SIMON II KERNEL

Reference Manual

A General Purpose Discrete Simulation Tool

Technical Report UUCS-86-001

21 May 1987

by

Steven M. Swope

and

Richard M. Fujimoto

University of Utah Computer Science Dept.

Table of Contents

SIMON II KERNEL

The principal objective of Simon II is to provide a flexible and adaptable framework for constructing simulators for a wide variety of parallel systems. A simulator consists of a set of software building blocks. Each building block, i.e. object, simulates a specific component of the parallel system. Objects may be defined in terms of other objects, supporting a hierarchical view of the system.

This building block approach offers several advantages. Comparison of alternative designs and approaches is simplified. Different aspects of the system can easily be modeled with different degrees of detail (mixed mode simulation). The standardization of interfaces between objects facilitates the partitioning of simulator development among several individuals, each of which may be a specialist in a different aspect of the system, allowing designers to concentrate on areas in which they are most knowledgeable. Finally, the richness and capabilities of the simulation system grow with time as new objects are developed and refined, in contrast to simulators which are developed to examine one particular aspect of the system, and then discarded. Thus, costs to develop simulators for new systems are reduced.

PART I

Simon II Basic Features

Part I of this manual describes basic features of Simon, including an introduction, a tutorial, and a discussion of basic features.

* Chapter 1 describes what a simulation program consists of and the methodology used in modeling the system. Simon abstractions (objects, ports, connections, and messages) are briefly described along with Simon's support for hierarchy.

* Chapter 2 is a tutorial of how to construct a Simon simulation program using only a few basic features. An overview of simulation execution is given, along with an explanation of global variables and data types used by Simon. Then Simon procedures typically used by objects are discussed, along with an example. And finally, Simon procedures generally used by the main program are presented, followed by a main program example.

* Chapter 3 describes all of the basic features of Simon by topic, including: features to support hierarchy, dynamic reconfiguration of the system, additional message handling features, identifying the arrival port using input port tags, Simon's double precision clock, memory management features, and multiple-run support.

Part II of the manual discusses advanced features.

1. Simon Fundamentals

This chapter first describes the composition of a Simon simulation program. It then discusses the methodology supported by Simon used to model a system.

1.1. A Simon Simulation Program

The Simon II kernel consists of a collection of subroutines which control the execution of the simulation and provide monitoring and debugging tools. The Simon kernel is linked together with other procedures to form a simulation program. A simulation program consists of:

* A main procedure,
* Object procedures,
* The Simon kernel, and
* Support procedures (optional).

The main procedure is responsible for creating the initial configuration of objects for the system being modeled and starting the simulation. Each object procedure is responsible for simulating the behavior of the object. The main procedure is provided by the user, while object procedures may be provided either by the user or obtained from object libraries. The Simon kernel (which is the subject of this report) includes facilities for creation and manipulation of objects, time-multiplexed execution of object procedures, synchronization of communication between objects, and debugging and monitoring facilities. Additional support procedures may also be used if necessary to provide specialized services and to provide a customized interface between object procedures and the Simon kernel.

1.2. Simulation Methodology

Simon is based on autonomous objects which communicate by exchanging messages. Each object defines a number of ports through which all interactions with other objects must pass. Connections between ports specify explicitly which objects interact with which other objects.

1.2.1. Objects: Partitioning the System

The system being modeled is composed of some number of concurrent entities. For example, a multi-computer system might be divided into a collection of micro-computers and an interconnection switch, such as a global bus. The most natural partitioning of this system is to create one instance of a switch object to model the bus, and multiple instances of a processor object, one for each micro-computer. This partitioning:

1. allows models for different types of processors and interconnection switches to be easily substituted, facilitating comparisons, and

2. allows the processors and interconnection switches to be simulated at different levels of abstraction or granularity.

Thus one can envision physically dividing the system into some number of components and defining an object to represent each one. An object is defined by an object procedure. When an object is instantiated, the object procedure is passed initialization parameters and begins execution. The procedure first performs initialization, such as the creation of ports, and then begins simulating the behavior of the object.

1.2.2. Ports and Connections

All interactions between each object and its external environment are through time-stamped messages. A message is sent to an object's output port where it propagates through a connection to an input port of another object.

Distinct ports are used to transmit different types of information. No restrictions are placed on the number or type of ports an object can create. Although communicating objects must agree on the type and format of information transmitted through the ports, objects do not in general know which or even how many other objects they are communicating with. This is because each connection from an output port to a neighboring object's input port is typically made by a procedure external to the object (such as the main procedure). This increases the autonomy of each object and facilitates arbitrary interconnections of objects.

Any output port may be connected to any accessible input port regardless of how

many other connections have already been established to either port, or on which object the port resides. Messages from several output ports may be merged by connecting them to a single input port. Messages from these ports will individually arrive at the input port in time-stamped order. Conversely, an output port may fanout to several input ports, implementing broadcast or multicast communications. In this case, Simon generates as many copies of the message as are needed. This latter capability may be used by a statistics gathering object to "eaves drop" on communications between other objects and monitor their interactions, transparent to the objects being observed.

It is also possible for an object to have one of its output ports connected to one of its input ports so that it can send messages to itself. This would correspond to events confined to the internal operation of the object which must be properly synchronized in time. For example, an object may need to "wake itself up" periodically to perform some action.

1.2.3. Messages

Messages typically carry requests, commands, or status information, and trigger some activity in the receiving object. All messages have a time stamp and a data record. Simon provides mechanisms to create, delete, send, and wait for messages. While some messages will be created in one object and consumed immediately after being sent to another object, other messages may persist for relatively long periods of time as they are forwarded from one object to another. An example of the latter situation arises when Simon messages model data packets transmitted through a store-and-forward communication network.

Each message carries a time stamp which corresponds to the time at which the event modeled by the message occurs in the real system. These events are simulated by Simon in correct time sequence in order to ensure that the behavior of the simulation faithfully models the real system.

The length of the data portion of each message is defined when the message is created. Objects exchanging messages must agree on the contents and format of this data.

1.2.4. Support for Hierarchy

An object may be defined in terms of other objects to provide a hierarchical description of the system. Such an object is called a "structured" object. Objects which are not structured are referred to as "atomic" objects. For example, in figure 1-1, objects X and Y are "structured" objects. X is composed of objects A and B, plus a "shell" (which takes on the same name as the structured object, i.e. X). Y is composed of C, D, and E, and its shell Y. Objects A, B, C, D, and E may be either atomic or structured objects. Thus, a structured object consists of a collection of encapsulated objects and a shell.

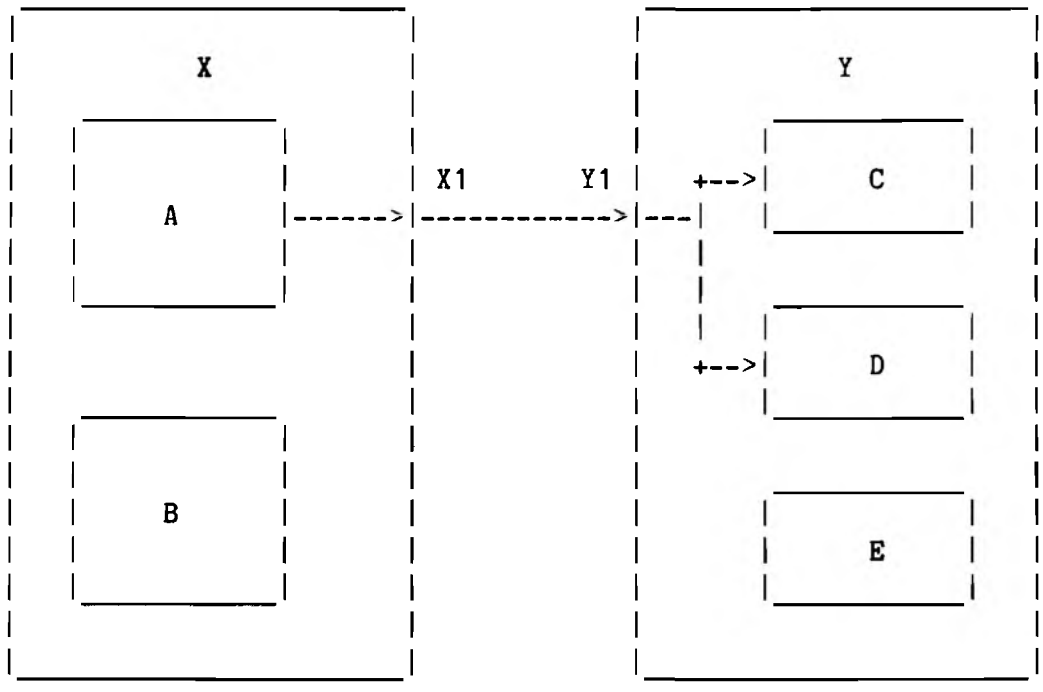

Figure 1-1: Example of Structured Objects

Connections from encapsulated objects cannot cross shell object boundaries. However, a shell object may define one or more "shell ports", so that internal objects may be connected to them. A shell port provides a tie point between internal and external connections. In figure 1, X1 and Y1 are shell ports. Thus connections can be made between objects X and Y without knowing about the inner objects A, B, C, D, or E, allowing details of the simulation model for X and Y to be hidden.

Instantiation of object X requires the instantiation and interconnection of objects A

and B. Also, parameters passed to X must be converted to parameters required by A and B. Simon provides such a mechanism for creating "structured" objects. Instantiation of any object causes the object procedure to begin execution. In a structured object, the shell object procedure instantiates the component objects and the internal connections between them. In this way any number of levels of object hierarchy can be created.

When a structured object is instantiated, mechanisms are required to:

1. Establish connections between the shell port and ports of objects INTERNAL to the shell.

2. Establish connections between the shell port and ports of other objects EXTERNAL to the shell.

These functions are performed using the same mechanism that is used to make connections between ports of atomic objects. The only difference is that connections to shell ports can be made to the "inside" as well as to the "outside" of the object. External to the structured object, a shell port appears to function like any other port, thus hiding internal connections from the external environment. In figure 1 for example, shell output port X1 of object X, can be connected to shell input port Y1 of object Y, independent of any connections internal to X or Y.

Even though shell ports are indistinguishable from regular ports to the external environment, messages are handled differently by shell ports than by regular ports. When a message arrives at a regular port, the object receives the message and performs some action. However, when a message arrives at a shell port, Simon automatically passes it through the shell and on to subsequent ports to which the shell port is connected.

2. Simon Interface Tutorial

This chapter is a tutorial on the construction of simple simulation programs using Simon. For the sake of simplicity, only a few basic Simon features are discussed. Nevertheless, significant simulation programs can be constructed from these few features.

Simon II uses the programming language Modula 2. Unlike C, the language used for Simon I, Modula 2 provides support for data abstraction and coroutines, both of which are used extensively in the simulator.

2.1. Simulation Overview

When a simulation begins, the main procedure configures the structure of the system by calling Simon routines to instantiate and interconnect objects. For each instantiated object, a process (a Modula 2 co-routine) is created to execute the associated object procedure. After the system has been configured, the main procedure starts the simulation by calling a Simon procedure called "Simulate". Simon then manages the overall execution of the simulator, updating the global clock, deciding which objects execute when, and synchronizes message passing between objects. At the conclusion of the simulation, Simon returns to the main procedure at which time the program usually terminates.

To create a simulation program, one must understand how main procedures and object procedures are constructed. But before describing these features, it is necessary to understand some of the global variables and data types exported by Simon.

2.2. Global Variables, Data Types, and Sizes

Simon exports a number of global variables and several data types for access by user procedures. However, only a few are typically needed. One global variable of particular interest is CurTime (type CARDINAL), which is the global clock. This variable is automatically set to the current simulation time by Simon, and may be read by user procedures. There are three Simon-defined data types and three corresponding pointer types of interest to the user. However, the pointer types will be used most often. These data structure types and their corresponding pointer types are:

1. Object, for representing an object, and pObject for a pointer to an object;

2. Port, for representing a port, and pPort for a pointer to a port; and

3. Msg, for representing a message, and pMsg for a pointer to a message.

A user will typically use all three pointer types (i.e.: pObject, pPort, pMsg). However, of the three data structure types, only Msg is usually needed.

Several Simon procedures require sizes to be specified (such as the size of a message which is to be created). The units of size is not specified by Simon. Rather, Simon expects sizes in the same units that are used by TSIZE. The only exception is "wkspcsiz" in the call to MakObject, where the units are WORDs.

2.3. Object Procedures

The user must supply an object procedure for each kind of object he wishes to instantiate. An object procedure must conform to the following definition.

Object Definition

```
PROCEDURE <object procedure> (
                        param:          ADDRESS         );
```

param is the address of the parameter record.

The procedure creating this object can pass a parameter record (i.e. a block of contiguous memory) to the object procedure. This record can contain various parameterization values which are intended to customize the behavior of the object. Simon makes a copy of the parameter record, and passes the address of this copy to the object procedure. It is generally NOT a good idea to use pointers in the parameter record, since the values pointed to could be changed by other objects and/or the main procedure at unpredictable times. However, Simon will not generate an error if this is done. The procedure then begins execution as a process.

2.3.1. Object Procedure Initialization

The primary task which the object procedure must perform as it begins execution is to establish its interface to the external environment. This is done by creating various input and output ports as needed by the object. Procedures for creating these ports are described below.

Port Creation

```
PROCEDURE MakOPort ( name:          ARRAY OF CHAR     ) : pPort;
```

name	is the external (string) name of the port, which should be unique with respect to other port names of the same local object.
MakOPort	returns a pointer to the created output port.

```
PROCEDURE MakIPort ( name:          ARRAY of CHAR;
                     server:        SrvrPrc;
                     contxt:        ADDRESS           ) : pPort;
```

name	is the external (string) name of the port, which should be unique with respect to other port names of the same local object.
server	is an input server procedure (see Advanced Features section).
contxt	is the address of a context record for the input server (see the Advanced Features section).
MakIPort	returns a pointer to the created input port.

MakOPort creates an output port. The argument is the name used by the external environment to uniquely identify this port from other ports residing on this same object. It therefore must be different from the names used for other ports on this object. MakOPort returns a pointer to the newly created output port which will be needed later by the object procedure for sending messages out of this port. MakIPort creates an input port. The first argument is identical to the argument of MakOPort. The second and third arguments however, are unique to input ports. They specify an input server procedure and its corresponding context. For simple simulations, NilSrvr (exported by Simon) and Nil should be specified, respectively. MakIPort returns a pointer to the newly created input port which is usually not needed by the object procedure. Although if desired, it could later be used to determine if a received message arrived at this particular port or not.

After the object procedure has created the necessary ports and completed its initialization, it is then ready to start simulating the behavior of the object.

2.3.2. Simulation of Object Behavior

The behavior of the object is simulated explicitly by the object procedure code. This code should accurately simulate the object's behavior to the level of detail required by this simulation, or any future simulation that might use this object. When the object's behavior requires interaction with the external environment, the relevant data is transmitted or received using messages. To send a message, the message must first be created using MakMsg.

Message Creation

PROCEDURE MakMsg (datasize: CARDINAL) : pMsg;

datasize	is the size (in TSIZE units) of the data portion of the message.
MakMsg	returns a pointer to the message.

The size of the data record to be sent is passed to MakMsg in datasize. MakMsg then creates a message large enough to contain the message header information (which Simon automatically initializes), as well as the data record. A pointer to the message is then returned to the object procedure. The format of the data record is defined by the user. The data is then placed into the data record of the message by the object procedure. The data type "Msg", and the type for the data record can be useful in doing this. For example, suppose that an integer (I) and a real (X) are to be sent in a message. The following type and variables could be defined.

```
TYPE DataRecord = RECORD
    IValue:  INTEGER;
    XValue:  REAL;
END;

DataRec:  POINTER TO DataRecord;
mesg:     pMsg;
```

To create a message and place the values of I and X in it:

```
mesg:=MakMsg(TSIZE(DataRecord));

DataRec:=ADR(mesg^.Data);
WITH DataRec^ DO
    IValue:=I;
    XValue:=X;
END;
```

After the message has been created and the data placed in it, it can be sent to the external environment. This can be done using Send.

