protocol for evaluator functions: receive a pointer to the access, and return TRUE if the value is invalid, else FALSE.

Note that no promoter function is provided for conversion to UNSIGNED because EXPR structures are initialized to zeroes. That is, the upper Word of the value LWord is already zero, so if the access expression is to be used as an unsigned value, then all that is required is for the evaluator function to set the lower Word. Since the vast majority of variables are unsigned integers, the savings obtained by not having to call a function to make an LWord out of the access expression's value is substantial.

The last field in the ACCESS_CLASS structure is the cleanup entry point, here an underscore. This is another field you will probably never need to use. It is a pointer to a routine to be called when an access expression (the copy of an access produced when an access is used in an expression) is freed via expr_free. The field takes a pointer to the access as an argument and has a void return. Its job is to release any resources allocated by the ck_syntax entry point for this access expression.

8.4 Derived Access Classes

If you create a new access class by using or adding onto the access data structure of some other class, you might want to set the base pointer to point to the ACCESS_CLASS structure of the class whose access data structure you borrowed. That way, the access_isa and access_class_isa framework functions will identify the new class as being derived from the old (base) class.

Creating new classes by deriving them from old classes is strongly encouraged.

If a new access class is being derived just for use within a particular module, the module init entry point for that module is probably the best place to create the new class. If its use will span modules, the new class is probably best defined within the layer initialization entry point for the lowest layer (the one closest to the framework) that will use that class. As an example, the following code might be used to create a new access class called Word2_addr, derived from Word2, which differs from its base class only in the way it prints a Word (using routine Word2_addr_print, not shown).

```
ACCESS_CLASS AC_Word2_addr;
           ...
        AC_Word2_addr = AC_Word2;
        AC_Word2_addr->base = &AC_Word2;
        AC_Word2_addr->name = "Word2_addr";
        AC_Word2_addr->print = Word2_addr_print;
           ...
```

8.4.1 Existing Access Classes

TABLE 8-1 shows the framework access classes from which you might want to derive your own classes. There are other access classes, but they are less general or more dangerous to inherit from, so you should not derive new classes from them. The table lists for each class the name of the class's ACCESS_CLASS variable, the name of the extended ACCESS data structure used by the class, and the access-creating macros from common_access.h which use the class.

TABLE 8-1 Framework Access Classes from Which New Classes Can Be Derived

Class Variable	ACCESS Type	Macros That Use It
AC_Bool	ACCESS_common	BOOL
AC_HWord	ACCESS_common	HWord, S_HWORD
AC_Word	ACCESS_common	Word, S_WORD
AC_LWord	ACCESS_common	LWord, S_LWORD
AC_bit_field	ACCESS_bit_field	BITF, S_BITF, BBITF, S_BBITF, LBITF, S_LBITF, MEMBER_BITF, MEMBER_LBITF
AC_string	ACCESS_common	STR
AC_group	ACCESS_group	GROUP

8.5 Parameters

The Word2 access class demonstrated much of what goes into writing an access class. One thing it did not demonstrate is the handling of parameters.

You can expect the use of parameters to substantially complicate an access class. Here are some of the complications introduced.

- The access class's ck_syntax entry point must check the parameters supplied and their types.
- For efficiency, you might want to have more than one evaluator function: one for the case where all the parameters are constants, for example. In addition, the access class's ck_syntax entry point will have to make this determination and assign the appropriate evaluator's address to the access->e.eval field.
- Depending upon what the parameter means, the access class's print, set, dump, and restore entry points might need to refer to its value in doing their jobs.

Some of the access classes used with the ram and rom modules understand several different uses of parameters. For example, the word access can be used in these ways to select different parts of the module instance's memory:

- word — All Words
- word(*addr*) — The Word containing the Byte at address *addr*
- word(*addr,num*) — number of Words starting with the one containing the Byte at address *addr*
- word(*addr1,,addr2*) — All Words from the one containing the Byte at address *addr1* to the one containing the Byte at address *addr2*, inclusive

The following code is from the ck_syntax entry point of an access class used with the ram and rom modules; it checks that an access expression fits one of the above patterns and that any parameters supplied evaluate to integers.

```
        if ( PARAMS_ABOVE(it, 2) ||
             (!PARAM_USED(it, 0) && PARAMS_ABOVE(it, 0)) ||
             (PARAM_USED(it, 1) && PARAM_USED(it, 2)) )
                goto usage_err;
        for (i=0; i<3; i++)
           if (PARAM_USED(it, i) && !EXPR_IS_INTEGER(access->params[i]))
                        goto usage_err;
          ...
usage_err:
        fwprintf("usage: %s[(start[,num])] or %s(start,,end)\n",
            access->full_name, access->full_name);
        return FALSE;
```

The PARAMS_ABOVE macro returns TRUE if in the specified access expression any parameters of higher number than that specified were used. Parameters are numbered left to right from 0 to EXPR_MAX_PARAMS-1 (currently 5). So PARAMS_ABOVE(it,2) is TRUE for variable expression word(,,,1) but FALSE for word(,,1).

The PARAM_USED macro returns TRUE if in the specified access expression the specified parameter number was used. The second line of the conditional checks that if any parameters were used, parameter 0 was used. The third line of the conditional checks that parameters 1 and 2 were not both used, as they are in word(1,1,1).

The loop that follows checks parameters 0 to 2 (the only ones which could have been used) to see that if they were used, they are of integer type. Macro EXPR_IS_INTEGER returns TRUE if the value_type field of the specified access expression is either UNSIGNED or SIGNED.

8.6 Macros for Creating Accesses

Given an access class like Word2, the following things need to be done to describe a module state variable of that class:

- Create a new access of that class and of the appropriate type.
- Call a constructor for that class with the appropriate arguments.
- Add the access to the module's list of accesses.

The same procedure holds for describing a message field, except that the access is added to a different list—the list of fields for the most recently defined message type. Since modules typically create many accesses, macros such as WORD hide these steps. You might want to create similar macros for access classes you write.

For the Word2 class, one might want to create macros WORD2, s_WORD2, and FLOAT2, which create UNSIGNED, SIGNED, and FLOAT_PT Word2 accesses, respectively. They could be implemented as follows.

```
/* Macros to create unsigned and signed Word2 accesses, e.g. WORD2("pc", &msp->pc); */
#define   WORD2(name,datap) ADD_ACCESS(  WORD2_ACCESS(name,datap))
#define S_WORD2(name,datap) ADD_ACCESS(S_WORD2_ACCESS(name,datap))
#define  FLOAT2(name,datap) ADD_ACCESS( FLOAT2_ACCESS(name,datap))
#define   WORD2_ACCESS(name,datap) \
                           construct_Word2(access_new_ck(name,&AC_Word2,UNSIGNED),datap)
#define S_WORD2_ACCESS(name,datap) \
                           construct_Word2(access_new_ck(name,&AC_Word2,  SIGNED),datap)
#define  FLOAT2_ACCESS(name,datap) \
                           construct_Word2(access_new_ck(name,&AC_Word2,FLOAT_PT),datap)
```

These macros, like those in fw/include/common_access.h, are each broken into two parts. WORD2_ACCESS calls access_new_ck with the name of the access, a pointer to the access class, and the data type of the access. Access_new_ck creates the access, does any class-independent initialization on it, and returns a pointer to it. That pointer and a pointer to the actual Word for this access are passed to construct_Word2, the constructor for the Word2 class, which does the class-dependent initialization.

But WORD2_ACCESS just creates the access without putting it on a list of accesses for a module instance or message type. The WORD2 macro is built on top of WORD2_ACCESS, taking the access pointer returned by it and passing it to the ADD_ACCESS macro. ADD_ACCESS is defined by state_access.h to be add_state, a routine that adds an access to the list of accesses for the currently initializing module instance. If you are defining a message type, then instead of including state_access.h, you include msg_access.h, which defines ADD_ACCESS to be add_field, a routine that adds an access to the list of fields for the most recently added message type.

The same division of labor exists between S_WORD2 and S_WORD2_ACCESS, and between FLOAT2 and FLOAT2_ACCESS.

Note – Different constructors might require different arguments, and therefore different access-creating macros will require different arguments.

APPENDIX **A**

Framework Manual Pages

This chapter contains detailed descriptions of the global variables and routines provided by the framework for use by the module programmer. These descriptions are in a form similar to that of the UNIX manual pages.

Name

cmd_result, cmd_result_double, SET_CMD_RESULT_AS_WORD, fw_terminate_cmd — Global variables useful to user interface commands

Synopsis

```
#include "types.h"

LWord cmd_result;

double cmd_result_double;

Word SET_CMD_RESULT_AS_WORD(return_code)
Word return_code;

Bool fw_terminate_cmd;
```

Description

The user interface maintains variables called cmd_result and cmd_result_double. User Interface commands set the variables by setting the global variables cmd_result and cmd_result_double.

SET_CMD_RESULT_AS_WORD() sets cmd_result to the 32-bit value specified by *return_code*.

Fw_terminate_cmd is a read-only Boolean variable that is set to true by the framework when a user types Control-C during the execution of a user interface command. User interface commands that take some time to execute (for example, generate large amounts of output) should check this variable periodically to allow the user to interrupt the command. It is reset to false at the start of each user interface command.

Name

`cyclecount`, `instrcount`, `opt_simpleprint` — Miscellaneous framework global variables

Synopsis

```
int instrcount;

int cyclecount;

int opt_simpleprint;
```

Description

The global variable `instrcount` corresponds to the user interface variable of the same name. Each time a processor executes an instruction, it should increment this variable. `instrcount` is used by the framework in the `time` command to calculate the instructions-per-second value.

The global variable `cyclecount` also corresponds to a user interface variable of the same name. It is incremented by the framework each cycle and should be considered read-only by the modules. The framework also uses this value in the `time` command to calculate the cycles-per-second value. Modules can use `cyclecount` to obtain the number of cycles simulated.

The `opt_simpleprint` global variable contains the value of the `simpleprint` option. For details, see the manual page for the `option` command, in Appendix B of the *Multiprocessor SPARC Architecture Simulator (MPSAS) User's Guide*. If you plan to parse the simulator's output in another program, you might want your code to generate output that is more easily parsed when this variable is `true`.

Name

ac_sex_Byte, ac_sex_HWord, ac_sex_Word, ac_float_to_double, ac_err_set_type, ac_err_too_big, ac_warn_set_extra, PARAM_USED, PARAMS_ABOVE, access_new_ck, access_new, add_state, add_field — Routines used in writing access classes

Synopsis

```
#include "types.h"
#include "expr.h"
#include "access_class.h"

Bool ac_sex_Byte( ACCESS *access)
Bool ac_sex_HWord(ACCESS *access)
Bool ac_sex_Word( ACCESS *access)
Bool ac_float_to_double(ACCESS *access)

Bool ac_err_set_type(ACCESS *access, EXPR *expr)
Bool ac_err_too_big( ACCESS *access, EXPR *expr)

void ac_warn_set_extra(ACCESS *access)

Bool   PARAM_USED(ACCESS *access, unsigned param_num)
Bool PARAMS_ABOVE(ACCESS *access, unsigned param_num)

ACCESS *access_new_ck(char *name, ACCESS_CLASS *class,
                                              DATA_TYPE datatype)
ACCESS *access_new(   char *name, ACCESS_CLASS *class,
                                              DATA_TYPE datatype)

#include "module.h"
ACCESS *add_state(ACCESS *access)

#include "msgtype.h"
ACCESS *add_field(ACCESS *access)
```

Description

These functions are useful in developing new access classes. An access class is essentially a template from which accesses can be instantiated. For details see Chapter 8, *Access Classes*.

ac_sex_Byte(), ac_sex_HWord(), and ac_sex_Word() sign-extend Byte, HWord, and Word values to s_LWords for use in expression evaluation. Although accesses can represent data of any size and format, there is a well-defined size and format for each data type to be used in an expression; an access class must provide for such conversions, called *promotions*. Similarly, a float value must be promoted to a double before it can be used in an expression, and that is just what ac_float_to_double() does.

The rest of the routines are useful primarily in writing the set entry point for an access class. Each reports a particular type of error in a standard way; hence, use of these routines gives a common feel to all accesses.

ac_err_set_type() displays the following error message:

```
Cannot set access’s name (type datatype) to datatype value
```

The routine always returns false, so it can be used as follows within the set routine (which indicates a problem by returning false):

```
if (access cannot be set to expr’s value type)
                return ac_err_set_type(access, expr);
```

ac_err_too_big() displays the following error message and returns false:

```
Value doesn’t fit in access’s classname access’s name
value is expr’s value
```

ac_warn_set_extra() displays the following warning message:

```
Extra values ignored in set of access’s name
```

PARAM_USED() returns true if the specified parameter is used in access expression *access*, else false. PARAMS_ABOVE() returns true if any of the parameters above *param_num* are used in *access*. Parameters are numbered 0 through EXPR_MAX_PARAMS-1 (currently 5).

If you create a new access class, you may want to create new macros for it like those in fw/include/common_access.h. Common_access.h is included by state_access.h and msg_access.h, which are included by code using accesses to describe module state variables or message types, respectively. The macros you write should use the following routines to do their work.

access_new_ck() creates a new access with the specified *name*, *class*, and *datatype* and returns a pointer to it. You must call a constructor for the class to ready the access for use. access_new_ck() is actually a wrapper around access_new(), which has the same arguments. access_new() returns NULL if it cannot create the specified access, and the wrapper simply causes a fatal error in that event.

