User's Guide
CSIM20 Simulation Engine
(C++ Version)

Mesquite Software, Inc.
P.O. Box 26306
Austin, TX 78755-0306
(512) 338-9153
info@mesquite.com
www.mesquite.com

Mesquite Software, Inc.

Table of Contents

Table of Contents

Table of Contents

1 Introduction

CSIM[1] is a process-oriented discrete-event simulation package for use with C or C++ programs. It is implemented as a library of classes and procedures which implement all of the necessary structures and operations. The end result is a convenient tool which programmers can use to create simulation programs.

A CSIM program models a system as a collection of CSIM processes which interact with each other by using the CSIM structures. The purpose of modeling a system is to produce estimates of time and performance. The model maintains simulated time, so that the model can yield insight into the dynamic behavior of the modeled system.

This document provides a description of:

- CSIM structures (objects) and the statements that manipulate them
- Reports available from CSIM
- Information on compiling, executing and debugging CSIM programs.

[1] CSIM is copyrighted by Microelectronics and Computer Technology Corporation, 1987 – 1994 and by Mesquite Software, Inc., 1995 – 2009.

1.1 CSIM Objects (Static and Dynamic)

The C++ version of CSIM provides a number of classes which define the objects CSIM models use. In many cases, you can choose to use either static or dynamic instances of these classes (each instance is an object).

In many cases, it may be preferable to use dynamic objects. The reason for this is that with dynamic objects, you have explicit control over construction and destruction of each object. With static global objects, each object is constructed before the program begins execution and they cannot be deleted. With local static objects, each object is constructed when the procedure starts and is deleted when the procedure exits.

In some CSIM models, the program can use the *rerun* procedure to “tear down” a model at the end of a run so that a new model can be built for the next run. In this case, dynamic objects must be used.

For this reason, all of the examples in the following sections will use dynamic classes. However, static classes can be used in almost all cases.

CSIM provides the following simulation objects:

- Processes - the active entities that request service at facilities, wait for events, etc. (i.e. processes deal with all of the other structures in this list)
- The facility class - queues and servers reserved or used by processes
- The storage class - resources which can be partially allocated to processes
- The buffer class – resources which can be partially allocated to processes; a buffer can be empty and/or partially full

- The event class - used to synchronize process activities
- The mailbox class - used for inter-process communications
- Data collection structures - used to collect data during the execution of a model
- Process classes - used to segregate statistics for reporting purposes
- Streams - streams of random numbers

It is the processes which mimic the behavior of active entities in the simulated system.

1.2 Syntax Notes

- All parameters are required.
- Whenever a parameter is included within double quotes (e.g. “name”), it can also be passed as a pointer to a character array which contains the string.

Constants, which are represented by names that are entirely in capital letters, are defined in the header file, “csim.h”.

1. Introduction

2 Simulation Time

Time is an important concept in any performance model. CSIM maintains a simulation clock whose value is the current time in the model. This simulation time is distinctly different than the cpu time used in executing the model or the “real world” time of the person running the model. Simulation time starts at zero and then advances unevenly, jumping between times at which the state of the model changes. It is impossible to make time move backwards during a simulation run.

The simulation clock is implemented as a double precision floating point variable in CSIM. For most models there is no need to worry that the simulation clock will overflow or that round-off error will impact the accuracy of the clock.

The simulation clock is used extensively within CSIM to schedule events and to update performance statistics. CSIM processes may retrieve the current time for their own purposes and may indirectly cause time to advance by performing certain operations.

2.1 Choosing a Time Unit

The CSIM simulation clock has no predefined unit of time. It is the responsibility of the modeler to choose an appropriate time unit and to consistently specify all amounts of time in that unit. All performance statistics reported by CSIM should also be interpreted as being in that chosen time unit.

A good time unit might be close to the granularity of the smallest time periods in the model. For example, if the smallest time periods being modeled are on the order of tens of milliseconds, an appropriate time unit might be either milliseconds or

seconds. Using microseconds or minutes as the time unit would produce performance statistics that are either very large or very small numbers.

Most numbers appearing in CSIM performance reports are printed with up to six digits to the left of the decimal point and six digits to the right of the decimal point. A time unit should be chosen to avoid numbers so large that they overflow their fields or so small that interesting digits are not visible.

2.2 Retrieving the Current Time

There are two equivalent ways to retrieve the current value of the simulation clock. One is to call the *simtime* function.

Prototype: `double simtime(void)`

Example:
```
x = simtime();
```

The other is to reference the variable *clock*.

Example:
```
x = clock; or
x = csim_clock;
```

2.3 Delaying for an Amount of Time

A CSIM process can delay for a specified amount of simulation time by calling the *hold* function.

Prototype: `void hold(double amount_of_time)`

Example: `hold(1.0);`

If there are other processes waiting to run, the calling process will be suspended. Otherwise, time will immediately advance by the specified amount.