Message Transmission

```
PROCEDURE Send (    mesg:       pMsg;
                    prt:        pPort;
                    timeincr:   CARDINAL        );
```

mesg	is a pointer to the message to be sent.
prt	is a pointer to an output port to which the message is to be sent.
timeincr	is the time interval (from now, i.e. CurTime) at which the message is to arrive.

Send has three arguments. The first is mesg, which is a pointer to the message to be sent (returned by MakMsg). The second is prt, which is a pointer to the output port that the message is to be sent out of (returned by MakOPort). The third argument is the time increment. This specifies how may clock ticks from now (CurTime) that the message is to be received. This must be a positive value or zero. Simon then takes the message and places it in an internal queue where it waits to be delivered at the specified time. After a message is sent, it should not be accessed or sent again by the sender. (Exceptions are described in the Advanced Features section of this manual.)

To receive a message, the object procedure must wait (allow CurTime to advance) until a message destined for the object is scheduled to arrive. The procedure Wait accomplishes this.

Waiting to Receive a Message

```
PROCEDURE Wait (    VAR mesg:    pMsg;
                    VAR dat:     ADDRESS    );
```

mesg — is a pointer to the newly arrived message.

dat — is a pointer to the data record of the message.

Wait suspends execution of the object procedure until a message (any message) is delivered to the object. When a message does arrive, execution of the object procedure resumes at the next executable statement following the Wait. The object procedure can access the message by moving "dat" into an appropriate pointer variable (in accordance with Modula 2 record access rules), in a manner similar to the above example where values were placed into a message. At this point the object procedure does not know where the message came from. The input port at which the message arrived is not explicitly passed to the object procedure. If this is needed, the object procedure can access this information in the header of the message (mesg^.DstPrt), and can compare it with pointers to its own input ports (see MakIPort, above). (See also: Input Port Tags.) When the object procedure has finished processing the message, it must dispose of the message either by forwarding it or by deleting it. Messages are forwarded by simply using Send. (Since the message has already been created once, MakMsg is not needed again.) Messages are deleted with DelMsg.

Message Deletion

```
PROCEDURE DelMsg (    mesg:    pMsg    );
```

mesg — is a pointer to the message to be deleted.

DelMsg has only one argument, a pointer to the message to be deleted. DelMsg simply frees the memory space used by the message.

2.3.3. Object Procedure Example

With the above procedures, object procedures for simple simulations can be written. As an example, the object procedures for a simple statistics experiment will be presented. Consider an experiment where a ball is dropped through a maze of pegs, to ultimately land in a slot (see figure 2-1, below).

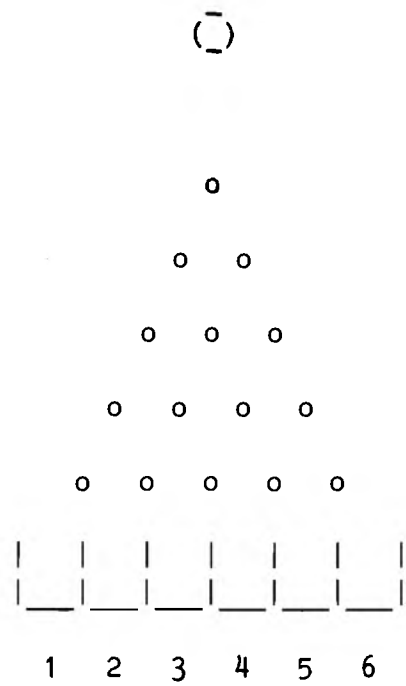

Figure 2-1: Simple Statistics Experiment

The ball is dropped over the top center peg. The ball lands on the peg and rolls off either to the right or to the left. In either case the ball will land on a peg at the next level and again either roll to the right or to the left. This process continues until the ball falls through the maze and lands in a slot. This system can be modeled using a message to represent each ball, and three kinds of objects. First, an object is needed to release balls into the system. Secondly, an object is needed to represent each of the pegs, receiving a message (ball) and randomly sending it to the right or to the left (output port). And thirdly, an object is needed to represent each of the slots, reporting whenever it receives a message (ball).

The first object procedure, Dispenser, is designed to accept two values in the parameter record, the number of balls to be released and the time interval between the releasings.

```
PROCEDURE Dispenser (param: ADDRESS);
(*
TYPE
    DropParam = RECORD          (* Format of parameter record *)
        Total:    CARDINAL;     (* Number of balls to be dropped *)
        Interval: CARDINAL;     (* Time between dropping each ball *)
    END;
    pDropParam = POINTER TO DropParam;
*)
VAR
    params:   pDropParam;       (* For accessing parameter values *)
    ball:     pMsg;             (* Pointer to the ball *)
    drop:     pPort;            (* The ball drop port *)
    droptime: CARDINAL;         (* The drop time of the ball *)

BEGIN
    drop:=MakOPort("Ball Drop"); (* Create the ball drop port *)
    params:=param;              (* Access parameter values *)
    WITH params^ DO
        droptime:=0;
        FOR I:=1 TO Total DO    (* Count the balls dropped *)
            ball:=MakMsg(0);              (* Create the ball *)
            Send(ball, drop, droptime);   (* Dispense it *)
            droptime:=droptime+Interval;  (* Get next drop time *)
        END;
    END;
    RETURN;                     (* Nothing else to do, so quit *)
END Dispenser;
```

Dispenser uses the parameter values to calculate when each ball is to land on the top peg. It dispenses all of the balls at once, scheduling each one to arrive at the appropriate time. An alternative approach could have been to dispense the balls one at a time. In this case, Dispenser would require an additional output port and an input port connected together (i.e. a connection to itself). After dispensing a ball, Dispenser would then send a message out of this output port using Interval as the "timeincr". Dispenser would then wait for this "delay" message to arrive before dispensing the next ball. Obviously, there are a variety of approaches that can be used for any given system. The best approach to use is the one that best meets the needs of the simulation application.

The second object procedure, Peg, has only one parameter value, the time it takes for a ball to roll off of the peg and land on the next peg or slot. To determine which way

the ball is to roll (right or left), Peg uses a simple random number generating procedure, RandCard, which selects a random cardinal value from a uniform distribution each time it is called.

```
PROCEDURE Peg (param: ADDRESS);

VAR
    right, left:  pPort;          (* Right and left output ports *)
    ball:         pMsg;           (* Pointer to the ball *)
    dat:          ADDRESS;        (* Dummy parameter *)

BEGIN                                          (* Create ports *)
    right:=MakIPort("Peg Top", NilSrvr, Nil); (* Entrance *)
    right:=MakOPort("Peg Right");              (* Exit right *)
    left:=MakOPort("Peg Left");                (* Exit left *)
    LOOP
        Wait(ball, dat);          (* Wait for next ball to arrive *)
        IF ODD(RandCard) THEN     (* If the random number is odd *)
            Send(ball, left, CARDINAL(param^));    (* send ball left *)
        ELSE                                       (* otherwise *)
            Send(ball, right, CARDINAL(param^));   (* send ball right *)
        END;
    END;
END Peg;


PROCEDURE RandCard(): CARDINAL;

BEGIN
    NextRand:=NextRand DIV 2;
    IF NextRand = 0 THEN
        NextRand:=Seed;
    END;
    RETURN NextRand;
END RandCard;
```

Notice that the pointer to the input port was not needed, so it was discarded (overwritten by MakOPort in the next statement). The main body of the procedure is a loop which waits for a message (ball) to arrive. When one does arrive, the random number generator is used to determine whether to send it out of the left port or the right port. In either case the ball is sent to arrive at the next peg or slot a certain amount of time in the future as specified by the parameter record.

The third object, Slot, has one parameter value, a slot number. When a ball arrives, Slot announces the arrival along with its slot number.

```
PROCEDURE Slot (param: ADDRESS);

VAR
    pt:     pPort;        (* Needed only to create the input port *)
    slotno: CARDINAL;     (* Slot number *)
    ball:   pMsg;         (* Pointer to the ball *)
    dat:    ADDRESS;      (* Dummy parameter *)

BEGIN
    pt:=MakIPort("Ball Slot", NilSrvr, NIL); (* Create the slot port *)
    slotno:=CARDINAL(param^);
    LOOP
        Wait(ball, dat);              (* Wait for next ball to arrive *)
        WriteF(Output, "A ball landed in slot #%d at time %d.\n",
            slotno, CurTime);         (* Announce arrival of the ball *)
        DelMsg(ball);                 (* Dispose of the message *)
    END;
END Slot;
```

The main body of the procedure is similar to Peg. It is a simple loop which waits for a message (ball) to arrive, then reports this fact and disposes of the message. Note that the statement reporting the arrival is dependent on the Modula 2 I/O implementation. (For this very reason, I/O was purposely excluded from the Simon module.)

The above three object procedures provide all the needed building blocks for constructing a simulation program. The only thing missing is the main procedure.

2.4. The Main Procedure

The main procedure is responsible for configuring the structure of the system and starting the simulation. It accomplishes these tasks by calling the appropriate Simon procedures. This section first discusses the Simon procedures which are used to configure the system, and then how the simulation is started.

2.4.1. System Configuration

Configuring the system consists of instantiating objects and then interconnecting them. To instantiate an object, a Simon procedure called MakObject is used.

Object Creation

```
PROCEDURE MakObject ( name:        ARRAY of CHAR;
                      proc:        ObjPrc;
                      wkspcsiz:    CARDINAL;
                      param:       ADDRESS;
                      paramsize:   CARDINAL;
                      objectowner: pObject      ) : pObject;
```

name	is the external (string) name of the object, which does not have to be unique.
proc	is the object procedure.
wkspcsiz	is the size (in WORDs) of the work space to be allocated for this object's process.
param	is the address of the parameter record to be passed to the object.
paramsize	is the size (in TSIZE units) of the parameter record. A copy of the parameter record will be passed to the object.
objectowner	is a pointer to the shell object which immediately encapsulates this object. If it is to be placed at the outermost level then objectowner is System (Nil).
MakObject	returns a pointer to the created object.

MakObject creates a process (a Modula 2 co-routine) to execute the object procedure. MakObject allows the object procedure to execute immediately so that the object will be able to create its ports and to initialize its variables. After the object relinquishes control (by calling Wait or returning), MakObject returns to its caller. MakObject has six arguments. The first argument, name, is the external name used by error reporting and debugging facilities for referring to this object. For this reason, it is a good idea to make "name" unique to avoid confusion, even though Simon does not require uniqueness. The second argument, proc, is the object procedure which simulates the behavior of the object. Note that any number of objects can be created using the

same object procedure. Each object will have its own process and work space in memory, even though the object procedure code may be shared. The third argument, wkspcsiz, specifies how many WORDs should be provided for the work space. A work space will contain variables local to the object procedure as well as the run-time stack for the process, though the exact details are implementation dependent. (Typically, a work space requires a few hundred to a few thousand WORDs.) The fourth and fifth arguments, param and paramsize, are a pointer to a parameter record and its size, respectively. A parameter record, as defined by the respective object procedure, allows the main procedure to supply parameterization values to customize the particular object being instantiated. The sixth argument, objectowner, is a pointer to the immediately encapsulating object of the object being created. Its purpose is for hierarchal support. For simple simulations not using hierarchy, this should be specified as "System" (which is a constant equated to Nil). MakObject returns a pointer to the newly created object, which will be needed later for interconnection.

To interconnect the ports of objects the procedure, Connect, is used.

Port Connection

```
PROCEDURE Connect (  srcprt:    pPort;
                     dstprt:    pPort      );
```

srcprt: is a pointer to the port sending messages.

dstprt: is a pointer to the port receiving messages.

Connect links two ports together so that messages flowing out of srcprt will be transmitted to dstprt. Any number of connections can be made to or from a port. Connect simply has two arguments, a pointer to the source port, srcprt, and a pointer to the destination port, dstprt. The only restrictions on connections are that they do not cross through object boundaries (such as object shells in hierarchical configurations), and that messages flow from the srcprt to the dstprt.

Typically, object procedures create ports while the main procedure interconnects them. Therefore, the main procedure will not usually have pointers to the ports it is to connect, since it did not create them. To obtain these pointers, the FindPort procedure is provided.

Port Identification

```
PROCEDURE FindPort (    owner:          pObject;
                        name:           ARRAY of CHAR     ) :  pPort;
```

owner: is a pointer to the object on which the port resides.

name: is the external (string) name of the port being sought.

FindPort returns a pointer to the port, or Nil if not found.

FindPort provides a facility to obtain a pointer to a port if provided with the external name of the port and the object on which the port resides. The first argument, owner, is a pointer to the desired object. (MakObject returns a pointer to the object it creates.) The second argument is the external name of the port being sought. If the port is found, FindPort will return a pointer to the port, otherwise FindPort outputs an error message and returns Nil.

Using the above procedures, the system can be completely configured. Once this is done, simulation execution can begin.

2.4.2. System Execution

Simon provides a procedure called Simulate, which manages the execution of the simulation.

Starting Simulation Execution

```
PROCEDURE Simulate;
```

The main procedure calls Simulate to start the execution of the simulation. Simulate then controls the entire execution until the simulation is concluded. During the simulation, Simulate schedules each object for execution whenever a message is to be delivered to that object. When the object receives the message it then continues executing until it is ready to receive a subsequent message. To wait for a message it calls Wait, which returns control to Simulate. In this manner the simulation proceeds until there are no more messages to be delivered. This marks the end of the simulation and subsequently, Simulate returns to the main procedure. At this point the main procedure usually terminates.

Typically, Simulate continues the simulation execution until the event queue is empty (no messages to be delivered). However, the simulation may be stopped at any point in simulation time by setting a "stop time". This may be done by the main procedure and/or object procedures calling StopAt.

Stopping Simulation Execution

```
PROCEDURE StopAt (    era:     CARDINAL
                      time:    CARDINAL    );
```

era	is the "era" (CurEra - see Extended Simulation Time) at which to stop the simulation.
time	is the "time" (CurTime) at which to stop the simulation.

After the "stop time" has been set, Simulate will stop (return) whenever the next message in the event queue has a time stamp greater than or equal to the "stop time". After the simulation has been stopped, it can be resumed simply by setting the "stop time" to a future time and calling Simulate. (Initially, Simon sets "stop time" to: StopAt(TicLimit,TicLimit).)