After the constructor has been called, you usually want the access placed on a list of variables to be found later by the expression parser. add_state() adds *access* to the list of variables for the currently initializing module instance. Add_field() adds *access* to the list of fields for the message type currently being described (see `add_msgtype()`, discussed in the *Message Type* manual page). Both return *access*. Errors in either are fatal.

SEE ALSO

Access Create
Access Control
Access Misc
Expressions
Message Types

Name

FLOAT, DOUBLE, BOOL, BYTE, S_BYTE, CHAR, HWORD, S_HWORD, WORD, S_WORD, LWORD, S_LWORD, WORD_ADDR, LWORD_ADDR, STR, BBITF, S_BBITF, BITF, S_BITF, LBITF, S_LBITF, GROUP, ARRAY, FLOAT_ACCESS, DOUBLE_ACCESS, BOOL_ACCESS, BYTE_ACCESS, S_BYTE_ACCESS, CHAR_ACCESS, HWORD_ACCESS, S_HWORD_ACCESS, WORD_ACCESS, S_WORD_ACCESS, LWORD_ACCESS, S_LWORD_ACCESS, WORD_ADDR_ACCESS, LWORD_ADDR_ACCESS, STR_ACCESS, BBITF_ACCESS, S_BBITF_ACCESS, BITF_ACCESS, S_BITF_ACCESS, LBITF_ACCESS, S_LBITF_ACCESS, GROUP_ACCESS, ARRAY_ACCESS, MEMBER_BITF, MEMBER_LBITF — Macros to create accesses

Synopsis

```
#include "types.h"
#include "expr.h"
#include "state_access.h" or "msg_access.h"

/* macros to create accesses and add to appropriate list */
ACCESS      *FLOAT(char *name, float   *datap)
ACCESS     *DOUBLE(char *name, double  *datap)
ACCESS       *BOOL(char *name, Bool    *datap)
ACCESS       *BYTE(char *name, Byte    *datap)
ACCESS     *S_BYTE(char *name, s_Byte  *datap)
ACCESS       *CHAR(char *name, char    *datap)
ACCESS      *HWORD(char *name, HWord   *datap)
ACCESS    *S_HWORD(char *name, s_HWord *datap)
ACCESS       *WORD(char *name, Word    *datap)
ACCESS     *S_WORD(char *name, s_Word  *datap)
ACCESS      *LWORD(char *name, LWord   *datap)
ACCESS    *S_LWORD(char *name, s_LWord *datap)
ACCESS  *WORD_ADDR(char *name, Word    *datap)
ACCESS *LWORD_ADDR(char *name, LWord   *datap)
ACCESS        *STR(char *name,  char    **datap)

ACCESS   *BBITF(char *name, Byte  *datap, Byte  mask)
ACCESS *S_BBITF(char *name, Byte  *datap, Byte  mask)
ACCESS    *BITF(char *name, Word  *datap, Word  mask)
ACCESS  *S_BITF(char *name, Word  *datap, Word  mask)
ACCESS   *LBITF(char *name, LWord *datap, LWord mask)
ACCESS *S_LBITF(char  *name, LWord *datap, LWord mask)

ACCESS *GROUP(char *name, Bool (*eval)())
ACCESS *ARRAY(ACCESS *access, char *name,
                         unsigned num_elements, unsigned spacing,
                         void (*flex_func)(), void *flex_arg)
```

```
/* macros to create accesses without adding them to a list */
ACCESS      *FLOAT_ACCESS(char *name, float   *datap)
ACCESS     *DOUBLE_ACCESS(char *name, double  *datap)
ACCESS       *BOOL_ACCESS(char *name, Bool    *datap)
ACCESS       *BYTE_ACCESS(char *name, Byte    *datap)
ACCESS     *S_BYTE_ACCESS(char *name, s_Byte  *datap)
ACCESS       *CHAR_ACCESS(char *name, char    *datap)
ACCESS      *HWORD_ACCESS(char *name, HWord   *datap)
ACCESS    *S_HWORD_ACCESS(char *name, s_HWord *datap)
ACCESS       *WORD_ACCESS(char *name, Word    *datap)
ACCESS     *S_WORD_ACCESS(char *name, s_Word  *datap)
ACCESS      *LWORD_ACCESS(char *name, LWord   *datap)
ACCESS    *S_LWORD_ACCESS(char *name, s_LWord *datap)
ACCESS  *WORD_ADDR_ACCESS(char *name, Word    *datap)
ACCESS *LWORD_ADDR_ACCESS(char *name, LWord   *datap)
ACCESS        *STR_ACCESS(char *name, char    **datap)

ACCESS   *BBITF_ACCESS(char *name, Byte  *datap, Byte  mask)
ACCESS *S_BBITF_ACCESS(char *name, Byte  *datap, Byte  mask)
ACCESS    *BITF_ACCESS(char *name, Word  *datap, Word  mask)
ACCESS  *S_BITF_ACCESS(char *name, Word  *datap, Word  mask)
ACCESS   *LBITF_ACCESS(char *name, LWord *datap, LWord mask)
ACCESS *S_LBITF_ACCESS(char *name, LWord *datap, LWord mask)

ACCESS *GROUP_ACCESS(char *name, Bool (*eval)())
ACCESS *ARRAY_ACCESS(ACCESS *access, char *name,
                         unsigned num_elements, unsigned spacing,
                         void (*flex_func)(), void *flex_arg)

/* macros to create a bit field as a member of something else */
ACCESS  *MEMBER_BITF(ACCESS *access, char *name, Word  mask)
ACCESS *MEMBER_LBITF(ACCESS *access, char *name, LWord mask)
```

Description

These macros create accesses. An access is a construct that gives the framework enough information about a variable to allow the user to print it, set it, use it in expressions, dump it to a trace file, and so on. The use of accesses is discussed in Section 3.2, *Use of Accesses*.

A module instance uses accesses to describe parts of its internal state. In this case, include `state_access.h`. A common place for such code is in the module's create instance entry point; however, if the allocation or use of a variable depends upon the number and kind of interfaces configured for the module instance, it may be desirable to create certain accesses later in the configuration phase, for example, in the verify config entry point.

Accesses are used to describe the fields of a message type. In this case, include `msg_access.h`. Description of message types is typically done in a layer's initialization routine.

Note – You may *not* include both state_access.h and msg_access.h in a single file.

Macros FLOAT(), DOUBLE(), …, ARRAY() define accesses and put them where they can be found by the expression-handling code in the framework, thus making them available for use in the user interface as described above. These, along with MEMBER_BITF() and MEMBER_LBITF(), are the macros you will generally use to create accesses. Macros FLOAT_ACCESS(), DOUBLE_ACCESS(), …, ARRAY_ACCESS() are identical to their counterparts in the previous set except that they do not make the accesses available to the user. Rather, an access created with one of these macros is usable only by the handle it returns, which can be used in further calls. These macros are generally only used when describing arrays, to create the template for an element of the array.

TABLE A-1 lists the type of data handled by each macro.

TABLE A-1 Macrodata Described by Access

Macro	Data Type
ARRAY	multiple repetitions of some other access
BBITF	unsigned bit field within a Byte
BITF	unsigned bit field within a Word
BOOL	variable of type Bool
BYTE	variable of type Byte
CHAR	variable of type char
DOUBLE	variable of type double
FLOAT	variable of type float
GROUP	arbitrary collection of other accesses
HWORD	variable of type HWord
LBITF	unsigned bit field within an LWord
LWORD	variable of type LWord
LWORD_ADDR	variable of type LWord containing addresses
MEMBER_BITF	unsigned bit field as part of a Byte or Word access
MEMBER_LBITF	unsigned bit field as part of an LWord access
S_BBITF	signed bit field within a Byte
S_BITF	signed bit field within a Word
S_BYTE	variable of type s_Byte

TABLE A-1 Macrodata Described by Access *(Continued)*

Macro	Data Type
S_HWORD	variable of type s_HWord
S_LBITF	signed bit field within an LWord
S_LWORD	variable of type s_LWord
S_WORD	variable of type s_Word
STR	variable of type char* (null-terminated string)
WORD	variable of type Word
WORD_ADDR	variable of type Word containing addresses

The user's view of a piece of module instance state or a message field, through an access, is called a *variable*. Each of the preceding macros takes a *name* argument, which is the name by which the variable will be known to the user. If the access describes module instance state, the full name of the access is actually *module_instance.name*. The user need not type the full name of such an access when focused on the module instance for which it is defined. If the access describes a message field, its full name is actually *message_type.name*, where *message_type* is the name used in the call to add_msgtype(). The user must always type the full name of an access describing a message field.

Most of the macros take an additional *datap* argument that is a pointer to the datum. Those macros which define bit fields take a *mask* argument that isolates the part of the Byte, Word, or LWord of interest. *mask* may not have holes in it; that is, it must be a set of contiguous 1 bits. The *mask* of an LWord bit field may not span both words of the LWord.

Usually, a bit field is part of some larger entity that might be of interest as a whole. MEMBER_BITF() creates a bit field access as a member of *access*, such that when *access* is printed you will see all of its members. When using MEMBER_BITF(), the name field is taken to be an extension to the name rather than the name itself. For example, consider the following code, which might be used to create an access for a SPARC processor's PSR field and member accesses describing the PSR's S and CWP fields:

```
access = WORD("psr", &state->psr);
MEMBER_BITF(access, "s", S_MASK);
MEMBER_BITF(access, "cwp", CWP_MASK);
```

The full name of the variable corresponding to the CWP would be *module_instance*.psr.cwp.

For MEMBER_BITF(), *access* can refer to either a Byte or a Word (or some class derived from Byte or Word); *mask* should correspond to that type. MEMBER_LBITF() is like MEMBER_BITF() except that *access* should refer to an LWord and *mask* should be an LWord.

`GROUP()` creates an access that is a collection of other accesses. It is not possible to set the value of a group. When a group is printed, all of its members are printed (as is true of any access that has members). When the group is dumped or restored, all of its members are dumped or restored. You can supply your own evaluator function (see Chapter 8, *Access Classes*) for a group as *eval*. Use `NULL` for this argument to obtain the default evaluator, which yields an invalid result if any member of the group is invalid or otherwise yields the bitwise exclusive-or of the values of all of its members. Members are added to a group by the `access_add_member()` and `state_members()` function (see the *Access Control* manual page) if the group describes module instance state; groups are generally not used with message fields.

`ARRAY()` creates an access that describes an array of *num_elements* elements, using *access* as a template for an element of the array. The elements are spaced regularly, *spacing* bytes apart. You can create arrays whose *datap*, *num_elements*, or *spacing* vary over time by providing a function *flex_func* that is called immediately before the evaluate routine is called. An argument to *flex_func* can be supplied as *flex_arg*. For arrays fixed in location, size, and spacing, use `NULL` for *flex_func*.

A flex function should look like this:

```
void flex_func_name(array)
    ACCESS_array *array;
{
    set array->num, array->spacing, array->base2 as needed
}
```

The `ACCESS_array` structure is defined as follows:

```
typedef struct {
    ACCESS     a;
    ACCESS*    element;    /* ptr to access we’re an array of */
    void       (*flex_func)(); /* see above */
    void*      flex_extra; /* for use by flex_func() */
    unsigned   num;        /* number of elements in array */
    unsigned   spacing;    /* number of bytes between elements */
    char*      base1;      /* base, as set by set_base() */
    char*      base2;      /* base, as set by set_base() and
                              perhaps modified by flex_func() */
    char*      element_base; /* base2 + index * spacing */
    unsigned   index;      /* for change routine */
} ACCESS_array;
```

The flex_extra field is initialized by the constructor to flex_arg. The flex function can set any or all of the num, spacing, and base2 fields. base2 is the pointer used by the array access class as the base of the array when it calculates element_base, the address of a particular element. When a change function is called, the index field is the index of the element being set.

The use of *name* in the user interface refers to the entire array. For example, printing *name* will show the values of all its elements. You can set *name* to a list of values, as in

```
set my_values = 87, 31, 23
```

The last value is duplicated as many times as necessary to fill the array. When used in an expression, for example, my_values changes, an array evaluates to the bitwise exclusive-or of all of its elements' values.

You refer to a particular element of the array by putting the index (an expression) in parentheses as a parameter, as in

```
print my_value(cpu1.l0+2)
```

To create an array access, first create some other access representing an element of the array. Use the version of the macro that does *not* make the access available to the user. The name you choose for the access is irrelevant. Then, use the ARRAY() macro to create the array. For example, the following code creates an array of 20 contiguous Words, called my_words:

```
access = WORD_ACCESS("", &state->my_words);
ARRAY(access, "my_words", 20, sizeof(Word), NULL, NULL);
```

While it would be perfectly valid to create a group with arrays as members, it would not be valid to create an array using a group as a template for the element (since there is no address associated with a group).

The RO() macro is just shorthand for the access_read_only() function, described in the *Access Control* manual page.

When accesses are dumped, the format they are dumped in depends upon the class of the access. In most cases, the format used is the same as the internal representation for the data; for example, a FLOAT variable is dumped in C float format.

When a string is dumped, the length is first written to the file as a long. If the string is a NULL pointer, -1 is written for the length. Next, the characters of the string—not including the null terminator—are written. Obviously, there are no characters to write if the string is a NULL pointer or the empty string.