Caution: It is a common mistake to call *hold* with the wrong type of parameter, such as an integer value.

A process can delay until a specified time by calling *hold* with a parameter value equal to the specified time minus the current time. To make a simulation begin with a clock value other than zero, simply call *hold* at the beginning of the *sim* function with an amount of time equal to the desired initial time.

Calling the *hold* function with a zero amount of time might at first seem to be meaningless. But, it causes the running process to relinquish control to any other process that is waiting to run at the same simulation time. This can be used to affect the order of execution of processes that have activities scheduled for the same simulation time.

2.4 Advancing Time

There is no way for a program to directly assign a value to the simulation clock. The simulation clock advances as a side effect of a process performing one of the following function calls:

`hold`	`allocate`	`get`
`put`	`wait`	`queue`
`use`	`timed_allocate`	`timed_put`
`timed_get`	`wait_any`	`queue_any`
`reserve`	`receive`	`timed_wait`
`timed_queue`	`timed_reserve`	`timed_receive`

```
synchronous_send  timed_synchronous_send
```

Calling one of these functions does not guarantee that time will advance. For example, calling the *allocate* function will cause time to pass only if the requested amount of storage is not available.

All CSIM function calls other than those in the above list, as well as all C++ language statements, occur instantaneously with respect to simulation time. A CSIM program can perform arbitrarily many activities in a single instant of simulation time.

A common programming error is to create a CSIM process that calls none of the functions in the above list. When this process receives control, it runs endlessly to the exclusion of all other CSIM processes.

2.5 Displaying the Time

There are several ways the simulation time can be automatically displayed while running a CSIM program. Every trace message contains the current simulation time. The variable *clock* and the function *simtime()* can be used to get the current simulated time. Also, when the *report* function is called to produce a report of all statistics, the report header contains the current simulation time.

2.6 Integer-Valued Simulation Time

In some simulation models, such as models of computer hardware, it is the case that time can only assume discrete integer values. Although CSIM maintains time as a floating point variable, some simple programming techniques can insure that the

clock will always have an integer value. (Here, we are using the word “integer” in the mathematical sense.)

Amounts of time appear as input parameters in calls to the following functions: *hold*, *use, timed_reserve, timed_allocate, timed_put, timed_get, timed_receive, timed_synchronous_send*, *timed_wait*, *timed_queue, time_wait_any* and *timed_queue_any*. To maintain an integer-valued clock, these parameters must have values that are integers (although of type double). This can be accomplished either by specifying a floating point numeric literal that has an integer value or by type casting an integer expression to type double.

Example:
```
hold(10.0);
```

Example:
```
bus->use((double) uniform_int(1,5));
```

Example:
```
bus->use((double) floor (exponential(1.0)));
```

The IEEE Floating Point Standard guarantees that addition and subtraction with integer valued operands will yield integer valued results. CSIM performs only addition on the simulation clock.

3 Processes

Processes represent the active entities in a CSIM model. For example, in a model of a bank, customers might be modeled as processes (and tellers as facilities). In CSIM, a process is a C++ procedure which executes a *create* statement. A CSIM process should not be confused with a UNIX process (which is an entirely different thing). The *create* statement is similar to a UNIX “fork" statement. A process can be invoked with input arguments, but it cannot return a value to the invoking process.

There can be several simultaneously "active" instances of the same process. Each of these instances appears to be executing in parallel (in simulated time) even though they are in fact executing sequentially on a single processor. The CSIM runtime package guarantees that each instance of every process has its own runtime environment. This environment includes local (automatic) variables and input arguments. All processes have access to the global variables of a program.

A CSIM process, just like a real process, can be in one of four states:

- Actively computing
- Ready to begin computing
- Holding (allowing simulated time to pass)
- Waiting for an event to happen (or a facility to become available, etc.)

When an instance of a process terminates, either explicitly or via a procedure exit, it is deleted from the CSIM system. Each process has a unique process id and each has a priority associated with it.

3.1 Initiating a Process

In CSIM, a process is a procedure which executes a *create* statement; a process is initiated (invoked, started, …) by executing a standard procedure call:

Prototype: `void proc(arg1, ..., argn);`

Example: `my_proc(a, 64, "label");`

Caution: It is bad practice to pass the address of a local variable to a CSIM process as an input argument.

Caution: A process cannot return a function value.

Caution: A create statement (see below) must appear in the initiated process.

The calling process can obtain the process identity of the most recently invoked process by calling the following function:

Prototype: `long last_process_id( );`

Example: `id = last_process_id();`

3.2 Executing the Process CREATE Statement

As stated above, a CSIM process is a procedure which executes the *create* statement:

Prototype: `void create(char * name)`

Example: `create("customer");`

The name of a process is just a character string which is used to identify the process in event traces and reports generated by CSIM. Typically, the *create* statement is executed at the beginning of a process. Each instance of a process is given a

unique process id (process id's are not reused). Processes can invoke procedures and functions in any manner. Processes can also initiate other processes.