2.4.3. Main Procedure Example

As an example, a main procedure will be presented which uses the object procedures defined in the example of the previous section. As was previously mentioned, the system to be simulated consists of a ball falling through a maze of pegs which finally lands in one of several numbered slots. The main procedure could configure an arbitrarily large system of pegs and slots. However, for the sake of simplicity, the following main procedure example will create only three pegs and three slots. Although, this example shows the object procedures in the main module (for the sake of simplicity), typically they would be found in one or more other modules.

```
MODULE Example;
                                                    (* System Imports *)
    FROM SYSTEM IMPORT ADR, ADDRESS, TSIZE;
                                                    (* I/O Imports *)
    FROM IO IMPORT Input, Output, ReadF, WriteF;[1]
                                                    (* Simon Imports *)
    FROM Simon IMPORT pObject, pPort, pMsg, MakObject,
        MakOPort, MakIPort, NilSrvr, FindPort, Connect,
        MakMsg, Send, Wait, DelMsg, Simulate, CurTime;

CONST
    System = Nil;

TYPE
    DropParam = RECORD
        Total:    CARDINAL;
        Interval: CARDINAL;
    END;
    pDropParam = POINTER TO DropParam;

VAR
    Dispensr, Peg1, Peg2, Peg3, Slot1, Slot2, Slot3: pObject;
    DropParams: DropParam;
    Seed, NextRand: CARDINAL;
    I, one, two, three, five: CARDINAL;
```

< Insert the object procedures here, for Dispenser, Peg, RandCard, and Slot. >

[1]These I/O procedures are provided with the DEC WRL implementation of Modula 2. They are similar to scanf and printf in "C".

```
BEGIN
    NextRand:=0;                (* Initialize RandCard variables *)
    Seed:=31380;
    one:=1;                     (* Initialize "constant" parameters *)
    two:=2;
    three:=3;
    five:=5;

    WITH DropParams DO
        WriteF(Output, "\nEnter # of balls to be dropped and ");
        WriteF(Output, "time interval between dropping: ");
        I:=ReadF(Input, "%d%d", Total, Interval);
        WriteF(Output, "\n");
    END;
    Dispensr:=MakObject("Ball Dispenser", Dispenser, 200, ADR(DropParams),
        TSIZE(DropParam), System);
    Peg1:=MakObject("Top Peg", Peg, 200, ADR(five), TSIZE(CARDINAL),
        System);
    Peg2:=MakObject("Left Peg", Peg, 200, ADR(five), TSIZE(CARDINAL),
        System);
    Peg3:=MakObject("Right Peg", Peg, 200, ADR(five), TSIZE(CARDINAL),
        System);
    Slot1:=MakObject("First Slot", Slot, 500, ADR(one), TSIZE(CARDINAL),
        System);
    Slot2:=MakObject("Second Slot", Slot, 500, ADR(two), TSIZE(CARDINAL),
        System);
    Slot3:=MakObject("Third Slot", Slot, 500, ADR(three), TSIZE(CARDINAL),
        System);
    Connect(FindPort(Dispensr, "Ball Drop"), FindPort(Peg1, "Peg Top"));
    Connect(FindPort(Peg1, "Peg Left"), FindPort(Peg2, "Peg Top"));
    Connect(FindPort(Peg1, "Peg Right"), FindPort(Peg3, "Peg Top"));
    Connect(FindPort(Peg2, "Peg Left"), FindPort(Slot1, "Ball Slot"));
    Connect(FindPort(Peg2, "Peg Right"), FindPort(Slot2, "Ball Slot"));
    Connect(FindPort(Peg3, "Peg Left"), FindPort(Slot2, "Ball Slot"));
    Connect(FindPort(Peg3, "Peg Right"), FindPort(Slot3, "Ball Slot"));
    WriteF(Output, "Simulation beginning\n");
    Simulate;
    WriteF(Output, "Simulation completed\n");
END Example.
```

3. Basic Simon Interface

In addition to the procedures described in the "Tutorial" chapter, the following procedures complete the description of the basic Simon kernel interface. Advanced features will be discussed in a later chapter.

3.1. Support for Hierarchy

As was briefly mentioned in the "Fundamentals" chapter, shell ports provide support for hierarchy. MakSOPort and MakSIPort create shell output and input ports, respectively. The only difference between shell output and shell input ports is the direction of message flow. A shell output port passes messages from inside the shell to the outside, while a shell input port passes messages from the outside of the shell to the inside.

3.1.1. Shell Ports

PROCEDURE MakSOPort (name: ARRAY OF CHAR) : pPort;

name	is the external (string) name of the port, which should be unique with respect to other port names of the same local object.
MakSOPort	returns a pointer to the created shell output port.

PROCEDURE MakSIPort (name: ARRAY of CHAR) : pPort;

name	is the external (string) name of the port, which should be unique with respect to other port names of the same local object.
MakSIPort	returns a pointer to the created shell input port.

Creating a hierarchical structure is straight-forward. Consider the following simple example of a shell containing one object and having two connections.

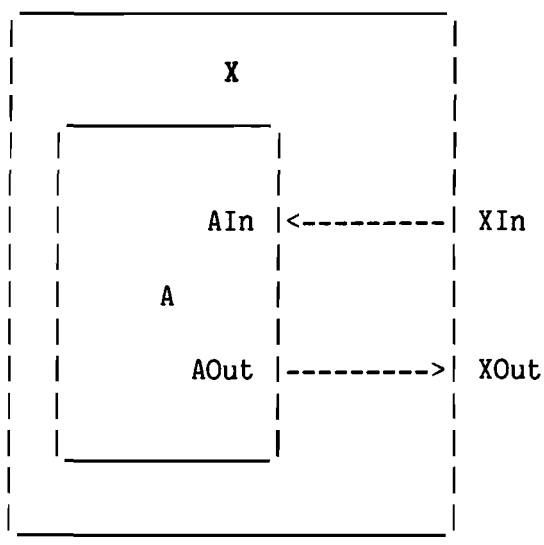

Figure 3-1: A Simple Example of Hierarchy

Assume that the object procedure for "A" is called AProc and the object procedure for "X" is called XProc. It is also assumed that no parameters need to be passed to "A" when it is instantiated. XProc would simply consist of:

```
PROCEDURE XProc(param: ADDRESS);

VAR
    in, out:  pPort;
    a:        pObject;

BEGIN
    a:=MakObject("A", AProc, 500, Nil, 0, CurObj); (* Create object "A" *)
    in:=MakSIPort("XIn");                          (* Make shell ports *)
    out:=MakSOPort("XOut");
    Connect(in, FindPort(a, "AIn"));               (* Connect ports *)
    Connect(FindPort(a, "AOut"), out);
END XProc;
```

XProc first creates the inner object "A" by calling MakObject. Note that the "objectowner" parameter is CurObj, which points to the currently executing object, X. (CurObj is a global variable exported by Simon. See Global Variables in the Advanced Features section.) Then XProc creates its shell ports, "XIn" and "XOut". (Actually these could be done in any order.) At this point, the two connections can be made, since all ports have been created. Finally, XProc terminates. Even though the XProc process (co-

routine) no longer exists after termination, the shell structure and shell ports remain intact.

3.2. Dynamic Reconfiguration

Typically, the main procedure deals with the structure of the simulation model, while object procedures deal with the behavior of each object. However, this need not be the case. Simon allows an object procedure to change the simulator's structure after execution has begun.

In addition to the procedures discussed in the tutorial, Simon provides procedures for disconnecting, deleting, and searching for ports, and moving and deleting objects. In using some of these procedures, special considerations need to be given to messages which have sent but not yet received.

3.2.1. Port Disconnection

The call to the procedure to disconnect two ports is similar to Connect and has the same arguments. For applications which will be dynamically disconnecting and reconnecting ports, it is important to keep in mind that a message is not bound to a destination until the time the message is to be delivered. Therefore, any pending message is delivered to the ports which are connected at the time the message is to be received.

```
PROCEDURE Disconnect ( srcprt:      pPort;
                       dstprt:      pPort          );
```

srcprt: is a pointer to the message source port (from).

dstprt: is a pointer to the message destination port (to).

3.2.2. Port Deletion

When a port is deleted, all connections to that port are also deleted. When an output port is deleted, any pending messages from that port are lost.

```
PROCEDURE DelPort (    prt:          pPort          );
```

prt	is a pointer to the port to be deleted.

3.2.3. Searching for Ports

To supplement the FindPort procedure described earlier, Simon provides procedures to specifically search for input (FindIPort) or output (FindOPort) ports. These procedures function in a manner similar to FindPort.

```
PROCEDURE FindIPort (  owner:        pObject;
                       name:         ARRAY of CHAR    ) : pPort;
```

owner:	is a pointer to the object on which the port resides.
name:	is the external (string) name of the port being sought.
FindIPort	returns a pointer to the input port, or Nil if not found.

```
PROCEDURE FindOPort (  owner:        pObject;
                       name:         ARRAY of CHAR    ) : pPort;
```

owner:	is a pointer to the object on which the port resides.
name:	is the external (string) name of the port being sought.
FindOPort	returns a pointer to the output port, or Nil if not found.

Simon also provides "silent" versions of the three FindPort procedures. These are identical to the original versions except that they never output an error message if a port cannot be found. They can be useful in writing code which automatically adjusts to configuration variations.

```
PROCEDURE FindPortS (  owner:        pObject;
                       name:         ARRAY of CHAR    ) : pPort;
```

owner:	is a pointer to the object on which the port resides.
name:	is the external (string) name of the port being sought.
FindPortS	returns a pointer to the port, or Nil if not found.

```
PROCEDURE FindIPortS ( owner:        pObject;
                       name:         ARRAY of CHAR    ) : pPort;
```

owner: is a pointer to the object on which the port resides.

name: is the external (string) name of the port being sought.

FindIPortS returns a pointer to the input port, or Nil if not found.

```
PROCEDURE FindOPortS ( owner:        pObject;
                       name:         ARRAY of CHAR    ) : pPort;
```

owner: is a pointer to the object on which the port resides.

name: is the external (string) name of the port being sought.

FindOPortS returns a pointer to the output port, or Nil if not found.

3.2.4. Object Relocation

Normally, when an object is moved, all connections to external objects are deleted, while internal connections remain intact. The whole object, including any internal structure, is moved intact into the "newowner" object. New connections need to be made explicitly (using Connect). All pending messages will be delivered to the newly connected destinations. Normal moves (as just described) are performed by MovObject.

```
PROCEDURE MovObject ( objct:     pObject;
                      newowner:  pObject          );
```

objct is a pointer to the object to be moved. If the object is a structured object then the internal structure is moved, too.

newowner is a pointer to the shell object in which the object is to be placed. If it is to be placed at the outermost level then the newowner is System (Nil).

MovObjectP is identical to MovObject except that all connections are preserved (even across shell boundaries). The intent of MovObjectP is to provide a means of moving several connected objects without having to reconnect them after the move.

```
PROCEDURE MovObjectP (objct:      pObject;
                      newowner:   pObject      );
```

objct — is a pointer to the object to be moved. If the object is a structured object then the internal structure is moved, too.

newowner — is a pointer to the shell object in which the object is to be placed. If it is to be placed at the outermost level then the newowner is System (Nil).

3.2.5. Object Deletion

When an object is deleted, all external connections are also deleted. The whole object, including any internal structure, is deleted. All pending messages are lost. It is an error for an object to attempt to delete itself or any object in which it is encapsulated.

```
PROCEDURE DelObject ( objct:      pObject      );
```

objct — is a pointer to the object to be deleted. If the object is a structured object then the internal structure is deleted, too.

3.3. Message Handling

In addition to the procedures for creating, sending, receiving, and deleting messages which were discussed in the tutorial, Simon provides several additional procedures for manipulating messages. Before discussing these procedures, some important considerations for sending and receiving messages will be explained.

When a message is sent, a time increment is specified, which represents a time at which the message is to be received relative to the time at which it was sent. The time increment must be a value from zero to TicLimit, inclusive. (TicLimit is an implementation dependent constant which is set to the largest power of ten that can be represented by a cardinal.) At the time when the message is to be received, the connections from the output port are traversed to determine which input ports are to receive the message. It is important to remember that the input ports which will receive the message are determined by the connections from the sending object's output port AT THE TIME THE

MESSAGE IS TO BE RECEIVED.

When a message is received, the receiving object's procedure is activated and (if the receiving port is not a read-only input port) the object procedure is provided an individual copy of the message. The object procedure must eventually dispose of this copy of the message by either deleting it, OR by sending it on (forwarding it) to another object. Before forwarding a message, the fixed-size data record of the message can be directly modified. If major modification of the data record is required (changing its size), then ChgMsg should be used which replaces the data record with a new record of specified size. This differs from using MakMsg to create an entirely new message in that the latter would not preserve any of the attributes which were attached to the original message. (Attributes are discussed in the Advanced Features section of the manual.)

3.3.1. Message Modification

ChgMsg is used to change the data record of a message. ChgMsg should only be used if the size of the data record must be changed. (Otherwise, the old data record can simply be over-written.) ChgMsg creates a new message, copies the new data record to it, and then transfers attributes from the old message to the new one. Lastly, the old message is deleted and a pointer to the new message is returned. (Note: Since the old message deletion is the final step, The new data record could be specified to contain data from the old message.)

```
PROCEDURE ChgMsg (   oldmsg:     pMsg;
                     newdata:    ADDRESS;
                     newsize:    CARDINAL        ) : pMsg;
```

oldmsg	is a pointer to the message to be changed.
newdata	is the address of the new data record.
newsize	is the size (in TSIZE units) of the new data record.
ChgMsg	returns a pointer to a new (changed) message.

3.3.2. Message Duplication

If a duplicate copy of a message is needed, DupMsg can be used. DupMsg creates a duplicate copy of a message and its attributes, and returns a pointer to the new copy. The message being duplicated can be any message, including messages received on read-only input ports.

```
PROCEDURE DupMsg (  mesg:  pMsg  ) : pMsg;
```

mesg	is a pointer to the message to be duplicated.
DupMsg	returns a pointer to a new duplicate copy of the message.

3.3.3. Message Creation and Transmission

MakMsgD and SendMsg are provided as a convenience to the user. MakMsgD creates a message as MakMsg does, but it also copies the data record (of length datasize) into the message before returning with a pointer to the message.

```
PROCEDURE MakMsgD ( dat:      ADDRESS;
                    datasize: CARDINAL  ) : pMsg;
```

dat	is a pointer to the data record to be sent.
datasize	is the size (in TSIZE units) of the data portion of the message.
MakMsgD	returns a pointer to the message.

SendMsg combines the functions of MakMsgD and Send. It creates a message, copies the "dat" record into the message, and then sends the message out of the specified port.

```
PROCEDURE SendMsg (  dat:        ADDRESS;
                     len:        CARDINAL;
                     prt:        pPort;
                     timeincr:   CARDINAL      );
```

dat	is a pointer to the data record to be sent.
len	is the size (in TSIZE units) of the data record.
prt	is a pointer to an output port to which the message is to be sent.
timeincr	is the time interval (from now) at which the message is to arrive.

3.3.4. Read-Only Input Ports

In addition to regular ports, Simon allows "read-only" input ports to be created. When a message arrives at a regular input port (created by MakIPort), an individual copy of the message is created and passed to the object procedure. The object procedure can modify, forward, save (for later reference), or delete this copy of the message. However, when a message arrives at a read-only port (created by MakIRPort), no copy is made of the message. The object procedure must not modify, forward, save, nor delete the message. It can only read the message. (Although using DupMsg is permitted.) Simon will automatically dispose of the message the next time Wait is called. Read-only ports can be used to increase simulation efficiency when there is a high degree of fan-out present, since message copying is minimized.

```
PROCEDURE MakIRPort ( name:     ARRAY of CHAR;
                      server:   SrvrPrc;
                      contxt:   ADDRESS        ) : pPort;
```

name	is the external (string) name of the port, which should be unique with respect to other port names of the same local object.
server	is an input server procedure (see Advanced Features section).
contxt	is the address of a context record for the input server (see the Advanced Features section).
MakIRPort	returns a pointer to the created input (read-only) port.

3.4. Input Port Tags

When a message arrives at an input port, the object procedure resumes execution. Although the object procedure does not know which port the message arrived at, it could determine this by comparing the DstPrt field of the message header to pointers to each input port. This approach is not very efficient if there are many input ports. To simplify identifying the destination port, Simon allows the user to associate a user defined tag value (integer) with an input port. Simon provides a global variable "CurTag" (integer), which it sets from the tag value associated with the destination port. Whenever a message arrives, Simon automatically sets CurTag so that the object procedure can simply access CurTag to determine the destination port. Since the value in CurTag is an integer, it can easily be used in CASE statements or array subscripting.