Bit fields of no more bits than a Byte are dumped as Bytes to conserve space; the rest are dumped as Words. Note that this optimization does not depend upon the value of the bit field at any point in time, but rather upon the mask used to define it.

When a group is dumped, it dumps its members.

If an array element is dumped, it dumps like an ordinary access of that type. When an entire array is dumped, each element is dumped like an ordinary access of that type; however, if the array has a flex function associated with it, the number of elements is dumped as an `unsigned` before the elements are dumped.

Bugs

When creating an array element template that has members, you must use a macro that makes the element template available to the user (if you didn't, you couldn't give it members). You can at least hide the element template by using `access_hide()`.

The array mechanism does not blend seamlessly with the members mechanism. Suppose you have created an access `bar` and given it members `a` and `b`. If you create an array `foo` using `bar` as a template, you might think that you could write expressions like `foo(3).a` and `foo(5).b`, but in fact these expressions are not allowed. Printing `foo(3)` will indeed print `a` and `b` for `foo(3)`, but you may not refer to the members directly.

This capability can be simulated, however, by also creating arrays called `foo.a` and `foo.b`, each with an appropriate element. This technique would make possible expressions like `foo.a(3)`.

See Also

Access Control
Access Class
Access Misc
Expressions
Message Types

Name

`access_read_only`, `access_hide`, `access_add_member`, `state_members`, `access_compact_print`, `access_invalid_print`, `access_address_generator`, `access_change_func`, `access_arg`, `access_custom_set`, `access_custom_print`, `access_custom_dump`, `access_custom_restore` — Change the behavior of accesses

Synopsis

```
#include "types.h"
#include "expr.h"

ACCESS *access_read_only(ACCESS *access)

ACCESS *access_hide(ACCESS *access)

Bool access_add_member(ACCESS *access, EXPR *member)
void state_members(ACCESS *access, char *member_name, ... NULL)

ACCESS *access_compact_print(ACCESS *access)

void access_invalid_print(ACCESS *access, Bool (*func)())

void access_address_generator(ACCESS *access, char *(*func)())

void access_change_func(ACCESS *access, Bool (*func)())
void access_arg(ACCESS *access, void *arg)

void access_custom_set(    ACCESS *access, Bool (*func)())
void access_custom_print(  ACCESS *access, Bool (*func)())
void access_custom_dump(   ACCESS *access, Bool (*func)())
void access_custom_restore(ACCESS *access, Bool (*func)())
```

Description

These functions provide fine control over certain aspects of the behavior of accesses.

`access_read_only()` makes it impossible for the user to change the value of the variable described by *access*. That is, the `set` command gives an error if such an access is the target of the set.

`access_hide()` prevents *access* from showing up in the `list` command's list of variables defined for a module instance. However, it will still show up in `list -v` output.

`access_add_member()` adds access expression *member* to the list of members for *access*. Member must be the result of the `expr_parse()` of an expression consisting of a variable and its parameters. When the user directs the simulator to print *access*, its members will be printed instead.

If you are describing module state, if you are sure that the member access exists, and if no parameters to the member need to be specified, then `state_members()` provides a more convenient interface than `access_add_member()`. For each *member_name* provided (a `NULL` marks the end of the names), an access expression corresponding to *module_instance.access_name.member_name* is added to the list of members for *access*. It is a fatal error if no such access exists. Note, however, that macros `MEMBER_BITF()` and `MEMBER_LBITF()` provide even more convenient interfaces for describing bit fields as members of another access.

`access_compact_print()` causes the `print` command to show the members of *access* all on one line rather than on separate lines. An example of this is

```
ui1: print cpu1.tbr
tba=0x0 tt=0x0
```

`access_invalid_print()` causes the `print` command to call *func* when *access* is invalid rather than just printing a message saying that the variable is invalid. See the *Access Class* manual page for a description of the interfaces to the print function.

`access_address_generator()` causes *func* to be called whenever an expression is evaluated which involves *access*. The expression evaluator uses the address returned by *func* as the base address of the variable; hence, with this feature you can describe a variable that moves around in memory. The address generator function should look like this:.

```
char *addr_gen_func(access)
    ACCESS *access;
{
    ...
    return current_address_of_datum;
}
```

`access_change_func()` causes *func* to be called whenever the user tries to set *access*. *func* approves or rejects the set with its return code and may even change the actual value assigned. The change function should look like this.

```
Bool change_func(access, old, new)
    ACCESS *access;
    void   *old;
    void   *new;
{
    if new points to a bad value {
        print an error message
        return FALSE;
    }
    perhaps change the value pointed to by new
    return TRUE;
}
```

Note that `old` and `new` are of type `void*`, so you will need to cast them to the appropriate pointer type before using them.

`access_custom_print()`, `access_custom_set()`, `access_custom_dump()`, and `access_custom_restore()` arrange for *func* to be called to print, set, dump, or restore *access* instead of the routine provided by *access*'s class. See the *Access Class* manual page for a description of the interfaces to these class functions.

`access_arg()` causes expressions involving *access* to have *arg* available in the `ACCESS` structure as `access->arg`. *arg* can be anything the module programmer deems useful to any of the access's functions written by the module programmer.

See Also

Access Create
Access Class
Access Misc
Expressions

Name

`access_valid`, `access_isa`, `access_class_isa` — Miscellaneous access-related routines

Synopsis

```
#include "types.h"
#include "expr.h"

Bool access_valid(ACCESS *access)

Bool access_isa(ACCESS *access, ACCESS_CLASS *class)
Bool access_class_isa(ACCESS_CLASS *class1, ACCESS_CLASS *class2)
```

Description

`access_valid()` evaluates *access* and returns `true` if it is valid. If it is not valid, `access_valid` prints an appropriate error message and returns `false`.

`access_isa()` returns `true` if *access* is of class *class* or is of any class derived from *class*.

`access_class_isa()` returns `true` if *class1* is the same as *class2* or if *class1* is derived from *class2*.

See Also

Access Create
Access Control
Access Class
Expressions

Name

assemble, print_disassembly — Assemble / disassemble instructions

Synopsis

```
#include “types.h”

Bool (*assemble)(str, addr, instr_p)
char *str;
LWord addr;
Word *instr_p;

Bool (*print_disassembly)(prev, inst, next, addr)
Word prev, inst, next;
LWord addr;
```

Description

These functions provide support for dealing with instructions as assembly language rather than machine code. They are actually pointers to functions, because the framework only provides the interface to this capability (the pointer) and not the implementation (the function); it is expected that some other layer will set these pointers to appropriately written routines. In the architectures delivered with MPSAS, the `sparc` layer does this. All calls to those routines, however, should be made through this framework interface.

`assemble()` sets the `Word` pointed to by *instr_p* to the numeric instruction corresponding to the assembly language instruction in *str* and returns `true`. If there is an error in the assembly language, `false` is returned. The instruction is assembled as though it were to end up in simulated memory at address *addr*. Although *addr* is an `LWord`, only the least significant `Word` of it is used.

`print_disassembly()` displays the assembly language equivalent of the numeric instruction *inst* and returns `true`. If the numeric instruction represents a badly formed instruction, `false` is returned. If you provide the instructions on either side of that one in *prev* and *next* and the instruction is part of a sethi / add sequence used to set the 32-bit value of a register, the address shown in the disassembly will be more accurate. However, the disassembly can be done using 0 for *prev* and *next*. *addr* is the address at which *inst* appears; it is needed to properly interpret instructions containing offsets relative to the program counter. Although *addr* is an `LWord`, only the least significant `Word` of it is used.

Name

get_config_mod_instance_name, get_num_interfaces, get_num_interfaces_by_type, get_module_extra, set_module_extra, get_object_ptr, get_object_size — Get and set module, module instance, interface information, and shared object information during configuration phase

Synopsis

```
#include "module.h"
#include "interface.h"

char *get_config_mod_instance_name()
int get_num_interfaces()

int get_num_interfaces_by_type(type_name)
char *type_name;

caddr_t get_module_extra()

void *get_object_ptr()
int   get_object_size()

void set_module_extra(extra)
caddr_t extra;
```

Description

These routines return information about instances, interfaces, and shared objects during the configuration phase. If any of the routines are called outside the configuration phase, the functions will exit by calling `fatal()`.

These routines can be called by all the module's configuration entry points except the module init entry point. `set_module_extra()` is an exception and must be called only from a module init entry point. `get_object_ptr()` and `get_object_size()` should only be called from the lookup shared object entry point.

During the configuration phase, `get_config_mod_instance_name()` returns a string pointer to the module instance name currently being configured. The module should only read this string; it must not overwrite it.

During the configuration phase, `get_num_interfaces()` returns the number of interfaces of the module instance currently being configured during the initialization phase.

During the configuration phase, `get_num_interfaces_by_type()` returns the number of interfaces of type *type_name* for the module instance currently being initialized.

During the configuration phase, `get_module_extra()` returns the value of the `extra` field of the module currently being initialized.

`get_module_extra()` can be called from a module's init entry point to set the module's `extra` field to the opaque value *extra*.

During the configuration phase, `get_object_ptr()` gets the pointer to the object being shared.

During the configuration phase, `get_object_size()` returns the size of the object being shared, in bytes.

Name

dtoa, ctoa, ctoa_hex, atoc — Convert between ASCII and other formats

Synopsis

```
#include "types.h"

char *dtoa(char *buf, double d)

char *ctoa(char *buf, char c, char quote)

char *ctoa_hex(char *buf, char c)

char *atoc(char *buf, char *cp, char quote)
```

Description

These functions convert internal data representations to their representation in the C programming language, and vice versa. They are useful when accepting character or string input from the user or when displaying characters, strings, or floating-point values for the user to see.

dtoa() puts into *buf* the ASCII C source representation of double *d*. You (as the programmer) are responsible for making sure that *buf* is large enough. The number of mantissa digits shown is specified through the option float_precision; because the maximum setting for float_precision is 500, you would need a buffer of 508 bytes to guarantee that the buffer would not be overrun.

ctoa() puts into *buf* the ASCII C source representation of char *c*. No quotes are placed in *buf*. This representation is always in one of these forms:

- c The character itself
- \c Special notation for certain control characters: \n \t \b \r \f \v
- \ddd Three octal digits, for example, \001, for other control characters
- \c Backslash is also used to escape metacharacters: \\ \\quote

quote should be either a single quote or a double quote; if *c* matches *quote*, it is preceded by a backslash.

ctoa_hex() is the same as ctoa() except that the \ddd form is replaced with the value represented as two hexadecimal digits and no quote character is used. The output from ctoa_hex() is limited to two characters.

atoc() puts at **cp* the character corresponding to the ASCII C source representation of a character in *buf* and returns the address within *buf* following the character's representation. The following are allowable forms for representing a character.

- c Any character except *quote*
- \c Special notation for certain control characters: \n \t \b \r \f \v
- \ddd One to three octal digits, for example, \001
- \c Backslash in front of any other character means use it literally

quote should be either a single quote or a double quote; it indicates which of the two marks the end of the string of character representations. If *buf* does not start with a legal representation for a character or if the first character in *buf* matches *quote*, `NULL` is returned. *buf* should not contain the opening quote, but should contain the closing quote; before the closing quote it may contain the representations of any number of characters.

The following example uses `atoc()` to convert a string obtained from the user (including quotes) to its internal representation.

```
char   user_input[80], *from = user_input+1;
char internal_rep[80], *to   = internal_rep;
gets(user_input);
if (user_input[0] != ’"’)
    error("string is missing opening quote");
while (*from != ’"’) {
    from = atoc(from, to++, ’"’);
    if (!from)
        error("bad character representation in string");
}
if (from[1])
    error("garbage after closing quote");
*to = ’\0’; /* terminate the internal representation */
```

Name

data_type_name, data_type_value — Convert between data type name and value

Synopsis

```
#include “types.h”

char *data_type_name(type)
DATA_TYPE type;

DATA_TYPE data_type_value(name)
char *name;
```

Description

These functions convert back and forth between a DATA_TYPE and its name. The defined data types (symbols defined for enum DATA_TYPE) and their corresponding names are listed in TABLE A-2.

TABLE A-2 Defined Data Types

Enum Symbol	Name
UNSIGNED	unsigned
SIGNED	signed
FLOAT_PT	float
STRING	string

If a type other than those listed is passed to data_type_name(), it returns the string "<invalid type!>".

If a name other than those listed is passed to data_type_value(), it returns 0.

Name

`dump_array_ptr`, `dump_buffer`, `dump_intf`, `dump_message`, `dump_string` — Utilities to help a module dump portions of its state to a file

Synopsis

```
int dump_array_ptr(stream, array_base, element_size, ptr)
FILE *stream;
char *array_base;
u_int element_size;
char *ptr;
```

```
int dump_buffer(stream, buf, size)
FILE *stream;
char *buf;
int size;
```

```
int dump_intf(stream, intf)
FILE *stream;
caddr_t intf;
```

```
int dump_message(stream, data, type, size, delay)
FILE *stream;
caddr_t data;
caddr_t type;
int size;
int delay;
```

```
int dump_str(stream, str)
FILE *stream;
char *str;
```

Description

These dump state routines work in conjunction with the restore state routines to provide facilities to dump data to, and restore data from, a stream.