When a procedure executes its *create* statement, the following actions take place:

- The process executing the *create* statement (the called process) is established and is made "ready to execute" at the statement following the *create* statement, and
- The calling process continues its execution (*i.e*., it remains the actively computing process) at the statement after the procedure call to the called process.

The calling process continues as the active process until it suspends itself.

No simulated time passes during the execution of a *create* statement.

Note: process id's are 32 bit integers; if the id value 2^{31}-1 is used, the sequence of id's is reset to 2.

3.3 Process Operation

Processes appear to operate simultaneously with other active processes at the same points in simulated time. The CSIM process manager creates this illusion by starting and suspending processes as time advances and as events occur. Processes execute until they "suspend" themselves by doing one of the following actions:

- execute a *hold* statement (delay for a specified interval of time),
- execute a statement which causes the processes to be placed in a queue, or
- terminate.

Processes are restarted when the time specified in a *hold* statement elapses or when a delay in a queue ends. It should be noted that simulated time passes only by the execution of *hold* statements. While a process is actively computing, no simulated time passes.

The process manager preserves the correct context for each instance of every process. In particular, separate versions of all local variables (variables resident in the runtime stack frame) and input arguments for a process are maintained. CSIM accomplishes this by saving and restoring process contexts (segments of the runtime stack) as processes suspend themselves and as processes are “resumed” (restored). A consequence of this kind of operation is that if one process passes an address of a local variable to another process, it is likely that when this address is referenced, the reference will be invalid. The reason is that when a process is not actually computing (using the real CPU), its stack frame with the local variables will not be physically located in the correct place in memory. This is not a major obstacle to writing efficient and useful models; it is a detail that must be remembered as CSIM models are developed.

3.4 Terminating a Process

A process terminates when it either does a normal procedure exit or when it executes a *csim_terminate* statement.

Prototype: `void csim_terminate(void)`

Example: `csim_terminate();`

The normal case is for a process to do a normal procedure exit or return. The *csim_terminate* statement is provided when this normal case is not appropriate.

3.5 Changing the Process Priority

The initial priority of a process is inherited from the initiator of that process. For the sim (main) process, the default priority is 1 (low priority).

Prototype: `void set_priority(long new_priority)`

Example: `set_priority(5);`

This statement must appear after the *create* statement in a process. Lower values represent lower priorities (*i.e*. priority 1 processes will run later than priority 2 processes when priority is a consideration in order of execution (see section 4.11, “Changing the Service Discipline at a Facility”)).

3.6 Inspector Functions

These functions each return some information to the process issuing the statement. The type of the returned value for each of these functions is as indicated.

Prototype:	Functional Value:
`char *process_name(void)`	retrieves pointer to name of process issuing the inquiry
`long identity(void)`	retrieves the identifier (process number) of process issuing the inquiry
`long priority(void)`	retrieves the priority of process issuing the inquiry

3.7 Reporting Process Status

To print the status of each active process in a model:

Prototype: `void status_processes(void)`

Example: `status_processes();`

To print the status of processes with pending state changes (the “next event list”):

Prototype: `void status_next_event_list(void)`

Example: `status_next_event_list();`

These reports will be written to the default output location or to that specified by *set_output_file* (see section 19.7, “Output File Selection”).

4 Facilities

A facility is normally used to model a resource (something a process requests service from) in a simulated system. For example, in a model of a computer system, a CPU and a disk drive might both be modeled by CSIM facilities. A simple facility consists of a single server and a single queue (for processes waiting to gain access to the server). Only one process at a time can be using a server. A multiserver-server facility contains a single queue and multiple servers. All of the waiting processes are placed in the queue until one of the servers becomes available. A facility set is an array of simple facilities; in essence, a facility set consists of multiple single server facilities, each with its own queue.

Normally, processes are ordered in a facility queue by their priority (a higher priority process is ahead of a lower priority process). In cases of ties in priorities, the order is first-come, first-served (fcfs). An fcfs facility can be designated as a synchronous facility. Each synchronous facility has its own clock with a period and a phase and all *reserve* operations are delayed until the onset of the next clock cycle. Service disciplines other than priority order can be established for a server. These are described in section 4.11, “Changing the Service Discipline at a Facility.”

A set of usage and queueing statistics is automatically maintained for each facility in a model. The statistics for all facilities which have been used are "printed" when either a *report* (see section 19.2, “CSIM Report Output”) or a *report_facilities* is executed (see section 4.4, “Producing Reports” for details about the reports that are generated). In addition, there is a set of inspector methods that can be used to extract individual statistics for each facility.