3.4.1. Setting Port Tags

When an input port is created (MakIPort or MakIRPort), the port tag field is initialized to zero. The user can set or change an input port tag by calling SetPortTag. SetPortTag will then set the tag field of the specified port to the integer value passed to it.

```
PROCEDURE SetPortTag( prt:      pPort;
                      tag:      INTEGER    );
```

prt — is a pointer to the port whose tag field is to be set.

tag — is the user defined value to which the tag field is to be set.

3.5. Extended Simulation Time

The global clock is a double precision cardinal value, consisting of the exported variables CurEra and CurTime. Since double precision cardinals are not supported by Modula 2, the global clock was designed to facilitate being output as a decimal:

Global Time = CurEra * TicLimit + CurTime,

where TicLimit is the largest power of ten that can be represented by a cardinal variable and CurTime is always less than TicLimit. For example, on the VAX, TicLimit is 10^9. However, for all but very long simulations, a single precision clock is adequate and CurEra can be ignored.

3.6. Memory Management

Before discussing memory allocation, some important issues related to memory use should be discussed. First of all, the user should be aware of things which affect the workspace size of objects. If an object procedure requires alot of workspace then the number of objects which can be instantiated is reduced due to main memory limitations of the host computer system. It is therefore advantageous to minimize the amount of workspace required by an object procedure. An object's workspace contains local variables as well as the run-time stack. An object procedure might use alot of memory for local variables. If every instantiation of the object actually uses all of the local variables, then there is no simple alternative. However, if each instantiation only uses a few of the local variables, then it might be better to have the object dynamically allocate memory for the variables that it will actually use. Another source of difficulty may come from procedures which the object procedure calls. These procedures may be recursive or just produce calls several levels deep. Additionally, each procedure may allocate temporary local variables on the stack. Any of these could cause the object to require substantial amounts of workspace. If the procedures called by the object procedure

cannot be pruned to use less memory, then one alternative is to create a centralized object which calls the procedures. Secondly, although Modula allows global variables to be defined, the user should be wary of using them in object procedures. For each object instantiation, a unique set of local variables is allocated. However there will be only one universal copy of each global variable and each object instantiation will be accessing that same variable. Although this might be the desired result in some cases, it can also be a major source of problems if the user is not aware of the situation.

3.6.1. Memory Allocation / Deallocation

In order to minimize the amount of memory required, Simon provides mechanisms to allow the user to explicitly control memory use. To provide the basic allocation and deallocation functions, Simon provides two procedures, GetMem and FreeMem. They are similar to Allocate and DeAllocate, (provided by Modula) and have identical arguments. (Note: GetMem and FreeMem round up size to the next whole number of "granules" (see Installation Notes).) However typically, GetMem and FreeMem perform two to three times faster than Allocate and DeAllocate. Although, it should be noted that GetMem and FreeMem do not provide bounds checking capability, and if they are to be used, array bounds checking and pointer checking should be disabled when compiling. Nevertheless, GetMem and FreeMem do provide a substantial performance improvement, especially when used for transient and relatively smaller allocations. In addition, GetMem and FreeMem are compatible with Allocate and DeAllocate, allowing both to be used in the same program.

```
PROCEDURE GetMem ( VAR memory:    ADDRESS;
                   size:          CARDINAL      );
```

memory — is the address returned of the allocated memory block.

size — is the length (in TSIZE units) of the requested memory block.

```
PROCEDURE FreeMem ( memory:       ADDRESS;
                    size:         CARDINAL      );
```

memory — is the address of the memory block to be deallocated.

size — is the length (in TSIZE units) of the memory block.

3.6.2. Object Termination Clean-Up

GetMem and FreeMem are sufficient for an object procedure to manage its own memory usage. However, if the object were to be deleted (by another object), then Simon would have no way of knowing about any memory which had been dynamically allocated by the object. Since the object itself would be most knowledgeable of its own dynamic data structures, it should be the one to deallocate this memory. This capability is provided by Simon by the "Clean-Up" mechanism. In order for the user to take advantage of this capability, one (or more if desired) "Clean-Up" procedures must be supplied by the user. When the object is to be deleted, Simon will automatically call each of the CleanUp procedures defined by the user for that object. After all CleanUp procedures have finished, Simon will delete the object. A CleanUp procedure is defined as follows.

```
PROCEDURE <clean-up procedure> (
                        param:          ADDRESS         );
```

param — is the address of what is to be cleaned up.

The parameter, "param", is specified by the user, and typically is a pointer to the head of the object's dynamic data structure. In order for Simon to be able to call a CleanUp procedure, Simon must be notified of its existence. This is done by calling MakCleanUp. MakCleanUp allows the user to associate a CleanUp procedure (proc) and a pointer to what is to be "cleaned up" (param), with the currently executing object. An object may have any number of CleanUp procedures associated with it. In fact, the same procedure may be used more than once by an object (although, the "param" pointer will typically be different in each instance).

```
PROCEDURE MakCleanUp (
                        proc:           ClnUpPrc;
                        param:          ADDRESS         );
```

proc — is a "clean-up" procedure for the currently executing object.

param — is the address of what to clean-up.

A second procedure, MakCleanUpR, allow a CleanUp procedure to be associated

with a remote object. It is equivalent to MakCleanUp except that the specified remote object is used instead of the currently executing object (CurObj).

```
PROCEDURE MakCleanUpR (
                        proc:       ClnUpPrc;
                        param:      ADDRESS
                        ob:         pObject          );
```

proc	is a "clean-up" procedure.
param	is the address of what to clean-up.
ob	is a pointer to the remote object.

3.7. Miscellaneous Procedures

Three miscellaneous procedures provided by Simon are ResetClock, Reset, and Copy. ResetClock and Reset are provided to support multiple runs, and Copy is a general purpose utility.

3.7.1. Multiple Run Support

At the completion of a simulation run, Simulate returns to the main procedure with the system configuration intact. If it is necessary to perform another simulation ResetClock or Reset can be called to prepare for another run.

```
PROCEDURE ResetClock;

PROCEDURE Reset;
```

ResetClock simply zeroes the simulation clock and clears the message (event) queue, while Reset completely deletes the current system configuration, as well. If the same configuration is to be used for a subsequent run, ResetClock can be called, a "start-up" GSM can be sent (see General System Messages), and then Simulate can be called. However, if a different configuration is needed, Reset can be called to delete the old configuration, and a new configuration can then be constructed before calling Simulate. (Note: Attribute types and GSM types (see Advanced Features section) are not deleted by Reset and can be reused, if desired.)

3.7.2. Copy Utility

```
PROCEDURE Copy (    source:         ADDRESS;
                    destination:    ADDRESS;
                    size:           CARDINAL        );
```

source	is the address of the data to be copied from.
destination	is the address of the area to be copied to.
size	is the length (in TSIZE units) to be copied.

The Copy procedure simply copies a block of memory of specified size from a source location to a destination. Note: The copy operation is performed a WORD at a time (size is rounded up to the next whole WORD). Copy is only provided as a convenience, since it would be a simple matter for the user to provide his own copy routine.

PART II

Simon II Advanced Features

Part II of this manual discusses advanced features of Simon concerning ports, messages, and the capability of extending Simon.

* Chapter 4 discusses advanced message features, namely: message preprocessing using input port server procedures, message attributes, message transmission using keys, broadcast messages (GSMs), and editing of messages in the event queue.

* Chapter 5 explains how Simon can be extended using software support layers. A discussion of support layers along with object contexts is presented, followed by an example.

4. Advanced Message Features

4.1. Input Servers

An input server is a procedure associated with an input port which pre-processes each message which arrives at that port. The same input server may be used to pre-process messages for more than one port, however each input port is assigned to exactly one input server. Before a message arrives, an object procedure will have called Wait. Wait causes Simon to suspend execution of the object's process until a message arrives, at which time Wait resumes execution. Wait then calls the input server procedure of the port at which the message arrived, passing a pointer to the message, its data record, and the address of the port's context (discussed later) to the input server. When the input server finishes, it returns a pointer to the message and its data record to Wait. (It can instead return a pointer to a different message, or Nil, if desired.) If the pointer to the data record is Nil, Wait will wait for another message to arrive. Otherwise, Wait will return to the object procedure with the two pointers.

Input servers can be used for a variety of purposes. They are quite useful for support layers (discussed later), allowing specialized input port behavior to be provided. For example, queued input ports can be supported by an input server which enqueues each message as it arrives. Input servers can also provide support for an "event based" approach to simulation. Using this approach, messages are events and input servers become event handlers. The object procedure is not needed for processing and can simply return after initialization. All messages (events) are then handled completely by the input servers (event handlers). In contrast, if a "process based" simulation methodology is used, all messages would be processed by the object procedure and the input servers would do little more than queueing functions. The user can use either the object procedure or input servers or both, to accomplish his objectives.

4.1.1. Input Server Definition

```
PROCEDURE <input server> (
                    VAR mesg:       pMsg;
                    VAR data:       ADDRESS;
                    contxt:         ADDRESS         );
```

mesg	is a pointer to the message.
data	is the address of the data portion of the message.
contxt	is the address of the port context record.

As was mentioned above, the address of the port's context record is passed to the input server. This context is a user defined record which essentially provides the input server with the capability of having static local variables. These variables can provide the state information necessary to process messages. For example, an application might require an object with several input ports, where input to each port consisted of a sequence of messages. Static variables would be needed for each port to keep tract of its progression in the sequence. These variables would be defined as fields in a context record. During initialization an individual record would be allocated and initialized for each port. This record would then be assigned to the input port when the port was created (see MakIPort and MakIRPort). Note that it is also possible for more than one port to share a context record.

4.2. Attributes

As was mentioned earlier, messages may have an arbitrary number of attributes associated with them. Attributes may be attached to any accessible existing message. Each attribute consists of a user defined data record and an identifying type so that it can be distinguished from other attributes. An attribute can be thought of as a transparent sub-message. This facility is useful for attaching additional information (such as statistics gathering information) to a message without disturbing the format of the message's data record, which might cause incompatibilities with other existing objects.

In order to use attributes, the Modula 2 data type for the attribute data structure is needed. Simon exports the type "Attrib", which represents an attribute, as well as the type "pAttrib", which represents a pointer to an attribute. These are used in much the

same way as Msg and pMsg are used for messages.

4.2.1. Attribute Type Creation

Before an attribute is created, a Simon attribute type must be defined. Even though the Simon defined type, "Generic", can be used, a unique type is preferable. DefAttrTyp is a Simon procedure which generates a unique attribute type each time it is called. When the system is initialized, DefAttrTyp can be used to create unique attribute types, which can then be distributed to objects as initialization parameters when the objects are instantiated.

PROCEDURE DefAttrTyp (name: ARRAY of CHAR) : CARDINAL;

name	is the external (string) name of the attribute type, which does not have to be unique.
DefAttrTyp	returns a cardinal attribute type.

4.2.2. Attribute Creation

When an attribute is created, a type, data record length, and a message are specified. The attribute type identifies the user defined type and format of the information in the data record. The data record length (which can be greater than or equal to zero) determines the length of the data portion of the attribute. The format of the data record is left to the user. Data can be placed in the data record using the same method as was used for messages (see MakMsg in the tutorial). At the time the attribute is created, it is attached to the specified message. It remains with the message until the attribute is changed or deleted, or the message is deleted. For this reason it is important that ChgMsg is used to make major modifications to messages which are to be forwarded, instead of deleting the message and creating a new one. Otherwise, any attached attributes would be lost.

```
PROCEDURE MakAttr (   type:       CARDINAL;
                      datasize:   CARDINAL;
                      mesg:       pMsg          ) : pAttrib;
```

type	is the attribute type (originally returned from DefAttrTyp).
datasize	is the size (in TSIZE units) of the data portion of the attribute.
mesg	is a pointer to the message to which the attribute is to be attached.
MakAttr	returns a pointer to the attribute.

4.2.3. Attribute Identification

After receiving a message, an object can examine any of the attributes attached to the message. To access a particular attribute, FindAttr is first called. In calling FindAttr the user specifies the attribute type to be sought and the message to which the attribute is attached. If FindAttr is able to locate the specified attribute, it will return a pointer to the attribute, otherwise it will return **Nil** (no error message is generated). If more than one attached attribute is of the type specified, FindAttr will return a pointer to the attribute which was last attached (LIFO order). In this case FindAttr will not be able to find any other attributes of the same type until the present attribute (last one attached of that type) has been deleted. The data record of the attribute can then be examined (or modified) in the same way messages are read (or modified).

```
PROCEDURE FindAttr (  type:       CARDINAL;
                      mesg:       pMsg          ) : pAttrib;
```

type	is the type of the attribute being sought.
mesg	is a pointer to the message to which the attribute is attached.
FindAttr	returns a pointer to the attribute, or Nil if not found.

4.2.4. Attribute Modification

The fixed-size data record of the attribute can be directly modified by the user. However, if major modification of the data record is required (changing its size), then ChgAttr should be used. ChgAttr replaces the old attribute with a newly created attribute of specified size and content, and returns a pointer to the new attribute. (Note: Since the old attribute is not deleted until the new attribute has been created, the new data record could be specified to contain data from the old attribute.)

```
PROCEDURE ChgAttr (   oldattr:    pAttrib;
                      newdata:    ADDRESS;
                      newsize:    CARDINAL      ) : pAttrib;
```

oldattr	is a pointer to the attribute to be changed.
newdata	is the address of the new data record.
newsize	is the size (in TSIZE units) of the new data record.
ChgAttr	returns a pointer to a new (changed) attribute.

4.2.5. Attribute Deletion

Attributes remain attached to a message until the attribute or the message is deleted. DelAttr deletes the specified attribute by first removing it from the message and then freeing the memory space used by the attribute.

```
PROCEDURE DelAttr (   attr:    pAttrib      );
```

attr	is a pointer to the attribute to be deleted.

4.2.6. Attribute Duplication

When a message is forwarded, the attached attributes are automatically forwarded with it. However, there may be instances where it would be desirable to forward a copy of the attributes with a different message (or possibly several messages). DupAttrs provides this capability. It creates a duplicate copy of all attributes attached to the original message (srcms), and then attaches these duplicates to the specified destination message (dstms), ahead of (LIFO order) any attributes previously attached.

```
PROCEDURE DupAttrs (  srcms:    pMsg;
                      dstms:    pMsg        );
```

srcms — is a pointer to a message from which attributes will be copied.

dstms — is a pointer to a message to which the duplicate attributes will be appended.

4.3. Keys

In addition to the normal message distribution mechanism, Simon also supports keyed message transmission. Keyed messages and keyed input ports provide enhanced efficiency for selective message distribution to input ports. This mechanism allows an object procedure to specify a key along with the message being sent. When the message is distributed to connected input ports, Simon first compares the message key with each input port key. The port will receive the message, only if the key matches. Simon implements the key as a pair of values: a key, and a mask. The mask is used to mask-out portions of the other key during the comparison. Hence, the actual test for a match is:

[Message Key AND Port Mask] = [Port Key AND Message Mask].

This implementation provides the user with the flexibility to break the Key field into sub-fields. By appropriately setting the mask, only selected sub-fields would be tested in the comparison. This approach can also provide limited range matching. For example, a memory object may service requests (messages) arriving from a bus for a range of memory addresses. The addresses comprising that range are typically within a power of two interval, such that the high order portion of all addresses within that range is identical. An input port key and mask can then be set up which would test only that high order portion of the address (key) for each service request (message). In this way the memory object would receive only the messages intended for that memory address range.

Keyed and unkeyed messages and ports can be used together. When a message or port is unkeyed, it is equivalent to a wild card key (key = 0, mask = 0). This means that each unkeyed message is delivered to every connected keyed and unkeyed port, and conversely, each unkeyed port receives every keyed and unkeyed message sent.