`dump_array_ptr()` converts the pointer *ptr* into its equivalent integer index and then writes it to *stream*. The *array_base* parameter points to the base of the array. The *element_size* parameter is the size of each array element in bytes. The *ptr* parameter may be `NULL`. The corresponding routine to read this information from a stream is `restore_array_ptr()`.

`dump_buffer()` writes *size* bytes from buffer *buf* into *stream*. If *buf* is `NULL`, *size* is ignored. *Size* must be nonnegative. The corresponding routine to read this buffer from a stream is `restore_buffer()`.

`dump_intf()` dumps interface handle *intf* to *stream*. It handles a `NULL` *intf* handle. The corresponding routine to read the interface handle from a stream is `restore_intf()`.

`dump_message()` writes a message to *stream*. *data* is a pointer to the message's data area, or `NULL` if there is no data with the message. The *type* parameter must be a valid message type, and *size* is the size of *data* in bytes (which can be `0`). The corresponding routine to read the message from a stream is `restore_message()`.

`dump_string()` writes the null-terminated string *str* to *stream*. It handles `NULL` string pointers and zero-length strings. The corresponding routine to read the string from a stream is `restore_string()`.

Return Values

On success, all dump state routines return `0`; on failure, they return `1`.

See Also

Restore State

Name

expr_parse, expr_set_msg_context, expr_free, expr_boolean, expr_access, expr_get_LWord, expr_get_Word, expr_eval_boolean, expr_show, expr_show_value, expr_equiv, EXPR_IS_INTEGER, EXPR_IS_ACCESS, BYTE_VALUE, HWORD_VALUE, WORD_VALUE, LWORD_VALUE, FLOAT_VALUE, DOUBLE_VALUE, STRING_VALUE, VALUE_HI_WORD — Parse and handle expressions

Synopsis

```
#include "types.h"
#include "expr.h"

EXPR *expr_parse(char *str, unsigned options, char *rest_p)

void expr_set_msg_context(struct event_cmd_msg *context)

void expr_free(EXPR *expr)

EXPR *expr_boolean(EXPR *expr)
EXPR *expr_access( EXPR *expr)

Bool expr_get_LWord(char *str, unsigned options, char *rest_p,
                                                   lword_p)
Bool expr_get_Word( char *str, unsigned options, char *rest_p,
                                                    word_p)

Bool expr_eval_boolean(EXPR *expr)

void expr_show(      EXPR *expr, (*print_p)(), void *arg)
Bool expr_show_value(EXPR *expr, (*print_p)(), void *arg)

void capture_printf(CAPTURE_INFO *capture, printf-style args)
Bool init_capture_info(CAPTURE_INFO *capture)

Bool expr_equiv(EXPR *expr1, EXPR *expr2)

Bool EXPR_IS_INTEGER(EXPR *expr)
Bool EXPR_IS_ACCESS( EXPR *expr)

Byte     BYTE_VALUE(EXPR *expr)
HWord   HWORD_VALUE(EXPR *expr)
Word     WORD_VALUE(EXPR *expr)
LWord   LWORD_VALUE(EXPR *expr)
float   FLOAT_VALUE(EXPR *expr)
double DOUBLE_VALUE(EXPR *expr)
char  *STRING_VALUE(EXPR *expr)

Word  VALUE_HI_WORD(EXPR *expr)
```

Description

These functions enable you to easily deal with expressions following the syntax defined in the *User Interface* chapter of *Multiprocessor SPARC Architecture Simulator (MPSAS) User’s Guide.*

`expr_parse()` parses an expression from the beginning of *str* and returns the corresponding expression tree. If there is an error in the expression, an appropriate message is printed and `NULL` is returned. *options* controls the behavior of the parse. Currently, the only choices for *options* are 0 and `EXPR_OPT_MSG`; the latter allows the use of message types and message fields within the expression (normally an error). If *rest_p* is `NULL`, it is considered an error if the entire string will not parse as a valid expression. If *rest_p* is not `NULL`, **rest_p* is set to the address of the first nonwhite character in *str* following the largest expression that could be formed (from the start of *str*).

Use code like the following to evaluate an expression tree named *expr*:

```
if (expr1->eval(expr1)) {
    /* use expression’s value */
 } else {
    /* expression is invalid */
 }
```

That is, the top `EXPR` structure in the expression tree contains a pointer to the routine that should be used to evaluate the expression. If that routine returns `true`, then the expression is valid and the value field of the `EXPR` structure contains the result. If the routine returns `false`, the expression is invalid—that is, it cannot be evaluated at this time—and the value field is meaningless.

`expr_set_msg_context()` sets the message context in which expressions will be evaluated to *context*. If *context* is `NULL`, then all message types and message fields used in expressions will be invalid. The appropriate message context is passed to every layer command entry point when invoked by the framework; see *Adding a Layer Command* on page 84 for details.

`expr_free()` frees an expression tree returned by `expr_parse()`.

`expr_boolean()` checks whether *expr* is of integer type, so that it could be used as a Boolean. Similarly, `expr_access()` checks whether *expr* represents an access expression. If the check passes, *expr* is returned. If the check fails, an appropriate error message is printed, *expr* is freed, and `NULL` is returned.

`expr_get_LWord()` parses an expression from *str* as if the call

```
expr_parse(str, options, rest_p)
```

had been made, makes sure the expression is of integer type, evaluates the expression, places the result in **lword_p*, frees the expression tree, and returns `true`. If the expression is invalid, an appropriate message is displayed and `false` is returned. expr_get_Word() is similar, except that the result is expected to fit into a Word (**word_p*). If the result is too large (that is, the most significant 32 bits of the LWord result are not 0 and the most significant 33 bits of the result are not 1), an appropriate message is displayed and false is returned.

expr_eval_boolean() evaluates *expr* (which must be of integer type); if the expression is invalid or its value is 0, expr_eval_boolean() returns false; otherwise, it returns true.

expr_show() prints the expression corresponding to *expr*, with enough parentheses to make the order of evaluation clear. All printing is done by calling the function pointed to by *print_p* with *arg* as the first argument, followed by printf-style arguments.

expr_show_value() evaluates *expr* and prints its value, and then returns true. If *expr* is invalid, false is returned instead. As with expr_show(), printing is done through print_p(). Integers are always shown in hexadecimal; if simulator option simpleprint is not on, other interpretations can be displayed, depending upon the value of *expr*. If the value fits in a Word, decimal is shown and a symbolic representation will be shown if there is one. If the value is valid ASCII, the character representation is also shown. Expressions of type FLOAT_PT are shown to the precision set in simulator option float_precision. Expressions of type STRING are shown as they would be represented in C, using backslash notation where appropriate.

Sometimes it is desirable to capture the output that would be generated by expr_show() or expr_show_value() in a string. You can capture that output by declaring a variable of type CAPTURE_INFO, initializing it with init_capture_info(); then, call expr_show() or expr_show_value() with capture_printf (the address of the capture_printf() routine) as *print_p* and the address of the CAPTURE_INFO variable as *arg*. When control returns, the CAPTURE_INFO variable's buf member will either be NULL (if no output was generated) or will point to a buffer containing the output accumulated into a single string. This buffer is allocated by malloc() and should be freed with free(). init_capture_info() returns NULL if it is unable to allocate the minimum buffer. The maximum number of bytes that should be generated by each call to capture_printf() is 256.

expr_equiv() returns true if *expr1* and *expr2* represent exactly the same expression; otherwise, it returns false.

All of the following routines are macros that take an EXPR* argument. However, they actually cast their argument to type EXPR*, so that they may be used more readily with ACCESS* arguments and pointers to other types derived from EXPR.

The EXPR_IS_INTEGER() macro returns true if the value of *expr* is of data type SIGNED or UNSIGNED.

EXPR_IS_ACCESS() returns true if *expr* represents an access.

BYTE_VALUE(), HWORD_VALUE(), … STRING_VALUE() all return the value of *expr*. Use the appropriate one for your interpretation, within the bounds of the data type of *expr* (for example, using STRING_VALUE() when *expr* is an UNSIGNED will likely result in a segmentation violation).

VALUE_HI_WORD() returns the most significant 32 bits of the LWord value of *expr*.

Name

fatal, fatal_sim, fatal_nodump, fatal_push — Exit-handling routines

Synopsis

```
#include "types.h"

void fatal(format [ ,arg ]...)
char *format;

void fatal_sim(debug, inst_name, format [ ,arg ]...)
int debug;
Char *inst_name
char *format;

void fatal_nodump(format [ ,arg ]...)
char *format;

void fatal_push(func)
void (*func)();
```

Description

fatal() writes printf-style output to stderr, exits the simulator, and produces a core dump.

fatal_sim() is similar to fatal() if debug is 0. If it is nonzero, fatal_sim() only writes printf-style output and then returns. In both cases, the message displayed is prefixed with the string “*inst_name*:”.

fatal_nodump() writes printf-style output to stderr and exits the simulator with an exit value of 1. No core dump is produced.

fatal_push() saves the function pointer *func* in the framework. *func* will be called by fatal() and fatal_nodump() before exiting. fatal_push() can be called as many times as needed to save function pointers. Functions registered in this way are called in LIFO order.

See Also

Halt

Name

`fwprintf`, `fwprintf_unbuf`, `fwflush`, `fwperror`, `fwputchar`, `fwputs` — Print utilities

Synopsis

```
#include "types.h"

void fwprintf(format [ ,arg ]...)
char *format;

void fwprintf_unbuf(format [ ,arg ]...)
char *format;

void fwfflush()

void fwperror(s)
char *s;

void fwputchar(c)
char c;

void fwputs(s)
char *s;
```

Description

These functions write to stderr and stdout. Modules should not call any of the UNIX system routines that write to stdout or stderr directly or indirectly; these routines should be used instead. Any output generated by these print routines is also written to the log file.

`fwprintf()` writes buffered `printf`-style output to stdout.

`fwprintf_unbuf()` writes unbuffered `printf`-style output to stderr.

`fwfflush()` flushes the buffer associated with stdout.

`fwperror()` writes a short error message to stderr describing the last error encountered during a call to a system or library function. If *s* is not `NULL` and does not point to an empty string, the error message is shown in the form

> *s*: *message*

`fwputs()` writes the null-terminated string pointed to by *s*, followed by a newline character, to stdout.

`fwputchar()` writes a character to stdout.

Name

halt, halt_simulation — Routines to halt the simulation

Synopsis

```
#include "types.h"

void halt(format [ ,arg ]...)
char *format;

void halt_simulation()
```

Description

halt() writes printf-style output to stdout, stops the simulation, and sets the simulation to the halt state (the simulation cannot be restarted).

halt_simulation() stops the simulation and sets the simulation to the halt state.

See Also

Fatal

Name

`register_dbg_intf_mode`, `register_dbg_intf_receive`, `register_sim_intf_mode`, `register_sim_intf_receive`, `register_intf_state` — Interface registration routines

Synopsis

```
#include "interface.h"

void register_dbg_intf_receive(rcv)
void (*rcv)();

void register_sim_intf_receive(rcv)
void (*rcv)();

void register_dbg_intf_mode(mode)
int intf_mode;

void register_sim_intf_mode(mode)
int intf_mode;

void register_intf_state(state)
caddr_t state;
```

Description

These routines are called by the module during the configuration phase to register the different attributes of an interface. These routines can only be called from the module's configure interface entry point and can only be called once per call to a module's configure interface entry point.

`register_dbg_intf_receive()` and `register_sim_intf_receive()` register the receive entry points for the interface being configured for the debug and simulation channels, respectively. The *rcv* parameter is the receive entry point for the interface being configured. A receive entry point needs to be registered only if messages will be sent or queued to the interface.

If and only if a receive routine was registered for an interface, a mode needs to set for the interface. `register_dbg_intf_mode()` and `register_sim_intf_mode()` register the mode of the interface being configured for the debug and simulation channels, respectively. There are two valid interface modes: queued and immediate. The `interface.h` file defines two constants, `QUEUED_MODE` and `IMMEDIATE_MODE`, that can be used to specify the two modes. These routines accept these two defined constants in the *mode* parameter.

Modules have the option of associating state with an interface. `register_intf_state()` registers state (specified by the parameter *state*) to be associated with the interface being configured. It is not interpreted by the framework and can be obtained in receive entry points by calling the `get_interface_state()` function.

See Also

Interface Info

Name

`connect_interfaces` — Connect two interfaces routine

Synopsis

```
#include "interface.h"

void connect_interfaces(intf1, intf2)
caddr_t intf1, intf2;
```

Description

A module can call the `connect_interfaces()` routine at any time to connect two interfaces it owns. If the interfaces are already connected to some other interface, they are unconnected before being connected together.

The *intf1* and *intf2* parameters are the handles of the two interfaces to connect together.

If one of the interfaces is a `READ_ONLY` interface and there is at least one interface connected to it, the `connect_interfaces()` routine fails.

If one of the interfaces is a read-only interface, the other interface must be a write-only interface.

An interface can be connected to itself (provided it is a read-write interface).

See Also

Interface Create

Name

`create_unconnected_interface` — Interface creation routine

Synopsis

```
#include "interface.h"

void create_unconnected_interface(name, type, rw_type)
char *name, *type;
int rw_type;
```

Description

`create_unconnected_interface()` allows a module to create an interface independently of the contents of the configuration file. The routine is called by a module during the configuration phase. It can be called by any configuration phase entry point that is associated with a module instance except for the configure interface entry point (for example, verify configuration entry point).

The parameters to the `create_unconnected_interface()` routine provide information usually specified by an interface declaration in the configuration file. The *name* parameter is a pointer to the string containing the name of the interface to create. The *type* parameter is a pointer to the string containing the type of the interface. The *rw_type* parameter is one of `READ_WRITE`, `READ_ONLY`, or `WRITE_ONLY`.

`create_unconnected_interface()` calls the module's configure interface entry point to configure the interface (just as the framework does when it is configuring an interface specified in the configuration file).

`connect_interfaces()` connects the newly created interface to another interface.

See Also

Interface Connect

Name

get_interface_mod_inst_name, get_interface_name, get_interface_type, get_interface_args, is_interface_connected, get_interface_sim_mode, get_interface_dbg_mode, get_interface_state, set_interface_state — Set and get interface handle information

Synopsis

```
#include "interface.h"
```

```
char *get_interface_mod_inst_name(caddr_t intf)

char *get_interface_name(caddr_t intf)

char *get_interface_type(caddr_t intf)

char *get_interface_args(caddr_t intf)

u_char is_interface_connected(caddr_t intf)

u_char get_interface_sim_mode(caddr_t intf)

u_char get_interface_dbg_mode(caddr_t intf)

caddr_t get_interface_state(caddr_t intf)

void set_interface_state(caddr_t intf, caddr_t state)
```

Description

These macros set and get information associated with an interface handle.

get_interface_mod_inst_name() returns a string pointer to the name of the module instance that owns interface handle *intf*.

get_interface_name() returns a string pointer to the name of the interface of interface handle *intf*.

get_interface_type() returns a string pointer to the interface type of the interface handle *intf*.

get_interface_args() returns a pointer to the interface's argument string for the interface handle *intf*. If no interface arguments were specified in the config file, a null string is returned.

is_interface_connected() returns 1 if the interface handle *intf* is connected to another interface and returns 0 if it is not.

`get_interface_sim_mode()` returns the interface mode for the simulation channel of the interface handle *intf*. This value can be `UNKNOWN_MODE`, `IMMEDIATE_MODE`, or `QUEUED_MODE`.

`get_interface_dbg_mode()` returns the interface mode for the debug channel of the interface handle *intf*. This value can be `UNKNOWN_MODE`, `IMMEDIATE_MODE`, or `QUEUED_MODE`.

`get_interface_state()` returns an opaque pointer to the interface state of the interface handle *intf*.

`set_interface_state()` sets the state pointer of the interface handle *intf* to *state*.

Name

`modify_sim_intf_receive`, `modify_pos_sim_intf_receive`, `modify_neg_sim_intf_receive`, `modify_dbg_intf_receive`, `get_sim_intf_receive`, `get_pos_sim_intf_receive`, `get_neg_sim_intf_receive`, `get_dbg_intf_receive`— Get and set the receive entry points for an interface

Synopsis

```
void modify_sim_intf_receive(intf, rcv)
caddr_t intf;
void (*rcv)();

void modify_pos_sim_intf_receive(intf, rcv)
caddr_t intf;
void (*rcv)();

void modify_neg_sim_intf_receive(intf, rcv)
caddr_t intf;
void (*rcv)();

void modify_dbg_intf_receive(intf, rcv)
caddr_t intf;
void (*rcv)();

void (*)() get_sim_intf_receive(intf)
caddr_t intf;

void (*)() get_pos_sim_intf_receive(intf)
caddr_t intf;

void (*)() get_neg_sim_intf_receive(intf)
caddr_t intf;

void (*)() get_dbg_intf_receive(intf)
caddr_t intf;
```

Description

These macros allow modules to get the current receive entry point and set the receive entry point for interfaces during the simulation phase.

`modify_sim_intf_receive()` and `modify_pos_sim_intf_receive()` change the receive entry point for the positive-phase simulation channel for an interface specified by the *intf* parameter.

`modify_neg_sim_intf_receive()` changes the receive entry point for the negative-phase simulation channel for an interface specified by the *intf* parameter.

`modify_dbg_intf_receive()` changes the receive entry point for the debug channel for an interface specified by the *intf* parameter.

For the preceding macros, the *rcv* parameter specifies the new receive entry point. These macros can only be called if a receive entry point was registered for the channel during the configuration phase.

To obtain the current receive entry point for an interface, you can use the `get_sim_intf_receive()`, `get_pos_sim_intf_receive()`, `get_neg_sim_intf_receive()`, and `get_dbg_intf_receive()` macros. These macros return the receive entry point for the interface specified by the *intf* parameter for the specified channel.

For more information on receive entry points, see Section 4.3, *Simulation Entry Points*.

Name

HI_W, LO_W, UCMP64, SCMP64, EQ64, ucmp_64_64, scmp_64_64, USET64, make64, strto64, hex64, hex32, add_64_64, add_64_32, sub_64_64, sub_64_32, shiftl_64, shiftr_64, arith_shiftr_64, and_64_64, or_64_64, xor_64_64, not_64, neg_64, umul_32_32, smul_32_32 — Arithmetic operations for 64-bit integers

Synopsis

```
#include “types.h”

Word HI_W(LWord lw)
Word LO_W(LWord lw)

Bool UCMP64(LWord lw1, relation, LWord lw2)
Bool SCMP64(LWord lw1, relation, LWord lw2)
Bool EQ64(LWord lw1, LWord lw2)
int ucmp_64_64(LWord lw1, LWord lw2)
int scmp_64_64(LWord lw1, LWord lw2)

void USET64(LWord lw, Word w)
LWord make64(Word w1, Word w2)
LWord strto64(char *buf, char **buf_p, int base)

char *hex32(char *buf, Word w)
char *hex64(char *buf, LWord lw)

LWord add_64_64(LWord lw1, LWord lw2)
LWord add_64_32(LWord lw,   Word w)
LWord sub_64_64(LWord lw1, LWord lw2)
LWord sub_64_32(LWord lw,   Word w)

LWord       shiftl_64(LWord lw, int num)
LWord       shiftr_64(LWord lw, int num)
LWord arith_shiftr_64(LWord lw, int num)

LWord and_64_64(LWord lw1, LWord lw2)
LWord  or_64_64(LWord lw1, LWord lw2)
LWord xor_64_64(LWord lw1, LWord lw2)
LWord not_64(LWord lw)

LWord neg_64(LWord lw)

LWord umul_32_32(Word w1, Word w2)
LWord smul_32_32(Word w1, Word w2)
```

Description

These macros and functions provide a fairly complete set of arithmetic capabilities on 64-bit integers. Two data types correspond to such integers: an `LWord` is an unsigned 64-bit integer, and an `s_LWord` is a two's-complement signed 64-bit integer. However, the distinction between an `LWord` and an `s_LWord` is usually unimportant—the programmer is responsible for picking the signed or unsigned version of an arithmetic operation as appropriate—and this manual page generally refers to an entity of either type generically as an `LWord`.

Be careful not to use the normal C operators with `LWord`s (other than assignment, which is done with the = operator). For efficiency, `LWord`s are actually seen by the compiler as being of type `double`. Therefore, if you were to try to add two `LWord`s together using the + operator, for example, the compiler would not complain; rather, it would perform a double-precision floating-point add of the two 64-bit integers, yielding some bizarre result.

`HI_W()` and `LO_W()` return the most significant and least significant `Word`s of *lw*, respectively.

`UCMP64()` and `SCMP64()` allow unsigned and signed comparison *lw1* and *lw2*. *relation* is any relational operator, as in

```
if (UCMP64(lw1, >, lw2))
```

`ucmp_64_64()` and `scmp_64_64()` compare *lw1* and *lw2* as unsigned or signed numbers and return

```
-1 if lw1 < lw2
 0 if lw1 == lw2
 1 if lw1 > lw2
```

Usually, `UCMP64()` and `SCMP64()` are more convenient ways to compare `LWord`s.

`EQ64()` tests whether *lw1* and *lw2* are equal. It is more efficient than `UCMP64()` used for this purpose.

`USET64()` sets the upper `Word` of *lw* to 0 and sets the lower `Word` to *w*.

`make64()` returns the `LWord` with *lw1* as its most significant `Word` and *lw2* as its least significant `Word`.

`strto64()` is identical to C library function `strtol()` except that it accepts larger integers and returns an `LWord`.

`hex64()` places in *buf* the hexadecimal representation of *lw*, including a leading `0x`, but without leading zeroes. `hex32()` does the equivalent for a `Word`.

`add_64_64()` and `sub_64_64()` return the `LWord` sum *lw1+lw2* or difference *lw1–lw2*.

`add_64_32()` and `sub_64_32()` return the `LWord` sum *lw+w* or difference *lw–w*. *w* is always interpreted as an unsigned integer.

`shiftl_64()` and `shiftr_64()` return the `LWord` obtained by shifting *lw* left or right by *num* bits, shifting in zeroes. Only the bottom 6 bits of *num* are used, and they are taken to be an unsigned number in the range of 0 to 63. `arith_shiftr_64()` is like `shiftr_64()` except that it copies the sign bit rather than shifting in zeroes.

`and_64_64()`, `or_64_64()`, and `xor_64_64()` return the bitwise AND, OR, or XOR of *lw1* and *lw2*.

`not_64()` returns the one's-complement (bitwise NOT) of *lw*.

`neg_64()` returns the two's-complement negation of *lw* (that is, *–lw*).

`umul_32_32()` and `smul_32_32()` return the `LWord` produced by multiplying *lw1* and *lw2*, interpreted as either unsigned or signed operands.

Name

add_msgtype, get_msgtype, get_msgname — Add and access message types

Synopsis

```
#include "msgtype.h"

caddr_t add_msgtype(name)
char *name;

caddr_t get_msgtype(msgname)
char *msgname;

char *get_msgname(msgtype)
caddr_t msgtype;
```

Description

The add_msgtype() function creates a new message type with the specified *name*. Any errors are fatal. An opaque handle for that message type is returned. By convention, this handle is saved in a global variable with the same name as the message type except that msgtype is substituted for pkt, as in

```
caddr_t gen_bus_msgtype;
   ...
   gen_bus_msgtype = add_msgtype("gen_bus_pkt");
```

so that it can be conveniently accessed by any module.

After this call, you can describe the fields (if any) of the message by using the macros described in the *Access Create* manual page (assuming you have included msg_access.h).

Message types are usually defined in the layer initialization entry point of the lowest layer (the layer closest to the fw (framework) layer) that uses them.

get_msgtype() and get_msgname() allow module code to examine the mapping of message type to name. Using these functions, module code can obtain one from the other. get_msgname() returns NULL if there is no message type with the name specified by the *name* parameter.

Name

queue_on_sim_channel, queue_on_pos_sim_channel, queue_on_neg_sim_channel, queue_on_dbg_channel —Queue a message on an interface

Synopsis

```
void queue_on_sim_channel(state, intf, datap, msgtype, size, delay)
caddr_t state, intf, datap, msgtype;
int size, delay;
```

```
void queue_on_pos_sim_channel(state, intf, datap, msgtype, size, delay)
caddr_t state, intf, datap, msgtype;
int size, delay;
```

```
void queue_on_neg_sim_channel(state, intf, datap, msgtype, size, delay)
caddr_t state, intf, datap, msgtype;
int size, delay;
```

```
void queue_on_dbg_channel(state, intf, datap, msgtype, size)
caddr_t state, intf, datap, msgtype;
int size;
```

Description

These routines are very much like the send routines (see the *Send* manual page) with one major difference—the direction of the message. When you *send* a message to an interface, it is delivered to the remote interface of the specified interface. When you *queue* a message on an interface, it is delivered to the local interface specified by *intf*. The queue operation can be thought of as “send this message to me on this interface.” The send operation can be thought of as "send this to the remote module connected to this interface."

The functions `queue_on_sim_channel()` and `queue_on_pos_sim_channel()` perform a queue operation on an interface for the positive-phase simulation channel. The function `queue_on_neg_sim_channel()` performs a queue operation on an interface for the negative-phase simulation channel. The function `queue_on_dbg_channel()` performs a queue operation on an interface for the debug channel.

The *state* parameter is the state pointer of the calling module. The *intf* parameter is the handle of the interface to which to queue the message. The *datap* parameter is a pointer to the message, and *size* is the size of the message being sent. Valid values for *size* are greater than or equal to zero. If *size* is 0, then *datap* can be any value including NULL. If *size* is greater than 0, *datap* must point to dynamically allocated memory. The *msgtype* parameter is the type of the message being sent.

For queue_on_sim_channel(), queue_on_pos_sim_channel(), and queue_on_neg_sim_channel(), the *delay* parameter specifies to the framework the number of cycles to delay before delivering the message.The *delay* value must be greater than or equal to 0.

See Also

Send

Name

`register_config_interface`, `register_create_instance`, `register_verify_config`, `register_cycle`, `register_pos_cycle`, `register_neg_cycle`, `register_shared_object_create`, `register_shared_object_lookup`, `register_dump`, `register_restore`, `register_load_file`, `register_module_command` — Utilities to register module entry points with the framework

Synopsis

```
void register_create_instance(create)
int (*create)();

void register_config_interface(config_intf)
int (*config_intf)();

void register_shared_object_create(shared_obj_create)
int (*shared_obj_create)();

void register_shared_object_lookup(shared_obj_lookup)
int (*shared_obj_lookup)();

void register_verify_config(verify_config)
int (*verify_config)();

void register_cycle(cycle)
int (*cycle)();

void register_pos_cycle(cycle)
int (*cycle)();

void register_neg_cycle(cycle)
int (*cycle)();

void register_dump(dump)
int (*dump)();

void register_restore(restore)
int (*restore)();

void register_load_file(load_file)
int (*load_file)();

void
register_module_command(name, func, short_helpstr, long_helpstr)
char *name;
```

```
int (*func)();
char *short_helpstr;
char *long_helpstr;
```

Description

These routines register the address of module entry points with the framework. They are called during the configuration phase.