4.3.1. Setting Port Keys

SetKey allows an input port's key (and mask) to be set or changed. It can be applied to shell input ports as well as regular and read-only input ports. All newly created ports are unkeyed (key = 0, mask = 0). If after a port key has been set, it is desired to set the port to unkeyed, simply call SetKey with key = 0 and mask = 0.

```
PROCEDURE SetKey (  prt:    pPort;
                    key:    WORD;
                    mask:   WORD    );
```

prt	is a pointer to the input port whose key is being set.
key	is the port key.
mask	is the mask which is applied to the message key.

4.3.2. Keyed Message Transmission

KeyedSend and KeyedSendMsg allow keyed messages to be sent. They are identical to Send and SendMsg except for the addition of the key and mask arguments.

```
PROCEDURE KeyedSend ( mesg:     pMsg;
                      prt:      pPort;
                      timeincr: CARDINAL;
                      key:      WORD;
                      mask:     WORD    );
```

mesg	is a pointer to the message to be sent.
prt	is a pointer to an output port to which the message is to be sent.
timeincr	is the time interval (from now) at which the message is to arrive.
key	is the message key.
mask	is the mask which is applied to the port key.

```
PROCEDURE KeyedSendMsg (
                    dat:        ADDRESS;
                    len:        CARDINAL;
                    prt:        pPort;
                    timeincr:   CARDINAL;
                    key:        WORD;
                    mask:       WORD        );
```

dat	is a pointer to the data record to be sent.
len	is the size (in TSIZE units) of the data record.
prt	is a pointer to an output port to which the message is to be sent.
timeincr	is the time interval (from now) at which the message is to arrive.
key	is the message key.
mask	is the mask which is applied to the port key.

4.4. General System Messages

Simon supports a broadcast message mechanism called GSM (General System Message). This mechanism allows messages to be sent and distributed without using output ports or connections, in a way analogous to radio broadcasts. The sender simply sends a GSM, and the receiver(s) (object(s) with a GSM port) will automatically receive the GSM. GSM's are identical to regular messages, except that:

* GSM's cannot be keyed.

* GSM's cannot have attributes attached to them.

* GSM input ports are always read-only (see MakIRPort).

Simon makes no distinction between regular messages and GSM's in its internal message (event) queue. The only way Simon can tell them apart is that regular messages have a message type of "Generic" (Simon defined type), and GSM's have a user defined type (other than "Generic"). However, distribution of a GSM to receiving ports is different from the distribution of regular messages. Whenever an object creates a GSM port, it must specify a GSM type. When a GSM is distributed, it is delivered to every GSM port in the system which has the same type as the GSM, regardless of where the port is located in the system configuration.

GSM's are useful in applications requiring a global broadcast of information. For instance, GSM's could be used to broadcast the ticks of a master clock to an array of processing elements. However, GSM's are typically used for system overhead functions, such as start-up messages, shut-down messages, or statistics gathering. In particular, the problem of starting up the system simulation is conveniently addressed by the GSM mechanism.

After the main procedure has initialized the configuration and before the simulation begins, at least one message will have to have been sent. Otherwise, when Simulate is called it will return immediately, since the message (event) queue will be empty. An initial message can be sent in one of two ways.

1. An object procedure can send a message during its initialization phase. This is valid, even though its ports are probably not connected to anything, because messages do not traverse the connection graph until they are to be received, which is after the system configuration has been completed.

2. The main procedure can send a "start-up" GSM before calling Simulate. All "up-stream" objects which feed the system can have a GSM port to receive the "start-up" GSM.

Although either method will work, the GSM mechanism provides greater flexibility. For example, suppose that a simulation "run" is to be performed several times with the same configuration, but with different input data (stored in separate files). The main procedure could contain a loop containing calls to ResetClock, SendGSM, and Simulate. This would successfully invoke each of the multiple runs, however a mechanism would be needed to pass the name of the data file to be used (and any other necessary parameters) to the "up-stream" objects. This could be done simply with the GSM. Since a GSM is a message, the file name (and any other parameters) could be included in the data record of the GSM. In this way new parameters may be easily passed to the appropriate objects, at the start of each run in a multiple run simulation.

4.4.1. GSM Type Creation

Unique GSM types can be obtained by calling DefGSMTyp. DefGSMTyp operates in a manner similar to DefAttrTyp, in that it generates a unique GSM type each time it is called. When the system is initialized, DefGSMTyp can be used to create unique GSM types, which can then be distributed to objects as initialization parameters when the objects are instantiated.

```
PROCEDURE DefGSMTyp (
                        name:          ARRAY of CHAR      ) : CARDINAL;
```

name	is the external (string) name of the General System Message type, which does not have to be unique.
DefGSMTyp	returns a cardinal GSM type.

4.4.2. GSM Port Creation

In order for an object to receive GSM's of a particular type, a GSM port of that same type must be created. An object may have any number of GSM ports of any type. If an object has two GSM ports of the same type, then that object will receive any GSM of that type, twice. To create a GSM port, MakGSMPort is called, specifying the GSM type, an input server and context, and a port tag value. The input server and input server

context function identically to input servers and input server contexts for regular input ports (see the section on Input Servers). The tag value is used to set the tag field of the GSM port. It functions similarly to regular input port tags (see the section on Input Port Tags). It should be remembered that GSM ports are always read-only input ports (see MakIRPort).

```
PROCEDURE MakGSMPort (
                        type:       INTEGER;
                        server:     SrvrPrc;
                        contxt:     ADDRESS;
                        tag:        INTEGER        );
```

type	is the type of GSMs to be received by this port.
server	is an input server procedure.
contxt	is the address of a context record for the input server.
tag	is the user defined value of the GSM port tag field.

4.4.3. GSM Port Deletion

DelGSMPort will delete a GSM port of the specified type, on the currently executing object. If no such port is found, an error message will be generated. If more than one such port exists, then the last one created will be deleted.

```
PROCEDURE DelGSMPort( type:       CARDINAL       );
```

type	is the type of GSM port to be deleted.

4.4.4. GSM Transmission

Sending a GSM differs slightly from sending a regular message. Simon automatically creates GSM's when they are sent; they cannot be created separately. Since GSM's are read-only, Simon also automatically deletes them, too. To create and send a GSM, SendGSM is called. It is identical to SendMsg except that a GSM type is specified instead of an output port.

```
PROCEDURE SendGSM (  data:       ADDRESS;
                     datasize:   CARDINAL;
                     type:       CARDINAL;
                     timeincr:   CARDINAL      );
```

data	is the address of the data record to be copied to the GSM before sending.
datasize	is the size (in TSIZE units) of the data record.
type	is the GSM type.
timeincr	is the time interval (from now) at which the message is to arrive.

4.5. Message [Event] Queue Editing

When a message is sent, it is placed in Simon's internal message (event) queue. It remains there until the time it is to be distributed to destination ports. While the message is in the queue, it typically should not be accessed. However, there are situations where it would be desirable to be able to change or delete messages in the message (event) queue. For example, an object might model a device which must process asynchronous, external interrupts. The behavior of the device is modeled by the object as a sequence of "computations". The object execution consists of constantly performing a "computation" and then sending a message to itself to appropriately advance the simulation clock. The object has no idea of when an interrupt (represented by an "interrupt" message) might occur (be received). If it occurs in the middle of a "computation", the object will not realize it until: it has finished the "computation", sent the message to itself, and then received the "interrupt" message instead of the message to itself (the message to itself will still be in the message (event) queue). At this point, the object may need to undo or modify part of its last "computation", including any messages that it may have sent. To allow an object to modify or delete any outstanding messages that it has sent, Simon provides several procedures.

4.5.1. Delaying All Messages

Sometimes the only thing that needs to be changed is the arrival times of the messages that were sent (to account for the time delay imposed by the processing of the interrupt). This is accomplished simply by calling DelayMsgsPrt or DelayMsgsObj. DelayMsgsPrt searches for all messages in the queue that were sent from the specified port, while DelayMsgsObj searches for all messages in the queue that were sent from any port on the specified object. In either case, the procedure delays (reschedules) each message that it found by the specified delay time (timeincr).

```
PROCEDURE DelayMsgsPrt (
                    prt:        pPort;
                    timeincr:   CARDINAL        );
```

prt is a pointer to the source port of the messages to be delayed.

timeincr is the amount of time the messages are to be delayed.

```
PROCEDURE DelayMsgsObj (
                    obj:        pObject;
                    timeincr:   CARDINAL        );
```

obj is a pointer to the source object of the messages to be delayed.

timeincr is the amount of time the messages are to be delayed.

4.5.2. Identifying Messages

If more substantial or varied changes are required, the object must change each message, one at a time. Before the changes can be made, each of the targeted messages must be found. To accomplish the search, Simon provides FindMsgsPrt and FindMsgsObj. FindMsgsPrt searches for all messages in the queue that were sent from the specified port, while FindMsgsObj searches for all messages in the queue that were sent from any port on the specified object. In either case, the procedure creates a "list" of all the messages that it found. Note: Any messages found remain in the event queue, unless explicitly deleted by the user. At this point the object can repeatedly call FindNext

to obtain a pointer to the next message that it needs to change. FindNext returns a pointer to each subsequent message on the "list" until the "list" is empty, in which case it returns Nil. It should be noted that the FindMsgsPrt and the FindMsgsObj will always create a new "list"; any previous "list" is erased.

```
PROCEDURE FindMsgsPrt (
                        prt:            pPort           );
```

prt	is a pointer to the source port of the messages to be sought.

```
PROCEDURE FindMsgsObj (
                        obj:            pObject         );
```

obj	is a pointer to the source object of the messages to be sought.

```
PROCEDURE FindNext (                                    );
```

FindNext	returns a pointer to the next message sought, or Nil if no more.

4.5.3. Delaying, Modifying, and Deleting a Message

If an individual message must be delayed (rescheduled), then Resched can be called. The user simply specifies the message (which must be in the queue) and the time delay. Note: No message can be rescheduled to be delivered at more than twice the "TicLimit" from now. This restriction applies to all of Simon's rescheduling procedures. Since TicLimit (an implementation dependent constant) is typically a large value (10^9 on the VAX), this restriction usually poses no problems.

```
PROCEDURE Resched (     mesg:           pMsg;
                        timeincr:       CARDINAL        );
```

mesg	is a pointer to the message to be delayed.
timeincr	is the amount of time the message is to be delayed.

Other changes can be made to individual messages in the queue. Essentially, all changes which can normally be made to messages, can also be made to messages in the

message (event) queue. The contents of a message (the fixed-length data record) can be directly modified by the user. If major modification is required which would change the size of the data record, ChgMsg can be used. (ChgMsg will place the new message where the old message was in the queue.) If it is necessary to delete the message, DelMsg can be used. (DelMsg updates the queue after removing the message.)

With these capabilities, an object procedure should be able to undo or modify any processing which it had done. Obviously, it won't be able to undo any messages which were sent and have already been received. But it must be remembered that any messages which have been received before the interrupt occurred, were not affected by the interrupt, anyway. The only messages which could possibly be affected by an interrupt are messages in the message (event) queue.

5. Extending Simon

5.1. Software Support Layers

The Simon kernel provides a rich collection of primitives as an interface to the user. However, a particular user may need a higher level interface, oriented closer to an application. This can be provided by using a layer of software between the user and the Simon kernel.

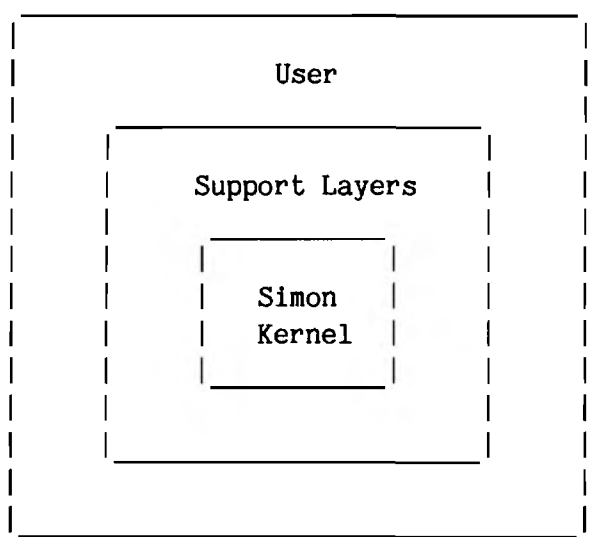

Figure 5-1: Extending Simon with Support Layers

The user can make calls to a support layer, requesting a "high level" service. The support layer then breaks the request down to "low level" services which it can provide directly and/or by calling the Simon kernel primitives. Each support layer is typically implemented as a separate software module so that it can be linked into the simulation program when needed. It is not necessary for a support layer to provide the entire user interface. Different layers providing various services, can be combined to obtain the desired user interface. Additionally, the user can always make direct calls to the Simon kernel for any other services.

Suppose that a user needed a queued input port mechanism for an application. This can be accomplished using a support layer. The support layer could provide a procedure, "MakQIPort" for creating a queued input port. MakQIPort would simply create a

regular input port using MakIPort (Simon kernel), but specify an input server provided by the support layer. This input server would then be responsible for enqueuing each message arriving at the port. The support layer would also have to provide procedures to allow the user to extract messages from the queue, check the queue status, etc. This support layer would thus provide the needed queued input port mechanism. Notice however, that this support layer does not preclude the direct use of other types of ports or services provided by the Simon kernel, nor services provided by other support layers.

5.2. Useful Global Variables and Procedures

In addition to the information passed in the call to a support layer procedure, Simon provides several global variables. CurObj is a pointer to the currently executing object (the calling object). CurMsg is a pointer to the most recent message received by the object, and CurLen is the length of the data portion of the message. CurInSrvr is the input server for the most recent message, and CurSrvrCt is a pointer to the receiving port's context. These variables can provide useful information to support layer procedures, as well as object procedures. Additionally, MsgHdrSiz and AttrHdrSiz provide the size (in TSIZE units) of the header portion of messages and attributes, respectively. These sizes can sometimes be helpful in message or attribute manipulations.

Simon provides a name server which can be used by support layer procedures.

PROCEDURE EnterName (name: ARRAY OF CHAR): pXName;

name	is the external (string) name to be defined.
EnterName	returns a pointer to the name definition.

EnterName will search Simon's external name table for the specified name, and if not found, creates a new entry. It then returns a pointer to the entry (of type pXName). Pointers to Simon external names can be compared directly to determine if names are identical. A name server can be useful for nameing new abstract entities used by a support layer.

5.3. Object Contexts and Support Layers

As a support layer provides services to an object procedure, it is very likely that the support layer will need some static local variables to maintain state information concerning the object. However, the support layer may be providing services to numerous objects, each with its own state information. To manage all of this state information, Simon provides object contexts. Just as an input port context provides static local variables concerning the port to an input server (see section on Input Servers), an object context provides static local variables concerning the object to a support layer. Although a port can have only one port context, an object may have any number of object contexts associated with it at a time. This means that an object is not restricted in the number of support layers it uses.

As with port contexts, the static local variables for an object context are accessed as fields within a context record. The address of the port context record is explicitly passed to an input server, but this approach cannot be used in the case of object context records. Instead, the support layer provides a global[2] pointer which points to the context record. All of the support layer procedures must access the context record though this global pointer. Obviously, this global pointer must be updated each time a different object resumes execution. Simon will perform this updating automatically, if it has been notified of the object context. The notification is performed by calling MakContext or MakContextR. MakContextR is identical to MakContext except that it sets up a context for a specified remote object.

[2]The pointer must be global within the support layer module, so that all support layer procedures can access it. However, it does not have to be visible outside of the module.

5.3.1. Object Context Creation

```
PROCEDURE MakContext (
                        refpntr:        ADDRESS         );
```

refpntr — is the address of a pointer to the context record.

```
PROCEDURE MakContextR (
                        refpntr:        ADDRESS
                        ob:             pObject         );
```

refpntr — is the address of a pointer to the context record.

ob — is a pointer to the remote object.