`register_create_instance()` registers the address of a module's create instance entry point.

`register_config_interface()` registers the address of a module's config intf entry point.

`register_shared_object_create()` registers the address of the module's shared object create entry point.

`register_shared_object_lookup()` registers the address of the module's shared object lookup entry point.

`register_verify_config()` registers the address of a module's verify config entry point.

`register_cycle()` and `register_pos_cycle()` register the address of a module's positive-phase cycle entry point. `register_neg_cycle()` registers the address of a module's negative-phase cycle entry point.

`register_dump()` registers the address of a module's dump state entry point.

`register_restore()` registers the address of a module's state restore entry point.

`register_load_file()` registers the address of a module's load file entry point.

`register_module_command()` is called once for each user interface command that a module creates. *name* is the name of the command. *func* is a pointer to the module command entry point. *short_helpstr* is a short description of the syntax of the command. The *long_helpstr* string is a complete description of the syntax and semantics of the command.

Name

restore_array_ptr, restore_buffer, restore_intf, restore_message, restore_string — Utilities to help a module restore portions of its state from a file

Synopsis

```
int restore_array_ptr(stream, array_base, element_size, ptr)
FILE *stream;
char *array_base;
u_int element_size;
char **ptr;

int restore_buffer(stream, buf, size)
FILE *stream;
char **buf;
int *size;

int restore_intf(stream, intf, inst_name)
FILE *stream;
caddr_t *intf;
char *inst_name;

int restore_message(stream, data, type, size, delay)
FILE *stream;
caddr_t *data;
caddr_t *type;
int *size;
int *delay;

int restore_string(stream, str)
FILE *stream;
char **str;
```

Description

These restore state routines work in conjunction with the dump state routines to provide facilities to restore data from a stream. The parameters passed to the state restore routines generally match those passed to the corresponding state dump routine, except that the state restore parameters are pointers.

restore_array_ptr() reads the output generated by the dump_array_ptr() routine from *stream* and converts it to its equivalent pointer *ptr* into the array *array_base*. The *element_size* parameter is the size of each array element in bytes.

`restore_buffer()` reads the output generated by the `dump_buffer()` routine from *stream* and returns a pointer to it in *buf*. Space is allocated for the buffer by calling `malloc()`. The *size* parameter is set to the number of bytes in the buffer.

`restore_intf()` reads the output generated by the `dump_intf()` routine from *stream* and restores an interface handle. The *intf* parameter points to the interface handle to be set, and *inst_name* is the name of the module instance that owns the interface handle.

`restore_message()` reads the output generated by the `dump_message()` routine from *stream*. It sets the *data*, *type*, *size*, and *delay* parameters. Space is allocated for the message data by calling `malloc()`.

`restore_string()` reads the output generated by the `dump_string()` routine from *stream* and returns it in *str*. Space is allocated for the string by calling `malloc()`.

Return Values

On success, the state restore routines return `0`; on failure they return `1`.

See Also

Dump State

Name

send_sim_channel, send_pos_sim_channel, send_neg_sim_channel, send_dbg_channel, send_either_channel, send_pos_either_channel, send_neg_either_channel — Send a message on an interface

Synopsis

```
#include "interface.h"

void send_sim_channel(intf, datap, size, msgtype, delay)
caddr_t intf, datap;
int size;
caddr_t msgtype;
int delay;

void send_pos_sim_channel(intf, datap, size, msgtype, delay)
caddr_t intf, datap;
int size;
caddr_t msgtype;
int delay;

void send_neg_sim_channel(intf, datap, size, msgtype, delay)
caddr_t intf, datap;
int size;
caddr_t msgtype;
int delay;

void send_dbg_channel(intf, datap, size, msgtype)
caddr_t intf, datap;
int size;
caddr_t msgtype;

void send_either_channel(intf, datap, size, msgtype, delay)
caddr_t intf, datap;
int size;
caddr_t msgtype;
int delay;

void send_pos_either_channel(intf, datap, size, msgtype, delay)
caddr_t intf, datap;
int size;
caddr_t msgtype;
int delay;
```

```
void send_neg_either_channel(intf, datap, size, msgtype, delay)
caddr_t intf, datap;
int size;
caddr_t msgtype;
int delay;
```

Description

These macros send messages on an interface.

`send_sim_channel()` and `send_pos_sim_channel()` send a message on the positive-phase simulation channel.

`send_neg_sim_channel()` sends a message on the negative-phase simulation channel.

`send_dbg_channel()` sends a message on the debug channel.

`send_either_channel()` and `send_pos_either_channel()` send a message on either the debug or the positive-phase simulation channel (depending on the *delay* parameter).

`send_neg_either_channel()` sends a message on either the debug or the negative-phase simulation channel (depending on the *delay* parameter).

The parameters *intf*, *datap*, *size*, and *msgtype* are common to each of the preceding macros. The message will be sent to the remote interface connected to the interface specified by the *intf* parameter. The *datap* parameter is a pointer to the message, and *size* is the size of the message being sent. Valid values for *size* are greater than or equal to 0. If *size* is `0`, *datap* can be any value including `NULL`. If *size* is greater than 0, *datap* must point to dynamically allocated memory. The *msgtype* parameter is the type of the message being sent.

For simulation channel send macros, the *delay* parameter specifies to the framework the number of cycles to delay before delivering the message. The *delay* value must be greater than or equal to 0. `send_dbg_channel()` does not have a *delay* parameter because delays are not allowed on the debug channel.

The send `either` macros can be used to send a message on the debug channel or the simulation channel. The *delay* parameter tells the framework which channel to send the message on. If *delay* is zero or positive, the message is sent on the simulation channel with that amount of delay. If the *delay* parameter is negative, the message is sent on the debug channel.

The send `either` macros are useful for receive entry points that receive messages on both the debug and simulation channels. The *delay* parameter passed to receive entry points when delivering a simulation channel message is either zero or a positive number. When a message is delivered on the debug channel, the receive entry point is called with a negative value in the *delay* parameter. Because of this characteristic of the *delay* parameter, the send `either` macros allow the receive entry point to use the same code to react to both simulation and debug messages. The receive entry point can call the appropriate send either macro with the delay passed into the receive entry point, and the message will be sent on the channel on which the message was received.

Each of the above macros also has a function with the same name and _func suffixed that performs the same way as the macro.

See Also

Queue

Name

`register_object_ptr`, `register_object_size` — Shared object registration routines

Synopsis

```
#include "module.h"

void register_object_ptr(ptr)
void *ptr;

void register_object_size(size)
int size;
```

Description

These routines are called by the module during the configuration phase to register the pointer and the size of the object being shared. These routines can only be called from the module’s create shared object entry point and can only be called once per call to a module’s create shared object entry point.

The pointer to the object being registered as a shared object is passed to `register_object_ptr()` as a parameter. The size in bytes of the object being shared is passed to `register_object_size()` as a parameter.

See Also

Config Info

Name

sigio_set_input_mapping, sigio_set_output_mapping — Map a file descriptor for input or output with an interface

Synopsis

```
#include "sigio.h"

caddr_t sigio_set_input_mapping(fd, mode, intf)
int fd;
int mode;
caddr_t intf;

caddr_t sigio_set_output_mapping(fd, mode, intf, fd_name)
int fd;
int mode;
caddr_t intf;
char *fd_name;
```

Description

sigio_set_input_mapping() and sigio_set_output_mapping() are called by a module's verify config entry point to ask the sigio facility to perform input and output, respectively, on file descriptor *fd*. This file descriptor must have been opened by the module.

The *mode* parameter must be SIGIO_RAW_MODE or SIGIO_BLOCK_MODE.

The *intf* parameter is the interface handle owned by the calling module that will be used to communicate with the sigio facility.

The *fd_name* parameter for sigio_set_output_mapping is used for error messages associated with output sigio performs to *fd*.

The sigio message type is used to communicate with sigio. The message format is described in Chapter 7, *Asynchronous Input (sigio)*.

Return Values

Both functions return the handle for the interface previously mapped to *fd*, or NULL if none.

Name

stop_simulation, start_simulation, sim_running — Start, stop, and check the status of the simulation

Synopsis

```
#include "types.h"

void stop_simulation()

void start_simulation()

Bool sim_running()
```

Description

stop_simulation() can be used by module code to stop the simulation. The remainder of the cycle will be simulated before the simulation is stopped. This allows each module to be at the same point of simulation before everything is stopped. stop_simulation() is commonly used by a module to stop the simulation when an error condition is reached.

start_simulation() starts the simulation if the simulation is currently not running. It has no effect if the simulation is already running.

sim_running() can be used to determine whether the simulation is currently running.

Return Values

sim_running() returns true if the simulation is currently running; otherwise, false.

Name

strmcat, strmcpy, strmdup, strdup_fatal — Concatenate many strings

Synopsis

```
#include “types.h”

char *strmcat(buf, s1, s2, ..., NULL)
char *buf, *s1, *s2;

char *strmcpy(buf, s1, s2, ..., NULL)
char *buf, *s1, *s2;

char *strmdup(s1, s2, ..., NULL)
char *s1, *s2;

char *strdup_fatal(s)
char *s;
```

Description

`strmcat()`, `strmcpy()`, and `strmdup()` concatenate an arbitrary number of strings—*s1*, *s2*, …—to make a new string. A `NULL` argument marks the end of the list of strings.

`strmcat()` and `strmcpy()` create the new string in *buf*; the caller is responsible for ensuring that *buf* is large enough to hold the new string.

`strmdup()` allocates space for the string from the heap using `malloc()`; the caller is responsible for freeing the space for the string (using `free()`) when it is no longer needed. Both functions return a pointer to the new string.

`strdup_fatal()` allocates space (using `malloc()`) to duplicate the string *s*. It copies *s* into this new buffer and returns a pointer to it. It calls `fatal_nodump()` if `malloc()` fails.

Name

ui_parsew, ui_parse_delimiter, look_for_keyword, get_argc_argv — String parsing utilities

Synopsis

```
#include "types.h"

char *ui_parsew(str, wp)
char *str;
char **wp;

char *ui_parse_delimiter(str, wp, del)
char *str;
char **wp;
char del;

int look_for_keyword(argc, argv, keyword, num_params, errmsg)
int argc;
char **argv;
char *keyword;
int num_params;
char *errmsg;

int get_argc_argv(argvp, args)
char ***argvp;
char *args;
```

Description

ui_parsew() parses through the string *str*. It deletes leading white space characters and writes a null character after the first word (sequence of nonwhite characters) in the string. The word pointer *wp* is set to point to this first word. It returns a pointer to the rest of the line. If the string does not contain any text, *wp* and the returned pointer point to the null character.

ui_parse_delimiter() searches through the string *str* for the delimiter *del*. It places a null character after the word to the left of the delimiter. The word pointer *wp* is set to point to this word. The string to the right of *del* will be returned. If no delimiter is found, *wp* points to the null character at the end of the string and the entire string *str* is returned.

look_for_keyword() scans through the arguments pointed to by *argv* and looks for *keyword*. *num_params* specifies how many parameters must be after *keyword*. The routine verifies that each parameter exists. If *keyword* has no parameters, *num_params* should be 0. If the caller wants a message to be displayed when an error occurs, it must set *errmsg* to the string it wants to display as a prefix to the error message; otherwise, it sets *errmsg* to `NULL`. If *keyword* is not found, `-1` is returned. If the parameter validation fails, 0 is returned. If everything is successful, the index after *keyword* into *argv* is returned.

`get_argc_argv()` parses the string pointed to by the parameter *args* in a way that simulates `argc` and `argv` for `main`. It considers *args* to point to a string containing tokens separated by one or more spaces, newlines, tabs, carriage returns, or formfeeds. This routines sets the value pointed to by *argvp* to point to an array of strings (character pointers). This array has `argc + 1` elements, and the last element is a null character pointer. The number of arguments found is returned. Null characters will be placed in the string pointed to by the *args* parameter. `get_argc_arv()` allocates an array that is returned in the pointer pointed to by the parameter *argvp*, which should be freed by the caller, using `free()`.

Name

`toSymbolic`, `toAddr` — Convert between addresses and their symbolic representations

Synopsis

```
#include “types.h”
#include “symtabhndlr.h”

char *toSymbolic(addr, buf)
LWord addr;
char *buf;

Bool toAddr(buf, addr_p)
char *buf;
LWord *addr_p;
```

Description

These functions allow conversion between numeric addresses and symbolic addresses. All currently loaded symbol tables are searched (starting with the last one loaded), each for the range of addresses associated with it. Note that addresses are 64-bit `LWord`s.

`toSymbolic()` puts the symbolic representation—or, if there is none, the hexadecimal representation— of address *addr* into character buffer *buf*. The caller is responsible for ensuring that *buf* is large enough. *buf* is returned. The representation is always in one of the following forms:

symbol
symbol+0x*hex*
0x*hex*

`toAddr()` puts the numeric address corresponding to the symbolic address in *buf* into the `LWord` pointed to by *addr_p*, and returns `true`. If the symbol used cannot be found in any of the currently loaded symbol tables, however, `toAddr()` returns `false`. Each symbol table is searched first for the symbol as given; if that fails, the search is performed again for the same symbol with an underscore prepended, because C compilers generally prepend an underscore to identifier names.

APPENDIX **B**

Example Module Listing

Some chapters in this document refer to the `example` module class for examples. This appendix contains a complete listing of the `example` module class. Note that this example was designed to compile under Sun C; it will not compile under C++. For an example of a C++ style module class, see the `cxx.c` and `cxx.h` files in the `computer` layer.

Here is the listing of the example file.

```
#include "types.h"
#include "module.h"
#include "interface.h"
#include "gen_bus_pkt.h"
#include "gen_int_pkt.h"
#include "expr.h"
#include "state_access.h"

/* Names the config file uses for the interfaces. */
#define SLAVE_INTF_TYPE_NAME    "slave"
#define INTR_INTF_TYPE_NAME     "interrupt"
struct example_psr {
       u_int rsv : 26;
       u_int et  : 1;
       u_int rsv_1 : 5;
};

union u_example_psr {
       struct example_psr s;
       Word w;
};

#define MAX_CORE_SIZE   0x4000
struct example_state {
        char           *inst_name;  /* module instance name */
        caddr_t         intr_intf;  /* interrupt interface handle */
        LWord           reg_addr;   /* address of timer register */
        Byte           *core_ptr;   /* pointer into core array */
        union u_example_psr *sh_example_psr;
#define EXAMPLE_DUMP_PT count
        Word            count;      /* timer register */
        Byte            core[MAX_CORE_SIZE]; /* memory array */
};
/*
 * EXTERNALS
 */
extern caddr_t  gen_int_msgtype;

/*
 * FORWARD DECLARATIONS
 */

int             example_module_init();

/* registration routines */
static caddr_t  example_create_instance();
static int      example_config_intf();
static int      example_create_shared_obj();
static int      example_lookup_shared_obj();
static int      example_verify_config();
static Bool     example_dump();
static Bool     example_restore();
static void     example_cycle();
static int      example_core_cmd();

/* interface receive routines */
static void     example_slave_sim_rcv();

/* misc forward declarations */
static void     example_create_accesses();

static char       core_shorthelp[] = "core [<index>] - display core memory array";
static char     core_longhelp[] = "core [<index>]\n\
   /*  This command displays the core memory array. If index is specified, it\n\
     *  displays the specified element. If index is omitted, it displays the\n\
     * element pointed to by the current core pointer."; */
/*
 * This routine is called once for the example module class. It registers the other
 * module class entry points with the framework.
 */
int
example_module_init()
{
        /* Register the module class entry points with the framework. */
        register_create_instance(example_create_instance);
        register_config_interface(example_config_intf);
        register_shared_object_create(example_create_shared_obj);
        register_shared_object_lookup(example_lookup_shared_obj);
        register_verify_config(example_verify_config);
        register_dump(example_dump);
        register_restore(example_restore);
        register_cycle(example_cycle);
         register_module_command("core", example_core_cmd, core_shorthelp,
            core_longhelp);

        return 0;       /* no errors */
}
/*
 * This routine is called once by the framework for each instance of the
 * example module class defined in the config file.
 * This routine parses the config file arguments for this instance declaration,
 * allocates a copy of my state structure, and returns a pointer to it.
 */
static caddr_t
example_create_instance(args)
        char           *args;
{
        struct example_state *msp;
        int             argc;
        char          **argv;
        int             index;

        /* Allocate a chunk of memory big enough for my state. */
        if ((msp = (struct example_state *) calloc(1,
            sizeof(struct example_state))) == NULL) {
                fatal_nodump("Unable to malloc.\n");
        }

        /* Store pointer to instance name in my state. */
        msp->inst_name = get_config_mod_instance_name();

         /* Get framework to convert args to an argc/argv data structure. */
        argc = get_argc_argv(&argv, args);

        /* Get address (64-bit value) of my counter register. */
        if ((index = look_for_keyword(argc, argv, "REG_ADDR", 1,
            msp->inst_name)) <= 0) {
                    fatal_nodump("%s: error in config file\n", msp->inst_name);
        }

        msp->reg_addr = strto64(argv[index], (char **)NULL, 0);

          /* Create access to allow user to read and write count register. */
        WORD("count", &msp->count);
        /* Return an opaque pointer to my state. */
        return (caddr_t)msp;
}
/*
 * This routine is called by the framework for each interface defined
 * in the config file for each example instance.
 * If it finds a problem it returns non-zero otherwise zero.
 */
static int
example_config_intf(state, intf)
        caddr_t         state;
        caddr_t         intf;
{
        struct example_state *msp = (struct example_state *)state;
        char            *type = get_interface_type(intf);
        int             connected = is_interface_connected(intf);
        int             okay_to_be_unconnected = 0;
        static char     duplicate_interface_type[] = "example: module instance
\"%s\", interface \"%s\":\n\tOnly one \"%s\" interface type allowed.\n";
        static char     unknown_interface[] = "example: module instance \"%s\",
interface \"%s\":\n\tUnknown interface type \"%s\".\n";
        static char     unconnected_interface[] = "example: module instance \"%s\",
interface \"%s\":\n\tInterface type \"%s\" cannot be unconnected.\n";

        if (strcmp(type, SLAVE_INTF_TYPE_NAME) == 0) {
                /*
                   * Any number of slave interfaces is allowed. Messages arriving
                   * on them are all handled by the example_slave_sim_rcv()
                   * routine. I never send to the slave interface unless it
                 * sends me a message so it can be unconnected.
                 */
                register_sim_intf_mode(IMMEDIATE_MODE);
                register_sim_intf_receive(example_slave_sim_rcv);
                okay_to_be_unconnected = 1;
        } else if (strcmp(type, INTR_INTF_TYPE_NAME) == 0) {
                /*
                   * Only one interrupt interface is allowed. Its interface
                   * handle is stored in my state. I never receive from the
                   * interrupt interface so I don’t register a receive routine
                 * for it.
                 */
                if (msp->intr_intf) {
                        fwprintf(duplicate_interface_type,
                            get_interface_mod_inst_name(intf),
                            get_interface_name(intf), type);
                        return 1;
                }
                msp->intr_intf = intf;
        } else {
                fwprintf(unknown_interface,
                    get_interface_mod_inst_name(intf),
                    get_interface_name(intf), type);
/*
 * This routine is called after all interfaces have been configured for all
 * modules. It is called for each example instance. If this routine likes the
 * state, it returns 0 otherwise non-zero.
 */
static int
example_verify_config(state)
        caddr_t         state;  /* per instance state pointer */
{
        struct example_state *msp = (struct example_state *)state;
        static char missing_interface[] = "example: module instance
\"%s\":\n\tInterface type \"%s\" not present.\n";

        /*
           * Make sure that an interrupt interface handle was stored in my state
         * during the example_config_intf() routine.
         */
        if (msp->intr_intf == NULL) {
                fwprintf(missing_interface, msp->inst_name,
                    INTR_INTF_TYPE_NAME);
                return 1;
        }

        return 0;
}
/*
 * This routine is called by the framework when the user requests the state of an
 * instance of the example module class to be dumped to a file.
 */
static Bool
example_dump(state, fp)
        caddr_t         state;
        FILE           *fp;
{
        struct example_state *msp = (struct example_state *) state;
        char           *src = (char *) &msp->EXAMPLE_DUMP_PT;
        int             size;  /* # bytes we dump */

        size = sizeof(struct example_state) - (int) (src - (char *) msp);

        /*
         * Write out portion of my state below the dump point.
         */
        if (fwrite(src, size, 1, fp) != 1) {
```

```
                  fwprintf_unbuf("%s: fwrite failed\n", msp->inst_name);
                  fwperror("");
                  return FALSE;
        }

       /*
            * Write out portion of my state above dump point that changes with
         * the simulation.
        */
           if (dump_array_ptr(fp, msp->core, sizeof(Byte), msp->core_ptr)) {
                  fwprintf_unbuf("%s: dump_array_ptr failed\n",msp->inst_name);
                  fwperror("");
                  return FALSE;
        }

        return TRUE;
}
/*
 * This routine is called by the framework when the user requests the state of an
 * instance of the example module class to be restored from a file.
 */
static Bool
example_restore(state, fp)
        caddr_t         state;
        FILE           *fp;
{
        struct example_state *msp = (struct example_state *) state;
        char           *src = (char *) &msp->EXAMPLE_DUMP_PT;
        int             size;  /* # bytes we dump */

         size = sizeof(struct example_state) - (int) (src - (char *) msp);

        /*
         * Read in portion of my state below the dump point.
         */
        if (fread(src, size, 1, fp) != 1) {
                  fwprintf_unbuf("%s: fread failed\n", msp->inst_name);
                fwperror("");
                return FALSE;
        }

        /*
            * Read in portion of my state above dump point that changes with
         * the simulation.
         */
         if (restore_array_ptr(fp, msp->core, sizeof(Byte), &msp->core_ptr)) {
                  fwprintf_unbuf("%s: restore_array_ptr failed\n", msp->inst_name);
                  fwperror("");
                  return FALSE;
        }
```

```
return 1;
        }

        if (!connected && !okay_to_be_unconnected) {
                  fwprintf(unconnected_interface,
                    get_interface_mod_inst_name(intf),
                    get_interface_name(intf), type);
                  return 1;
        }

        return 0;
}

/*
 * This routine is called by the framework for each shared object
 * declaration defined in the config file for each example instance.
 * If it finds a problem it returns non-zero, otherwise zero.
 */

static int
example_create_shared_obj(state, obj_name)
        caddr_t state;
        char *obj_name;
{
        struct example_state *msp = (struct example_state *) state;

        if(!strcmp(obj_name, "count")) {
                  register_object_ptr(&msp->count);
                  register_object_size(sizeof(Word));
                  return 0;
        }
        else {
               fwprintf("%s:example_create_shared_obj: %s Object unknown\n",
                            msp->inst_name, obj_name);
           return 1;
        }
}

/*
 * This routine is called by the framework for each shared object lookup
 * in the config file for each example instance.
 * If it finds a problem it returns non-zero; otherwise, zero.
 */

static int
example_lookup_shared_obj(state, obj_name)
        caddr_t state;
        char *obj_name;
```

```
{
        struct example_state *msp = (struct example_state *) state;
        union u_example_psr sh_psr;
        ACCESS*             access;
        void                *ptr = get_object_ptr();
        int                 size = get_object_size();

        if (!strcmp(obj_name, "example_psr")) {
                  msp->sh_example_psr = (union u_example_psr *) ptr;
                  msp->sh_example_psr_size = size;
                  /* Shared PSR */
                  access = WORD("sh_example_psr", &msp->sh_example_psr->w);
                  access_compact_print(access);
                  MEMBER_BITF(access, "et", WORD_MASK(sh_psr, et));
                  return 0;
        } else {
                  fwprintf("%s:example_lookup_shared: %s Object unknown\n"
                    msp->inst_name, obj_name);
                  return 1;
        }
}

/*
 * Slave simulation channel interface receive routine.
 *  This routine is called for each message received on the slave interface type.
 */
static void
example_slave_sim_rcv(state, intf, data, type, size, delay)
         caddr_t           state;    /* opaque ptr to this module’s state struct */
        caddr_t         intf;   /* interface message came in on. */
        caddr_t         data;   /* opaque ptr to the data pkt */
        caddr_t         type;   /* type id of the data pkt */
        int             size;   /* sizeof the data packet */
        int             delay;  /* always 0 for queued interfaces */
{
        struct example_state *msp = (struct example_state *) state;
        struct gen_bus_pkt *gbp = (struct gen_bus_pkt *) data;
        Word            tmp;

       /*
          * Compare physical address of request with my address and ensure that
         * the size of the request is 4 bytes.
        */
         if (EQ64(gbp->paddr, msp->reg_addr) && gbp->size == sizeof(Word)) {
                /* Examine the packet’s type field. */
                  switch (gbp->type) {
                  case GEN_BUS_RD:
                           /* Load my register’s contents into the packet. */
                           *(Word*)gbp->data = msp->count;
                           break;
                  case GEN_BUS_WR:
                           /* Set my register’s contents from the packet. */
                        msp->count = *(Word*)gbp->data;
                        break;
                  case GEN_BUS_RW:
                        /*
                            * Load my register’s contents into the packet and write
                          * my register’s contents with the packet’s original contents.
                        */
                        tmp = msp->count;
                        msp->count = *(Word*)gbp->data;
                        *(Word*)gbp->data = tmp;
                        break;
                  default:
                           fatal("%s: unknown gen_bus_pkt type of 0x%x.\n",
                            msp->inst_name, gbp->type);
                }
                  gbp->status = GEN_BUS_OK;
        } else {
                  gbp->status = GEN_BUS_FAULT;
        }

        send_sim_channel(intf, (caddr_t)gbp, size, type, delay);
}
/*
 * This routine is called once each cycle the simulator executes.
 */
static void
example_cycle(state)
        caddr_t         state;
{
        struct example_state *msp = (struct example_state *) state;
        struct gen_int_pkt *gip;

        /* Decrement counter. If zero, send an interrupt packet. */
        if (--msp->count == 0) {
                  gip = new_gen_int_pkt();

                  gip->action = INTERRUPT_SET;

                  send_sim_channel(msp->intr_intf, (caddr_t)gip, sizeof(*gip),
                    gen_int_msgtype, 0);
        }
}
/*
 * This routine is called each time the user invokes the "core" command.
*/
static int
example_core_cmd(state, cmd, args)
```

```
        caddr_t         state;
        char           *cmd;    /* Actual command string */
        char           *args;   /* Any arguments to the command */
{

        struct example_state *msp = (struct example_state *) state;
        char           *rest_of_line;
        char           *index_str;
        Byte           *cptr;
        int             index_num;

        if (*args == ’\0’) {
                /* nothing specified - display the core_ptr entry */
                  cptr = msp->core_ptr;
                  if (cptr == NULL) {
                           fwprintf("%s: core_ptr is NULL\n", msp->inst_name);
                       /* Set cmd_result variable to -1. */
                       cmd_result = make64(-1, -1);
                       return UI_CMD_IS_DONE;
                }
                  index_num = (msp->core_ptr - &msp->core[0]) / sizeof(Byte);
        } else {
                  /* Parse core argument. Make sure only one arg on line. */
                  rest_of_line = ui_parsew(args, &index_str);
                  if (*rest_of_line != ’\0’) {
                       fwprintf("Usage: core [<index>]\n");
                       return UI_CMD_IS_DONE;
                  }

                  /* Get expression parser to evaluate index expression. */
                  if (expr_get_Word(index_str, EXPR_OPT_MSG,
                   _, &index_num) == FALSE) {
                       return UI_CMD_IS_DONE;
                  }

                  if (index_num < 0 || index_num > MAX_CORE_SIZE-1) {
                           fwprintf("%s: invalid index specified \"%s\".\n",
                           msp->inst_name, index_str);
                       return UI_CMD_IS_DONE;
                  }

                cptr = &msp->core[index_num];
        }

         fwprintf("%s: core[%d] =  0x%x\n", msp->inst_name, index_num, *cptr);
        /* Set cmd_result variable to value of the element displayed. */
       SET_CMD_RESULT_AS_WORD(*cptr);

       return UI_CMD_IS_DONE;
}
```

```
/*
 * This routine is called when the user asks that a file be loaded into an instance
 * of the example module’s memory.
 */
static Bool
example_load_file(state, fp, addr, size)
        caddr_t         state;
        FILE           *fp;     /* stream to load memory from */
        LWord           addr;   /* address to load memory into */
         int             size;   /* number of bytes to load into my memory */
{
        struct example_state *msp = (struct example_state *) state;

        /*
         * Check for valid address range.
         * Valid addresses range from 0 to MAX_CORE_SIZE-1
         */
          if (UCMP64(add_64_32(addr, size), >, make64(0, MAX_CORE_SIZE-1))) {
                fwprintf("%s: invalid address 0x%x%08x specified.\n",
                 msp->inst_name, HI_W(addr), LO_W(addr));
                return FALSE;
        }

        if (fread(&msp->core[LO_W(addr)], size, 1, fp) != 1) {
                fwprintf_unbuf("%s: load file failed\n", msp->inst_name);
                fwperror("");
                return FALSE;
        }

        return TRUE;
}
```