To reiterate, an object context is created by following these steps:

1. A context record is created (allocated).
2. The address of the context record is placed in the support layer's global pointer.
3. The **address** of the global pointer is passed to MakContext. (The global pointer must contain the context record address when the call is made.)

From this point on, Simon will maintain the global pointer as follows:

* When an object resumes execution, Simon will restore the address of the context record to the pointer.
* When an object suspends execution, Simon will save the contents of the pointer (in case it changed during execution).
* If an object resumes execution which does not have a context record associated with it, Simon will set the pointer to Nil.

Typically, a support layer provides an initialization procedure to create an object context and initialize it. From then on, whenever the object procedure calls a support layer procedure, the proper object context record will be accessed. However, if the initialization procedure has not been called, the support layer's global pointer will be Nil. Each support layer procedure should first check to see if the global pointer is Nil, and if

so, should call the initialization procedure before proceeding. In this way support layer initialization can be made transparent to the object procedure.

5.4. Support Layer Example

Simon does not provide a "Send" procedure to schedule messages more than "TicLimit" units of time in the future. The following support layer provides this capability. It exports SendFar(Message, Port, Era, Time), which sends messages far into the future. SendFar has the same arguments as Send, except that the time increment is represented by Era and Time (actual time increment = Era * TicLimit + Time).

```
IMPLEMENTATION MODULE FarFuture;

FROM SYSTEM IMPORT TSIZE, ADR, ADDRESS;
FROM Simon IMPORT pPort, pMsg, pAttrib, CurObj, TicLimit, Generic,
    MakContext, MakCleanUp, MakOPort, MakIPort, Connect, Send,
    DefAttrTyp, MakAttr, FindAttr, DelAttr, GetMem, FreeMem;

TYPE
    DelayPort = RECORD
        DelayOut: pPort;
    END;

    pDelayPort = POINTER TO DelayPort;

    DelayAttr = RECORD
        prt: pPort;
        era: CARDINAL;
        tim: CARDINAL;
    END;

    pDelayAttr = POINTER TO DelayAttr;

VAR
    DPrt: pDelayPort;
```

The type, DelayPort, describes the object context record. The type, DelayAttr, describes the data portion of the attribute which is attached to messages to keep tract of the time delay. DPrt is the global pointer to the object context for the support layer.

```
PROCEDURE Init();

BEGIN
    GetMem(DPrt, TSIZE(DelayPort));
    MakContext(ADR(DPrt));
    WITH DPrt^ DO
        DelayOut:=MakOPort("DelayOut");
        Connect(DelayOut, MakIPort("DelayIn", ReSend, NIL));
    END;
    MakCleanUp(Dispose, DPrt);
    RETURN;
END Init;
```

Init allocates a context record, sets the global pointer, and then creates an object context. Then, a time delay mechanism is created and a pointer to the output port is saved in the object context. Finally, a clean-up procedure is set up for the object context.

```
PROCEDURE SendFar(ms: pMsg; pt: pPort; erainer, timincr: CARDINAL);

VAR
    at:  pAttrib;
    dat: pDelayAttr;

BEGIN
    IF DPrt = NIL THEN
        Init;
    END;
    WITH DPrt^ DO
        IF erainer = 0 THEN
            Send(ms, pt, timincr);
            RETURN;
        END;
        at:=MakAttr(Generic, TSIZE(DelayAttr), ms);
        dat:=ADR(at^.Data);
        WITH dat^ DO
            prt:=pt;
            era:=erainer;
            tim:=timincr;
        END;
        Send(ms, DelayOut, TicLimit);
        RETURN;
    END;
END SendFar;
```

SendFar first checks if the support layer has been initialized. If not, it calls Init. Then the global pointer is dereferenced to allow direct access to fields within the object context record. If the time increment is less than TicLimit, the message is sent directly; otherwise it must be delayed. To delay the message, an attribute (of type "Generic") is attached, which holds the sending port and the remaining time delay (both high and low order portions). The message is then sent out of the delay port (obtained from the object context), to be received TicLimit units of time from now.

```
PROCEDURE ReSend(VAR ms: pMsg; VAR d: ADDRESS; ct: ADDRESS);

VAR
    at:  pAttrib;
    dat: pDelayAttr;
    pt:  pPort;
    t:   CARDINAL;

BEGIN
    WITH DPrt^ DO
        at:=FindAttr(Generic, ms);
        dat:=ADR(at^.Data);
        WITH dat^ DO
            era:=era - 1;
            IF era # 0 THEN
                Send(ms, DelayOut, TicLimit);
            ELSE
                pt:=prt;
                t:=tim;
                DelAttr(at);
                Send(ms, pt, t);
            END;
        END;
        ms:=NIL;
        d:=NIL;
        RETURN;
    END;
END ReSend;
```

When the message arrives at the delay port, it is passed to the input server, ReSend. Here again, the global pointer is dereferenced. ReSend then finds the attached attribute (of type "Generic"). Note: ReSend finds the correct attribute, even though a previous "Generic" attribute may have been attached, since attributes are found in LIFO order (see FindAttr). The "era" portion of the remaining time is then decremented (since TicLimit units of time have elapsed). If the "era" portion is not zero, the message is sent out of the delay port, again scheduled at TicLimit from now. If the "era" portion is zero, the message can be sent to its real destination. The sending port and remaining time (low order portion) are obtained from the attribute before the attribute is deleted, and then the message is sent to its intended destination. Before returning, ReSend must return a data record pointer of Nil (to Wait). This is so that Wait will wait for another message, and ReSend will then appear transparent to the object.

```
PROCEDURE Dispose(drec: ADDRESS);

BEGIN
    FreeMem(drec, TSIZE(DelayPort));
    RETURN;
END Dispose;
```

The clean-up procedure, Dispose (created by MakCleanUp, see Init), is called by Simon if the object is deleted. Dispose then deallocates the object context. Note that Dispose cannot use the global pointer, since the corresponding object is not currently executing. Instead Simon passes the address of the object context to Dispose (Simon obtained this address from the MakCleanUp call). Obviously, the object context in this case is hardly worth the "clean-up" effort. However, this example serves to show how "clean-up" is performed, in case it is needed.

```
BEGIN
    DPrt:=NIL;
END FarFuture.
```

The above module initialization code is executed only once, before the main procedure begins execution. Here the global pointer is initialized to Nil.

Appendix I
Simon Procedures Reference

The following is a listing of the Simon definition module. It provides a reference of all the procedures, variables, constants, and data types exported by Simon. Although the number of items exported is large, a typical user will need to import only a fraction of them. The other items are exported for use by the more advance user and for integration with other specialized software tools (such as an on-line debugger).

```
(*****************************************************************************
 *                                                                           *
 *      Simon Simulation Kernel                                              *
 *                                                                           *
 *      Computer Science Department, University of Utah.                     *
 *                                                                           *
 *      Last Revision:  18 May 1987                                          *
 *                                                                           *
 *****************************************************************************)

DEFINITION MODULE Simon;

    FROM SYSTEM IMPORT PROCESS, WORD, ADDRESS, MAXINT;


(*      Data Structure and Pointer Types                                     *)

    EXPORT QUALIFIED Msg, Attrib, AttrDef, Object, Context,
        PortType, Port, Link, XName, GSMPort, pMsg, pAttrib, pAttrDef,
        pObject, pContext, pPort, pLink, pXName, pGSMPort, pGSMDef;

        (*      Note:  The user should define the constant:     *)
        (*               System = Nil;                          *)
        (*      The Owner of a top level object is "System".    *)


(*      Message Type Codes and Miscellaneous Constants and Types         *)

    EXPORT QUALIFIED SHshSiz, GHshSiz, Generic, TicLimit,
        ObjPrc, SrvrPrc, ClnUpPrc;
```

```
(*      Procedures to Manipulate Objects                                *)

    EXPORT QUALIFIED MakObject, DelObject, MovObject, MovObjectP,
        MakContext, MakContextR, MakCleanUp, MakCleanUpR;


(*      Procedures to Manipulate Ports                                  *)

    EXPORT QUALIFIED MakSOPort, MakOPort, MakSIPort, MakIPort, MakIRPort,
        DelPort, FindPort, FindPortS, FindIPort, FindIPortS, FindOPort,
        FindOPortS, Connect, Disconnect, SetKey, SetPortTag, MakGSMPort,
        DelGSMPort;


(*      Procedures to Manipulate Messages                               *)

    EXPORT QUALIFIED MakMsg, MakMsgD, ChgMsg, DelMsg, DupMsg, FindMsgsPrt,
        FindMsgsObj, FindNext, DelayMsgsPrt, DelayMsgsObj, Resched, Send,
        SendMsg, KeyedSend, KeyedSendMsg, DefGSMTyp, SendGSM, NilSrvr;


(*      Procedures to Manipulate Attributes                             *)

    EXPORT QUALIFIED DefAttrTyp, MakAttr, ChgAttr, DelAttr, DupAttrs,
        FindAttr;


(*      Miscellaneous Procedures                                        *)

    EXPORT QUALIFIED StopAt, Simulate, ResetClock, Reset,
        GetMem, FreeMem, Copy, EnterName, Wait;


(*      Global Variables                                                *)

    EXPORT QUALIFIED CurTime, CurEra, StopTime, StopEra, CurMsg, CurObj,
        CurInSrvr, CurSrvrCt, CurTag, CurLen, CurPrs, RetPrs, STrace,
        MTrace, Objects, GSMPorts, LastAttrTyp, LastGSMTyp, AttrDefs,
        GSMDefs, XNames, Q, XeqLst, MsgHdrSiz, AttrHdrSiz;
```

```
CONST

    SHshSiz     = 100;              (* Size of Hash Table for Ext Symbols *)
    GHshSiz     = 100;              (* Size of Hash Table for GSM Ports   *)
    Generic     = 0;                (* General Purpose Message Type       *)
    TicLimit    = 1000000000;       (* Clock Low Order Overflow Limit     *)
    TicLimitX2  = 2 * TicLimit;


TYPE

    pMsg = POINTER TO Msg;
    pAttrib = POINTER TO Attrib;
    pObject = POINTER TO Object;
    pContext = POINTER TO Context;
    pCleanUp = POINTER TO CleanUp;
    pPort = POINTER TO Port;
    pLink = POINTER TO Link;
    pXName = POINTER TO XName;
    pGSMPort = POINTER TO GSMPort;
    pAttrDef = POINTER TO AttrDef;
    pMsgDef = POINTER TO MsgDef;
    pGSMDef = POINTER TO GSMDef;

    ObjPrc = PROCEDURE(ADDRESS);
    SrvrPrc = PROCEDURE(VAR pMsg, VAR ADDRESS, ADDRESS);
    ClnUpPrc = PROCEDURE(ADDRESS);
```

```
(*************************************************************************
 *                                                                       *
 *      The fields of message (Msg) and attribute (Attrib) are user      *
 *      accessible.  However, fields where the description starts        *
 *      with "!", signify that the field is typically for Simon use.     *
 *                                                                       *
 *************************************************************************)

   Msg = RECORD
       Prev:   pMsg;           (* ! Event Queue Linkage                 *)
       NextL:  pMsg;           (* ! Event Queue Linkage                 *)
       NextR:  pMsg;           (* ! Event Queue Linkage                 *)
       QLevel: CARDINAL;       (* ! Queue Level (0 if not Queued)       *)
       Owner:  pObject;        (* ! Owner of this Message Copy          *)
       Attr:   pAttrib;        (* ! Attribute List Linkage              *)
       List:   pMsg;           (* ! Message Edit List (FindMsgsX)       *)
       Time:   CARDINAL;       (* Time Stamp of Message Arrival         *)
       Typ:    CARDINAL;       (* Message or GSM Type                   *)
       DstPrt: pPort;          (* Pointer to Destination Port           *)
       SrcPrt: pPort;          (* Pointer to Originating (Out) Port     *)
       SrcObj: pObject;        (* Pointer to Originating Object         *)
       Key:    WORD;           (* Key / Mask for KeyedSend              *)
       Mask:   WORD;           (* Key / Mask for KeyedSend              *)
       Len:    CARDINAL;       (* Length of Message (Data Record)       *)
       Data:   ARRAY [0..MAXINT DIV 16] OF WORD;
                               (* Message (Data)                        *)
   END;

   Attrib = RECORD
       Prev:   pAttrib;        (* ! Attribute List Linkage              *)
       Next:   pAttrib;        (* ! Attribute List Linkage              *)
       Mes:    pMsg;           (* ! Pointer to Message                  *)
       Typ:    CARDINAL;       (* Attribute Type                        *)
       Len:    CARDINAL;       (* Length of Attribute (Data Record)     *)
       Data:   ARRAY [0..MAXINT DIV 16] OF WORD;
                               (* Attribute (Data)                      *)
   END;
```

```
(*************************************************************************
 *                                                                       *
 *      The following structures are typically not accessed by the       *
 *      user, but are for Simon use.                                     *
 *                                                                       *
 *************************************************************************)

   Object = RECORD
       Name:    pXName;          (* Pointer to Symbol Table              *)
       Owner:   pObject;         (* Pointer to Surrounding Object        *)
       Prev:    pObject;         (* Sibling Object Linkage               *)
       Next:    pObject;         (* Sibling Object Linkage               *)
       SubObj:  pObject;         (* Sub-Object Linkage                   *)
       BkUp:    pObject;         (* Used in Structure Traversal          *)
       Ports:   pPort;           (* Port List Linkage                    *)
       GSMPrts:CARDINAL;         (* Count of Object's GSM Ports          *)
       Prs:     PROCESS;         (* Process of Object                    *)
       WSpc:    ADDRESS;         (* Process Work Space                   *)
       WSiz:    CARDINAL;        (* Process Work Space Size (in bytes)   *)
       Parm:    ADDRESS;         (* Address of Object's Parameters       *)
       ParmSiz:CARDINAL;         (* Length of Parameters (in bytes)      *)
       Cntxt:   pContext;        (* Linked List of Contexts              *)
       ClnUp:   pCleanUp;        (* Context Clean Up Procedure           *)
   END;

   Context = RECORD
       Next:    pContext;        (* Context List Linkage                 *)
       ObjCntxt:
                ADDRESS;         (* Address of Object's Context Record   *)
       RefPntr:
                ADDRESS;         (* Address of the Context Reference
                                          Pointer Variable               *)
   END;

   CleanUp = RECORD
       Next:    pCleanUp;        (* Clean Up List Linkage                *)
       ClnUp:   ClnUpPrc;        (* Clean Up Procedure                   *)
       Param:   ADDRESS;         (* Parameter for Clean Up Procedure     *)
   END;
```

```
PortType = (SOut, SIn, Out, Inp, InRO);
                              (* S = Shell, RO = Read-Only          *)

Port = RECORD
    Name:   pXName;           (* Pointer to Symbol Table            *)
    Objct:  pObject;          (* Pointer to Object                  *)
    Next:   pPort;            (* Port List Linkage                  *)
    Typ:    PortType;         (* Port Type                          *)
    QdMsgs: CARDINAL;         (* Count of Port's Queued Messages    *)
    DLink:  pLink;            (* Pointer to a List of
                                        Destination Pointers        *)
    SLink:  pLink;            (* Pointer to a List of Source Pointers *)
    Tag:    INTEGER;          (* User defined Port identifier       *)
    Key:    WORD;             (* Key / Mask for SetKey              *)
    Mask:   WORD;             (* Key / Mask for SetKey              *)
    List:   pPort;            (* Destination Port List which is to
                                     Receive Current Message        *)
    Cnt:    CARDINAL;         (* Number to be Received (See "List") *)
    InSrvr: SrvrPrc;          (* Input Server Procedure for Port    *)
    SrvrCt: ADDRESS;          (* Context for InSrvr                 *)
END;

Link = RECORD
    Next:   pLink;            (* "Link" List Linkage                *)
    Prt:    pPort;            (* Pointer to Destination / Source    *)
    BkUp:   pLink;            (* Pointer Used in Structure Traversal *)
END;

XName = RECORD
    Next:   pXName;           (* Symbol Table Linkage               *)
    Hash:   INTEGER;          (* Sum of Characters in Name          *)
    Len:    CARDINAL;         (* Length of Name (not including Nul) *)
    Name:   ARRAY [0..MAXINT DIV 16] OF CHAR;
                              (* External (Symbolic) Name String    *)
END;

GSMPort = RECORD
    Objct:  pObject;          (* Pointer to Owner Object            *)
    Next:   pGSMPort;         (* GSM Port Linkage                   *)
    Typ:    CARDINAL;         (* GSM Type Received by This Port     *)
    Tag:    INTEGER;          (* User defined Port identifier       *)
    InSrvr: SrvrPrc;          (* Input Server Procedure for Port    *)
    SrvrCt: ADDRESS;          (* Context for InSrvr                 *)
END;
```

```
AttrDef = RECORD
    Name:   pXName;         (* Pointer to Symbol Table              *)
    Typ:    CARDINAL;       (* Attribute Type                       *)
    Next:   pAttrDef;       (* Attribute Type Definition
                                   List Linkage                     *)
END;

GSMDef = RECORD
    Name:   pXName;         (* Pointer to Symbol Table              *)
    Typ:    CARDINAL;       (* GSM Type                             *)
    Next:   pGSMDef;        (* GSM Type Definition List Linkage     *)
END;
```

```
VAR

(*************************************************************************
 *                                                                       *
 *      Global Variables                                                 *
 *                                                                       *
 *************************************************************************)

   STrace:      BOOLEAN;          (* Structure Tracing Flag              *)
   MTrace:      BOOLEAN;          (* Message Tracing Flag                *)
   StopEra:     CARDINAL;         (* Era to Stop Simulation              *)
   StopTime:    CARDINAL;         (* Time to Stop Simulation             *)
   CurTime:     CARDINAL;         (* Simulation Clock - Low Order        *)
   CurEra:      CARDINAL;         (* Simulation Clock - High Order       *)
   CurMsg:      pMsg;             (* Current Message for Process         *)
   CurLen:      CARDINAL;         (* Length of Data Record of Current Msg *)
   CurTag:      INTEGER;          (* Current User defined Port identifier *)
   CurInSrvr:   SrvrPrc;          (* Input server for current msg        *)
   CurSrvrCt:   ADDRESS;          (* Context for input server            *)
   CurPrs:      PROCESS;          (* Current Object's Process            *)
   RetPrs:      PROCESS;          (* Simon Process to Return to          *)
   Objects:     pObject;          (* Pointer to Object Structure         *)
   GSMPorts:    ARRAY [0..GHshSiz-1] OF pGSMPort;
                                  (* Hash Table of Pointers to GSM Ports  *)
   LastAttrTyp:CARDINAL;          (* Value of Last Attribute Type
                                        Assigned                        *)
   AttrDefs:    pAttrDef;         (* Pointer to Attribute Type
                                        Definitions List                *)
   LastGSMTyp: CARDINAL;          (* Value of Last GSM Type Assigned     *)
   GSMDefs:     pGSMDef;          (* Pointer to GSM Type Definitions List *)
   XNames:      ARRAY [0..SHshSiz-1] OF pXName;
                                  (* Hash Table of Pointers to External
                                        Name Symbol Table Lists         *)
   Q:           pMsg;             (* Pointer to the Event Queue          *)
   XeqLst:      pPort;            (* Pointer to a Distribution List for
                                        the Current Message             *)
   MsgHdrSiz:   CARDINAL;         (* Size of Msg Header (TSIZE units)    *)
   AttrHdrSiz: CARDINAL;          (* Size of Attr Header (TSIZE units)   *)
   CurObj:      pObject;          (* Pointer to the Currently Executing
                                        Object                          *)
```

```
(*
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                                                                         *
 *      Procedures to Manipulate Objects                                   *
 *                                                                         *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

 *)

(*************************************************************************
 *                                                                       *
 *      MakObject - Creates an object inside of the owner (outside       *
 *      if ObjOwner = System) and names it nam.  The prc process is      *
 *      created and then called with parameters (param) to               *
 *      initialize the new object.                                       *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakObject(nam: ARRAY OF CHAR; prc: ObjPrc; wkspcsiz: CARDINAL;
       param: ADDRESS; paramsiz: CARDINAL; ObjOwner: pObject): pObject;


(*************************************************************************
 *                                                                       *
 *      <object procedure> - Initialization / Process procedure of       *
 *      an object.  First time slice is for initialization.  Then        *
 *      either a return is executed (if no process is desired), or       *
 *      emulation of object behavior begins.                             *
 *                                                                       *
 *************************************************************************

PROCEDURE <object procedure>(param: ADDRESS);                            *)
```

```
(*****************************************************************************
 *                                                                           *
 *      DelObject - Delete the object, ob, along with its                    *
 *      sub-structure.                                                       *
 *                                                                           *
 *****************************************************************************)

PROCEDURE DelObject(ob: pObject);


(*****************************************************************************
 *                                                                           *
 *      MovObject - Remove the object, ob, along with its sub-structure      *
 *      From their current location and put them inside of NewOwner          *
 *      (outside if NewOwner = System).                                      *
 *                                                                           *
 *****************************************************************************)

PROCEDURE MovObject(ob, NewOwner: pObject);


(*****************************************************************************
 *                                                                           *
 *      MovObjectP - Same as MovObject except do not delete connections      *
 *      between object being moved and other objects.                        *
 *                                                                           *
 *****************************************************************************)

PROCEDURE MovObjectP(ob, NewOwner: pObject);
```

```
(*************************************************************************
 *                                                                       *
 *      MakContext - Associate the reference pointer (refptr) with       *
 *      the Current Object (CurObj).  Before calling MakContext,         *
 *      refptr^ must be initialized.  A clean up procedure should        *
 *      be provided for deleting the context when the associated         *
 *      object is deleted.  During simulation, the contents of the       *
 *      reference pointer are restored each time the object executes.    *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakContext(refptr: ADDRESS);


(*************************************************************************
 *                                                                       *
 *      MakContextR - Associate the reference pointer (refptr) with      *
 *      the specified object (ob).  Before calling MakContext,           *
 *      refptr^ must be initialized.  A clean up procedure should        *
 *      be provided for deleting the context when the associated         *
 *      object is deleted.  During simulation, the contents of the       *
 *      reference pointer are restored each time the object executes.    *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakContextR(refptr: ADDRESS; ob: pObject);


(*************************************************************************
 *                                                                       *
 *      MakCleanUp - Associate a procedure and its parameter with        *
 *      the Current Object (CurObj), so that it will be called if        *
 *      the object is to be deleted.  This provides a mechanism for      *
 *      cleaning up contexts and other house-keeping problems.           *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakCleanUp(prc: ClnUpPrc; parm: ADDRESS);
```

```
(*************************************************************************
 *                                                                       *
 *      MakCleanUpR - Associate a procedure and its parameter with       *
 *      the specified object (ob), so that it will be called if          *
 *      the object is to be deleted.  This provides a mechanism for      *
 *      cleaning up contexts and other house-keeping problems.           *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakCleanUpR(prc: ClnUpPrc; parm: ADDRESS; ob: pObject);



(*

 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                                                                         *
 *      Procedures to Manipulate Ports                                   *
                                                                         *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

 *)

(*************************************************************************
 *                                                                       *
 *      MakSOPort - Create a shell output port for the current object    *
 *      and name it nam.                                                 *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakSOPort(nam: ARRAY OF CHAR): pPort;



(*************************************************************************
 *                                                                       *
 *      MakSIPort - Create a shell input port for the current object     *
 *      and name it nam.                                                 *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakSIPort(nam: ARRAY OF CHAR): pPort;
```

```
(*************************************************************************
 *                                                                       *
 *      MakOPort - Create an output port for the current object          *
 *      and name it nam.                                                 *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakOPort(nam: ARRAY OF CHAR): pPort;


(*************************************************************************
 *                                                                       *
 *      MakIPort - Create an input port for the current object           *
 *      with an input server (Server), server context (ct), and          *
 *      name it nam.                                                     *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakIPort(nam: ARRAY OF CHAR; Server: SrvrPrc;
        ct: ADDRESS): pPort;


(*************************************************************************
 *                                                                       *
 *      MakIRPort - Create a read-only input port for the current        *
 *      object with an input server (Server), server context (ct),       *
 *      and name it nam.                                                 *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakIRPort(nam: ARRAY OF CHAR; Server: SrvrPrc;
        ct: ADDRESS): pPort;
```

```
(*************************************************************************
 *                                                                       *
 *      NilSrvr - A do nothing input port server.                        *
 *                                                                       *
 *************************************************************************)

PROCEDURE NilSrvr(VAR ms: pMsg; VAR data: ADDRESS; ct: ADDRESS);


(*************************************************************************
 *                                                                       *
 *      DelPort - Delete the port (pt).                                  *
 *                                                                       *
 *************************************************************************)

PROCEDURE DelPort(pt: pPort);


(*************************************************************************
 *                                                                       *
 *      FindPort - Search the object, Owner, for a port by the name      *
 *      of nam, and return a pointer to it.                              *
 *                                                                       *
 *************************************************************************)

PROCEDURE FindPort(Owner: pObject; nam: ARRAY OF CHAR): pPort;


(*************************************************************************
 *                                                                       *
 *      FindPortS - Silent version of FindPort.  Same as FindPort        *
 *      except if the port is not found, no error message is printed     *
 *      (simply return NIL).                                             *
 *                                                                       *
 *************************************************************************)

PROCEDURE FindPortS(Owner: pObject; nam: ARRAY OF CHAR): pPort;
```

```
(*************************************************************************
 *                                                                       *
 *      FindIPort - Search the object, Owner, for a port by the name     *
 *      of nam, and return a pointer to it.  Only look for input ports.  *
 *                                                                       *
 *************************************************************************)

PROCEDURE FindIPort(Owner: pObject; nam: ARRAY OF CHAR): pPort;


(*************************************************************************
 *                                                                       *
 *      FindOPort - Search the object, Owner, for a port by the name     *
 *      of nam, and return a pointer to it.  Only look for output        *
 *      ports.                                                           *
 *                                                                       *
 *************************************************************************)

PROCEDURE FindOPort(Owner: pObject; nam: ARRAY OF CHAR): pPort;


(*************************************************************************
 *                                                                       *
 *      FindIPortS - Same as FindPortS but look for input port.          *
 *      If the port is not found, no error message is printed            *
 *      (simply return NIL).                                             *
 *                                                                       *
 *************************************************************************)

PROCEDURE FindIPortS(Owner: pObject; nam: ARRAY OF CHAR): pPort;
```

```
(*************************************************************************
 *                                                                       *
 *      FindOPortS - Same as FindPortS but look for output port.         *
 *      If the port is not found, no error message is printed            *
 *      (simply return NIL).                                             *
 *                                                                       *
 *************************************************************************)

PROCEDURE FindOPortS(Owner: pObject; nam: ARRAY OF CHAR): pPort;


(*************************************************************************
 *                                                                       *
 *      Connect - Connect the message source (srcpt), to the message     *
 *      destination (dstpt).                                             *
 *                                                                       *
 *************************************************************************)

PROCEDURE Connect(srcpt, dstpt: pPort);


(*************************************************************************
 *                                                                       *
 *      Disconnect - Disconnect the message source (srcpt), from the     *
 *      message destination (dstpt).                                     *
 *                                                                       *
 *************************************************************************)

PROCEDURE Disconnect(srcpt, dstpt: pPort);


(*************************************************************************
 *                                                                       *
 *      SetKey - Set the key of the input port (pt) to ky and msk.       *
 *                                                                       *
 *************************************************************************)

PROCEDURE SetKey(pt: pPort; ky, msk: WORD);
```

```
(*************************************************************************
 *                                                                       *
 *      SetPortTag - Set the tag of the input port (pt) to the user      *
 *      defined value (tg).                                              *
 *                                                                       *
 *************************************************************************)

PROCEDURE SetPortTag(pt: pPort; tg: INTEGER);


(*************************************************************************
 *                                                                       *
 *      MakGSMPort - Create a port for the current object with an        *
 *      input server (Server), and a server context (ct), to receive     *
 *      General System Messages of type, ty.  Associate tg (tag) with    *
 *      the port.    Note:  This is a READ-ONLY port!                    *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakGSMPort(ty: CARDINAL; Server: SrvrPrc; ct: ADDRESS;
        tg: INTEGER);


(*************************************************************************
 *                                                                       *
 *      DelGSMPort - Delete the GSM port of type, ty, on the             *
 *      currently executing object.                                      *
 *                                                                       *
 *************************************************************************)

PROCEDURE DelGSMPort(ty: CARDINAL);
```

```
(*
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                                                                         *
 *      Procedures to Manipulate Messages                                  *
 *                                                                         *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

 *)

(*************************************************************************
 *                                                                       *
 *      MakMsg - Create a message of length, datalen.  Data field is     *
 *      not filled in.                                                   *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakMsg(datalen: CARDINAL): pMsg;


(*************************************************************************
 *                                                                       *
 *      MakMsgD - Create a message of length, datalen.  Data field is    *
 *      copied from "data".                                              *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakMsgD(data: ADDRESS; datalen: CARDINAL): pMsg;


(*************************************************************************
 *                                                                       *
 *      ChgMsg - Replace the message with a copy containing new data     *
 *      (newdata) having a new length (newlen).  Delete the old          *
 *      message and return a pointer to the new message.                 *
 *                                                                       *
 *************************************************************************)

PROCEDURE ChgMsg(oldms: pMsg; newdata: ADDRESS; newlen: CARDINAL): pMsg;
```

```
(*************************************************************************
 *                                                                       *
 *      DelMsg - Delete a message (ms) and its attributes, if any.       *
 *      If the message is in the queue, update the queue.                *
 *                                                                       *
 *************************************************************************)

PROCEDURE DelMsg(ms: pMsg);


(*************************************************************************
 *                                                                       *
 *      DupMsg - Creates a duplicate copy of the message (and its        *
 *      attributes) pointed to by srcms, and returns a pointer to        *
 *      the newly created message.                                       *
 *                                                                       *
 *************************************************************************)

PROCEDURE DupMsg(srcms: pMsg): pMsg;


(*************************************************************************
 *                                                                       *
 *      FindMsgsPrt - Search the queue for messages originating from     *
 *      the port, srcpt, and place them in the edit list.                *
 *                                                                       *
 *************************************************************************)

PROCEDURE FindMsgsPrt(srcpt: pPort);


(*************************************************************************
 *                                                                       *
 *      FindMsgsObj - Search the queue for messages originating from     *
 *      the object, srcob, and place them in the edit list.              *
 *                                                                       *
 *************************************************************************)

PROCEDURE FindMsgsObj(srcob: pObject);
```

```
(*************************************************************************
 *                                                                       *
 *      FindNext - Return the next message in the edit list.  If         *
 *      empty, return Nil.                                               *
 *                                                                       *
 *************************************************************************)

PROCEDURE FindNext(): pMsg;


(*************************************************************************
 *                                                                       *
 *      Resched - Reschedule the message, ms, delaying it by timincr     *
 *      units of time.                                                   *
 *                                                                       *
 *************************************************************************)

PROCEDURE Resched(ms: pMsg; timincr: CARDINAL);


(*************************************************************************
 *                                                                       *
 *      DelayMsgsPrt - Find all messages in the queue originating from   *
 *      the port, srcpt, and delay them (reschedule) by timincr units    *
 *      of time.                                                         *
 *                                                                       *
 *************************************************************************)

PROCEDURE DelayMsgsPrt(srcpt: pPort; timincr: CARDINAL);


(*************************************************************************
 *                                                                       *
 *      DelayMsgsObj - Find all messages in the queue originating from   *
 *      the object, srcob, and delay them (reschedule) by timincr units  *
 *      of time.                                                         *
 *                                                                       *
 *************************************************************************)

PROCEDURE DelayMsgsObj(srcob: pObject; timincr: CARDINAL);
```

```
(*************************************************************************
 *                                                                       *
 *      Send - Send the message (ms) out of port (pt), to be received    *
 *      timincr units of time from now (CurTime).                        *
 *                                                                       *
 *************************************************************************)

PROCEDURE Send(ms: pMsg; pt: pPort; timincr: CARDINAL);


(*************************************************************************
 *                                                                       *
 *      SendMsg - Send data of length (len), as a message out of         *
 *      port (pt), to be received timincr units of time from now         *
 *      (CurTime).                                                       *
 *                                                                       *
 *************************************************************************)

PROCEDURE SendMsg(data: ADDRESS; len: CARDINAL; pt: pPort;
        timincr: CARDINAL);


(*************************************************************************
 *                                                                       *
 *      KeyedSend - Send the message (ms) out of port (pt), to be        *
 *      received timincr units of time from now (CurTime).  The          *
 *      message is only distributed to/through input ports with          *
 *      matching keys.  The matching test is:                            *
 *              Message Key AND Port Mask = Port Key AND Message Mask    *
 *                                                                       *
 *************************************************************************)

PROCEDURE KeyedSend(ms: pMsg; pt: pPort; timincr: CARDINAL;
        ky, msk: WORD);
```

```
(*************************************************************************
 *                                                                       *
 *      KeyedSendMsg - Send data of length (len), as a message out       *
 *      of port (pt), to be received timincr units of time from now      *
 *      (CurTime).  The message is only distributed to/through input     *
 *      ports with matching keys.  The matching test is:                 *
 *              Message Key AND Port Mask = Port Key AND Message Mask    *
 *                                                                       *
 *************************************************************************)

PROCEDURE KeyedSendMsg(data: ADDRESS; len: CARDINAL; pt: pPort;
        timincr: CARDINAL; ky, msk: WORD);



(*************************************************************************
 *                                                                       *
 *      DefGSMTyp - Define a new GSM type with the name, nam,            *
 *      and return its corresponding type.                               *
 *                                                                       *
 *************************************************************************)

PROCEDURE DefGSMTyp(nam: ARRAY OF CHAR): CARDINAL;



(*************************************************************************
 *                                                                       *
 *      SendGSM - Create a General System Message of type (ty), and      *
 *      containing data (GSMdata), of length (datalen), and send it      *
 *      to be received timincr units of time from now (CurTime).         *
 *                                                                       *
 *************************************************************************)

PROCEDURE SendGSM(GSMdata: ADDRESS; datalen: CARDINAL; ty: CARDINAL;
        timincr: CARDINAL);
```

```
(*
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                                                                         *
 *      Procedures to Manipulate Attributes                                *
 *                                                                         *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

 *)

(*************************************************************************
 *                                                                       *
 *      DefAttrTyp - Define a new attribute type with the name,          *
 *      nam, and return its corresponding type.                          *
 *                                                                       *
 *************************************************************************)

PROCEDURE DefAttrTyp(nam: ARRAY OF CHAR): CARDINAL;


(*************************************************************************
 *                                                                       *
 *      MakAttr - Create an attribute of length, datalen, and            *
 *      type, ty, and attach it to message, ms.                          *
 *                                                                       *
 *************************************************************************)

PROCEDURE MakAttr(ty: CARDINAL; datalen: CARDINAL; ms: pMsg):  pAttrib;


(*************************************************************************
 *                                                                       *
 *      FindAttr - Search the message, ms, for an attached               *
 *      attribute of type, ty.                                           *
 *                                                                       *
 *************************************************************************)

PROCEDURE FindAttr(ty: CARDINAL; ms: pMsg): pAttrib;
```

```
(*************************************************************************
 *                                                                       *
 *      ChgAttr - Replace the attribute with a copy containing new       *
 *      data (newdata) having a new length (newlen).  Delete the old     *
 *      attribute and return a pointer to the new attribute.             *
 *                                                                       *
 *************************************************************************)

PROCEDURE ChgAttr(oldat: pAttrib; newdata: ADDRESS;
        newlen: CARDINAL): pAttrib;



(*************************************************************************
 *                                                                       *
 *      DelAttr - Delete an attribute (at).                              *
 *                                                                       *
 *************************************************************************)

PROCEDURE DelAttr(at: pAttrib);



(*************************************************************************
 *                                                                       *
 *      DupAttrs - Creates a duplicate copy of each attribute            *
 *      attached to the message pointed to by srcms, and appends         *
 *      the newly created attributes to the message pointed to by        *
 *      dstms.                                                           *
 *                                                                       *
 *************************************************************************)

PROCEDURE DupAttrs(srcms, dstms: pMsg);
```

```
(*
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                                                                         *
 *      Execution and Miscellaneous Procedures                             *
 *                                                                         *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

 *)

(*************************************************************************
 *                                                                       *
 *      StopAt - Force simulation to stop (and return from Simulate)     *
 *      at the specified era and time.                                   *
 *                                                                       *
 *************************************************************************)

PROCEDURE StopAt(era, time: CARDINAL);


(*************************************************************************
 *                                                                       *
 *      Simulate - Begin simulation execution.  Simulation continues     *
 *      until no messages are left in the queue or the "StopAt" time     *
 *      is reached.  If the "StopAt" time is reached, simulation can     *
 *      be continued by setting the StopAt time higher and calling       *
 *      Simulate.  Upon returning, the configuration is still intact     *
 *      so that another run using the same configuration can be          *
 *      performed (see ResetClock).                                      *
 *                                                                       *
 *************************************************************************)

PROCEDURE Simulate();
```

```
(*************************************************************************
 *                                                                       *
 *      ResetClock - Removes any messages from the event queue and       *
 *      sets the current era and time to zero, so that another run       *
 *      may be performed using the same configuration.                   *
 *                                                                       *
 *************************************************************************)

PROCEDURE ResetClock();


(*************************************************************************
 *                                                                       *
 *      Reset - After returning from Simulate, Reset can be called       *
 *      to delete the current configuration, so that a new one can       *
 *      be constructed for a subsequent run.  (Reset automatically       *
 *      calls ResetClock.)  Only attribute types, GSM types, and         *
 *      the external symbol table are preserved.                         *
 *                                                                       *
 *************************************************************************)

PROCEDURE Reset();


(*************************************************************************
 *                                                                       *
 *      GetMem - Allocate a block of memory, of length size, and         *
 *      return the beginning address in mem.  Note:  GetMem will         *
 *      allocate only integral multiples of whole WORDs.                 *
 *                                                                       *
 *************************************************************************)

PROCEDURE GetMem(VAR mem: ADDRESS; size: CARDINAL);
```

```
(*************************************************************************
 *                                                                       *
 *      FreeMem - Deallocate a block of memory starting at mem of        *
 *      length size.                                                     *
 *                                                                       *
 *************************************************************************)

PROCEDURE FreeMem(mem: ADDRESS; size: CARDINAL);


(*************************************************************************
 *                                                                       *
 *      Copy - Copy a memory block of length len, from "src" to "dst".   *
 *      Note:  The copy operation is performed a WORD at a time (len     *
 *      is rounded up to the next whole WORD).                           *
 *                                                                       *
 *************************************************************************)

PROCEDURE Copy(src, dst: ADDRESS; len: CARDINAL);


(*************************************************************************
 *                                                                       *
 *      EnterName - Searches the symbol table for nam.  If not there     *
 *      it creates an entry.  A pointer to the entry is returned.        *
 *                                                                       *
 *************************************************************************)

PROCEDURE EnterName(nam: ARRAY OF CHAR): pXName;
```

```
(*************************************************************************
 *                                                                       *
 *      Wait - Wait for the next message from any port, and call its     *
 *      input server.  A pointer to the resulting message, and its       *
 *      data record are returned.  If the pointer to the data record     *
 *      (dat) is Nil, Wait waits for the next message to arrive.         *
 *                                                                       *
 *************************************************************************)

PROCEDURE Wait(VAR ms: pMsg; VAR dat: ADDRESS);


END Simon.
```