APPENDIX **C**

Services Provided by the Layers

Some of the layers provided with the product provide services of use to module programmers. This appendix details these services.

C.1 Message Types

The message types mentioned here are all discussed in detail in the chapter on Message Types in *Multiprocessor SPARC Architecture Simulator (MPSAS) User's Guide*. This section supplements the *User's Guide* by supplying information on routines of use to the module programmer in dealing with those message types. Since the structure of each message type is visible to the user through the `msg` command, the structure itself is presented in the *User's Guide* and is not duplicated here.

C.1.1 `computer` Layer Message Types

The `computer` layer defines (and describes using accesses) two message types in its `layer.c` file:

- `gen_bus_pkt`
- `gen_int_pkt`

The message types are available as global `caddr_t` variables `gen_bus_msgtype` and `gen_int_msgtype`.

`gen_bus_pkt` Message Type

The `new_gen_bus_pk` routine returns a new `gen_bus_pkt` structure allocated with `calloc`; the programmer is responsible for deallocating the packet.

```
#include "types.h"
#include "gen_bus_pkt.h"
struct gen_bus_pkt
     *new_gen_bus_pkt(data_size, extra_size, total_size_p)
int data_size, extra_size, *total_size_p;
```

data_size is the number of bytes of data the packet must hold in its `data` field. *extra_size* is the number of bytes beyond the `data` field that can be used for storing data; if *extra_size* is nonzero, the packet's `extra` field is initialized to the size of the packet *not including* the extra area at the end. The total size of the packet is returned at **total_size_p*. If memory cannot be allocated for the packet, a fatal error is issued; otherwise the address of the packet is returned.

The `new_gen_bus_pkt_dbg` function does the identical thing, except that if the memory cannot be allocated, a `NULL` is returned.

`char *GEN_BUS_EXTRA(struct gen_bus_pkt *gbp)`
is a macro that returns a pointer to the extra area of a `gen_bus_pkt` structure.

```
Byte   GBP_DATA_BYTE(     struct gen_bus_pkt*gbp)
HWord  GBP_DATA_HWORD(    struct gen_bus_pkt*gbp)
Word   GBP_DATA_WORD(     struct gen_bus_pkt*gbp)
Word   GBP_DATA_2ND_WORD( struct gen_bus_pkt*gbp)
LWord  GBP_DATA_LWORD(    struct gen_bus_pkt*gbp)
```

are all macros that return parts of the `data` field of a `gen_bus_pkt`. For example, `GBP_DATA_BYTE` returns the first `Byte` of the data field. The only macro that does not return the beginning of the `data` field is `GBP_DATA_2ND_WORD`, which returns the least significant Word of the `data` field taken as an `LWord`. Use these macros only when the `data` field is large enough to actually hold the item the macro will try to return.

`gen_int_pkt` Message Type

`New_gen_int_pkt` allocates a `gen_int_pkt` with `calloc`; you are responsible for deallocating the packet. If memory for the packet is not available, a fatal error is issued.

```
#include "types.h"
#include "gen_int_pkt.h"
struct gen_int_pkt *new_gen_int_pkt()
```

C.1.2 `sparc` Layer Message Types

The `sparc` layer defines (and describes using accesses) three message types in its `layer.c` file:

- trap_pkt
- cpu_pkt
- fpu_pkt

The message types are available as global caddr_t variables trap_msgtype, fpu_msgtype, and coproc_msgtype.

C.2 computer Layer Memory Routines

You might want to create a module that simulates some type of memory array. The computer layer of MPSAS includes a set of routines (called *memory routines* in this section) that, if they meet your needs, do most of this work for you. From these routines, you can easily build a module, as the computer layer does for RAM and ROM.

You will have to write the module init routine for your module. That module init routine will probably register some of the memory routines with the framework, to do such things as create instances of the module or handle requests from the framework. You can choose to register some of your own routines and some of those from this package, changing specific aspects of the behavior of the module. The following files supply this functionality:

- computer/include/mem.h — Include this to use memory routines
- computer/mem.c — Source for memory routines

These files might be of interest as examples:

- computer/ram.c — RAM module; trivial example
- computer/rom.c — ROM module; sets up to reject writes

The following memory routines are externally available. Each is a module entry point for some functionality needed by the framework.

- mem_create_instance — Creates instance entry point
- mem_config_intf — Config intf entry point
- mem_dump — Dump instance state entry point
- mem_restore — Restore instance state entry point
- mem_load_file — Load file entry point
- mem_slave_sim_rcv, mem_slave_sim_RO_rcv — Simulation channel receive entry points
- mem_slave_dbg_rcv — Debug channel receive entry point

These entry points are used in the ram and rom modules. Refer to the chapter on modules in *Multiprocessor SPARC Architecture Simulator (MPSAS) User's Guide* for a description of their behavior.

Here is the ram module:

```
/*
 * ram.c -- RAM module for MPSAS.
 */

#include "types.h"
#include "module.h"
#include "mem.h"

/*
 * ram_module_init() is called during initialization because of an entry
 * in the config table. Its job is to tell the framework the details about
 * this module so that instances of it can be created.
 *
 * This module relies on libmem to do all the work.
 */
int
ram_module_init()
{
        register_create_instance(mem_create_instance);
        register_config_interface(mem_config_intf);
        register_dump(mem_dump);
        register_restore(mem_restore);
        register_load_file(mem_load_file);

return 0; /* indicate success */
}
```

In this simple case, the module init routine simply registers routines out of the memory library to do all the work, and everything else is handled by those routines.

Mem_create_instance is a create instance routine. You might want to replace it or provide a wrapper for it if, for example, your module takes additional module parameters over those taken by the ram and rom modules: START_ADDR and SIZE.

Mem_config_intf is a config intf routine, so it is called by the framework for each interface defined in the configuration file for each instance of a memory object. This routine expects to be called to set up interfaces of type MEM_SLAVE_INTF_TYPE_NAME (defined in mem.h). A module based on these routines can be configured to have an arbitrary number of such interfaces, over which another module could read from and write to the memory. If you want to have additional interfaces of another type, you will need to replace this routine or provide a wrapper for it.

mem_dump and mem_restore are dump instance state and restore instance state entry points. You might need to replace them in order to implement other functionality for which you extended the module's state structure beyond the mem_state structure used by the routines discussed here. *The mem_state Structure* on page 200 discusses this matter further.

The config intf routine registers the receive routines for the simulation channel and the debug channel on the interface. Ordinarily, it registers mem_slave_sim_rcv and mem_slave_dbg_rcv to be the handlers for messages on the simulation and debug channels, respectively, but you can override those choices. To do so, your module init routine should create a mem_funcs structure (defined in mem.h) with pointers to the appropriate routines, then use the set_module_extra framework routine to put the address of the mem_funcs structure where the config intf routine can get it. It should all look something like this:

```
int
my_mem_module_init()
{
       static mem_funcs my_mem_funcs = { <names of various routines> };
        set_module_extra(get_config_mod_class(), &my_mem_funcs};
        register_config_interface(mem_config_intf);

        other initialization
}
```

The rom module uses this mechanism in order to use mem_slave_sim_RO_rcv as the receive routine rather than mem_slave_sim_rcv. The mem_funcs structure itself looks like this:

```
struct mem_funcs {
       void (*slave_sim_rcv)(); /* default is mem_slave_sim_rcv */
       void (*slave_dbg_rcv)(); /* default is mem_slave_dbg_rcv */
};
```

There are default routines for each of these members, and you can specify that you want the default used simply by setting the corresponding member to 0.

mem_slave_sim_rcv and mem_slave_dbg_rcv handle messages on the simulation and debug channels, respectively. Another variant of the simulation channel receive routine, mem_slave_sim_RO_rcv, always returns a status of GEN_BUS_FAULT on any write (to simulate read-only memory). Note that these routines can tell when you have put a wrapper around them, and if so they will *not* send back the response message, so that the wrapper has control over the response to the request. That is, these receive routines only send back the response if you did *not* override them in the mem_funcs structure.

C.2.1 The mem_state Structure

The memory routines use a module state structure called mem_state, defined in mem.h.

```
/* State structure for memory. Dump only saves the core array. Note that
 * since this structure is of fixed size, it can easily be extended by a
 * module that requires extra state information. */
struct mem_state {
        char           *inst_name;      /* name of inst that mem is in */
        LWord           lo_addr;        /* lowest address simulated */
        LWord           hi_addr;        /* highest address simulated */
        LWord           offset;         /* offset for disassembly */
        Word            size;           /* # bytes simulated - size of core[] */
        Word            signif_mask;    /* AND mask to get index from address */
        struct mem_funcs funcs;
        unsigned char  *core;           /* the actual ram being simulated */
};
```

If you need to add additional state information in order to implement your memory module, you can do so by defining a new structure that contains a mem_state as its first member, followed by whatever additional members you require. You will need to replace the mem_create_instance routine to allocate this new structure. Also, if the members you added need to be saved and restored, you will need to replace mem_dump and mem_restore.

The inst_name field is the string name of the module instance, used in printing messages to the user.

Lo_addr and hi_addr are the physical addresses of the first and last bytes of simulated memory.

The offset field is used when disassembling instructions. The user can set it to any 64-bit value via the offset_for_disassembly variable, and the access class for the instruction variable uses this when disassembling instructions. The memory routines themselves really don't use it.

core is a pointer to an array representing the memory being simulated. The size field is the size of the array—the number of bytes of memory being simulated. signif_mask is a value constructed from lo_addr and size, such that given LWord simulated memory address addr, the expression

```
    LO_W(addr) & signif_mask
```

yields the index into the core array corresponding to the byte at address addr. That is, by setting high-order bits to 0, this expression yields a number in the range 0 to size-1.

funcs is a mem_funcs structure, constructed from the programmer's mem_funcs structure (if one was supplied), but with all defaults filled in. That is, this structure will contain valid pointers to simulation and debug channel receive routines.

If you write your own create instance routine, it should initialize all of these members to appropriate values. One way to ensure that these members are set appropriately is to start with the code for mem_create_instance in mem.c. A better way would be to write a wrapper for mem_create_instance, calling it to do its part of the job and then finishing the job in the wrapper.

Index