Appendix II
Simulation Debugging

Simon is distributed with a debugging module called SimonDebug, which provides simple execution tracing capabilities and error reporting. The Simon kernel requires SimonDebug to always be included when the simulation program is linked. A user customized version of SimonDebug may be used instead of the distributed version, if desired (see Installation Notes). It should also be understood that the use of SimonDebug in no way precludes the use of debugging tools provided by the host computer system.

II.1. Execution Tracing

Simon Debug provides two types of tracing: structure and message tracing. When structure tracing is enabled, SimonDebug reports all changes to the configuration structure (objects, ports, connections). When message tracing is enabled, SimonDebug reports whenever a message is sent or received. Tracing is enabled and disabled by calling the following SimonDebug procedures. (Initially, tracing is disabled.)

PROCEDURE TraceOn ( );

TraceOn turns on structure and message tracing.

PROCEDURE TraceOff ( );

TraceOff turns off structure and message tracing.

PROCEDURE STraceOn ( );

STraceOn turns on structure tracing.

PROCEDURE STraceOff ( );

STraceOff turns off structure tracing.

PROCEDURE MTraceOn ( );

MTraceOn turns on message tracing.

PROCEDURE MTraceOff ( );

MTraceOff turns off message tracing.

The first pair of procedures enables and disables both structure and message tracing. The second pair affects only structure tracing, and the third pair affects only message tracing.

II.2. Error Reporting

SimonDebug provides an error reporter called "Error". This procedure is used to report all Simon errors, but can also be used by user programs. The first line of an error message appears as follows:

```
! Error Detected by <procedure name> While Executing <object name>
```

The second line of the error message is passed to Error by the calling procedure. After reporting the error, Error returns to the caller. The arguments for calling Error are:

```
PROCEDURE Error (   ProcNam:    ARRAY OF CHAR;
                    ErrMsg:     ARRAY OF CHAR;
                    Param1:     ADDRESS;
                    Param2:     ADDRESS         );
```

ProcNam	is the string name of the calling (reporting) procedure.
ErrMsg	is the string text of the second line of the error message.
Param1	is the address of an item to be formatted into ErrMsg.
Param2	is the address of an item to be formatted into ErrMsg.

Up to two items may be formatted into the ErrMsg string in a manner similar to printf in the "C" programming language. Formatting proceeds as follows. If a "%" (percent character) is encountered in the ErrMsg string, it signifies the point at which an item is to be formatted and inserted into the error message. The character following the "%" determines the type of the item and how it is to be formatted. The following characters are valid:

o	generates the external string name of the object.
p	generates the external string name of the port.
t	generates the port type (Shell Output, Shell Input, Output, Input, Input Read-Only).
a	generates the external string name of an attribute type.
g	generates the external string name of a GSM type.
s	formats a string variable (ARRAY OF CHAR).
d	formats an INTEGER or CARDINAL variable.

All other characters are ignored. If less than two items are to be formatted, then the unused parameters (Param2, Param1) should be Nil. For example, the Simon procedure SetKey is used to set the port key for input ports only. If the specified port "pt" (of type pPort) is not an input port, SetKey makes the following call:

```
Error("SetKey", "Inappropriate Operation for %t Port, %p", pt, pt)
```

In addition to Error, SimonDebug provides procedures to output the external string name of an object, port, port type, attribute type, or GSM type:

```
PROCEDURE PrintObjectName (
                        ob:         pObject         );

PROCEDURE PrintPortName (
                        pt:         pPort           );

PROCEDURE PrintPortType (
                        pt:         pPort           );

PROCEDURE PrintAttrType (
                        ptype:      ADDRESS         );

PROCEDURE PrintGSMType (
                        ptype:      ADDRESS         );
```

Note that while the parameter for the first three procedures is a pointer to the respective data structure, the parameter for the last two procedures is a pointer to a CARDINAL (the type field).

Appendix III
Installation Notes

When Simon is installed on a particular machine, some slight adjustments to the code may be necessary.

III.1. TSIZE Resolution Versus Address Resolution

Modula 2 should be implemented so that the units of TSIZE match the address resolution of the host machine. However, this is not always the case. Simon is designed to use the same unit size as TSIZE, whatever it is. Nevertheless, there is a line of code in GetMem (preceded by a special comment, "(*!*)") which is sensitive to this issue:

```
MemBlk:=MemBlk + size DIV TSIZE(CHAR);
```

The value in size is in TSIZE units. Dividing it by TSIZE(CHAR) converts it to addressable units (assuming a byte addressable machine). If the basic addressable unit is something else (such as WORD), then size should be divided by TSIZE of the addressable unit.

III.2. Memory Management

Near the beginning of the Simon implementation module (about the second page) are three constants which affect memory management (GetMem, FreeMem): UnitSiz, AllocBlkSiz, and MaxBlkSiz. Internal to Simon, memory is allocated and deallocated in units of "granules". The constant "UnitSiz" determines how big a granule is in TSIZE units. UnitSiz must be a power of two, and must be greater than or equal to TSIZE(ADDRESS), and must be an integral multiple of TSIZE(WORD). The second constant "AllocBlkSiz" is the size (in units of granules) of memory blocks which Simon gets from the operating system when it needs more memory. Simon maintains free chunks of memory in an array of bins. Each bin contains only one size of memory chunks. Since the array is fixed in size, only requests for sizes up to a certain limit can be handled. Larger requests are directly passed on to ALLOCATE and DEALLOCATE. The third constant "MaxBlkSiz" determines the maximum request size (in units of granules) which will be handled directly by Simon. It must be less than or equal to AllocBlkSiz.

III.3. Hash Table Sizes

The symbol table (containing external string names) and the GSM port list are implemented using hashing. In either case, Simon calculates a hash value which is then divided by the hash table size. The remainder is used as an index into the hash table. The hash table then points to a linked list which is searched serially for the desired entry. Obviously, the larger the hash table, the faster the search, but more memory is required. Near the beginning of the Simon definition module (about the second page) are the constants which determine the hash table sizes. SHshSiz sets the size of the symbol table hash table, and GHshSiz sets the size of the GSM port hash table.

III.4. Clock Limit

A couple of lines down from the hash table constants is the constant "TicLimit". The value of TicLimit determines the maximum value of the global clock "CurTime". When CurTime equals or exceeds this limit, TicLimit is subtracted from it and the clock overflow "CurEra" is incremented. TicLimit should be set to the maximum power of ten that can be contained in a CARDINAL variable. This allows CurEra and CurTime to be output as a single (concatenated) number without requiring the use of a complicated "double-precision" division procedure.

III.5. SimonDebug

SimonDebug is a module supplied with the distribution of Simon II. Although it will probably meet most user's needs for debugging and error reporting, it's possible that a more sophisticated debugger may be needed. SimonDebug was written in a separate module so that the user could substitute his own debugger module if necessary. Any debugger module must provide a compatible interface including: exporting of an error reporting procedure (Error) and all trace procedures, and importing and setting the trace flags (STrace and MTrace). The SimonDebug source code should be examined to determine the details of the procedure arguments. An interactive customized debugger module could provide a variety of capabilities, including the ability to examine the system configuration. This is possible since the Simon kernel exports all of its data types and data structures, including the symbol table of external names. (See the Simon definition module - Appendix I.)

SimonDebug was written using the I/O procedures provided by the DEC WRL implementation of Modula 2. If Simon is to be installed on a different machine, it will be necessary to change all of the I/O calls in the SimonDebug module. (In the distributed version of SimonDebug the I/O procedures have parameters which are very similar to those of the scanf and printf standard functions in the "C" language.) Since the Simon kernel contains no I/O calls, it requires no changes.

Index