First time users of facilities should focus on the following four sections, which explain how to set up facilities, use (and reserve and release) facilities, and produce reports. Subsequent sections describe the more advanced features of facilities.

4.1 Declaring and Initializing a Facility

A dynamic facility pointer is declared:

Dynamic Example: `facility *fd`

Before a dynamic facility can be used, it must be initialized by invoking the *facility* constructor by calling *new*:

Prototype: `facility::facility(char *name)`

Static Example: `facility fs("fac")`

Dynamic Example: `fd = new facility("fac");`

A newly created facility is created with a single server that is "free." The facility name is used only to identify the facility in output reports and trace messages.

Facilities should be declared with global pointers and constructed in the first process (normally the process named *sim*) prior to the beginning of the simulation part of the model. Unless changed by a *set_servicefunc* method (see section 4.11, "Changing the Service Disciplines at a Facility"), the scheduling policy of the facility will be first-come, first-served (fcfs).

4.2 Using a Facility

A process typically uses a server for a specified interval of time.

Prototype: `void facility::use(double);`

Dynamic Example: `fd ->use(exponential(1.0));`

If the server at this facility is free (not being used by another process), then the process gains exclusive use of the server and the usage interval starts immediately.

At the end of the usage interval, the process gives up use of the server and departs this facility. Execution continues at the statement following the *use* statement.

If the server at this facility is busy (is being used by another process), then the newly arriving process is placed in a queue of waiting processes; this queue is ordered by process priority, with processes of equal priority being ordered by time of arrival. As each process completes its usage interval, the process at the head of the queue is assigned to the server and its usage interval starts at that time.

The service discipline at a facility specifies how processes are given access to the server. One of several different service disciplines can be specified for a facility. Also, another form of facility has multiple servers. In addition, it is possible to have an array of facilities (a facility set). The difference between a multiserver facility and a facility set is that a multiserver facility has one queue for all of the waiting processes, while a facility set has a separate queue for each facility in the set.

4.3 Reserving and Releasing a Facility

In some cases, a process will acquire a server, but will do something other than enter the usage interval when it gets the server. The statements for doing this are *reserve* (to gain exclusive use of a server) and *release* (to relinquish use of the server acquired in a previous *reserve* statement)

Prototypes:
```
long facility::reserve(void);
void facility::release(void);
```

Dynamic Examples:
```
fd->reserve();
fd->release();
```

When a process executes a *reserve*, it either gets use of the server immediately (if the server is not busy) or it is suspended and placed in a queue of processes waiting to get use of the server. When it gains access to the server, it executes the statement following the *reserve* statement. The order of processes in the queue is

by process priority, with processes of equal priority being ordered by time of arrival. This process priority service discipline is called *fcfs* in CSIM; it (along with *fcfs_sy*, see below) is the only service discipline that can be specified for facilities where processes do this *reserve-release* style of access. If another service discipline is in force, then the processes must execute *use* statements instead of *reserve-release* pairs of statements.

The process releasing a server at a facility must be the same process as the one which reserved it. If this is not the case, then the *release_server* statement (see below) must be used. When a process executes a *release*, it gives up use of the server; if there is at least one process waiting to start using the server (*i.e*., there is at least one process in the queue at this facility), the process at the head of the queue is given access to the server and that process is then reactivated and will proceed by executing the statement following its reserve statement. No simulation time passes during execution of a release statement.

Note: Executing the sequence “f ->reserve(); hold(t);f ->release();” is equivalent to executing the statement ”f ->use(t)’” However, if the usage interval is specified by a random number function, then there is a subtle difference between these, as follows: the randomly derived interval is determined *after* gaining access to the server in the first sequence and *before* gaining access to the server with the *use* form; thus it is likely that the intervals in these two examples will be different. In other words, the sequence ”f ->reserve(); hold(exponential(t); f->release();” will not necessarily display exactly the same behavior as executing the statement “f->use(exponential(t));”.

4.4 Producing Reports

Reports for facilities are most often produced by calling the *report* function which prints reports of all the CSIM objects. Reports can be produced for all existing facilities by calling the *report* method.

Prototype: `void facility_report(void);`

Example: `facility_report();`

The report for the set of facilities, as illustrated below, includes, for each facility, the name of the facility, the service discipline, the average service time, the utilization, the throughput rate, the average queue length, the average response time and the number of completed service requests.

FACILITY SUMMARY

facility name	service disc	service time	util.	through put	queue length	response time	compl count
f	fcfs	0.40907	0.208	0.50900	0.27059	0.53162	509
ms fac	fcfs	1.50020	0.764	0.50900	0.83821	1.64678	509
> server	0	1.55358	0.494	0.31800			318
> server	1	1.41133	0.270	0.19100			191
q	rnd_rob	0.73437	0.507	0.69000	0.95522	1.38438	690

The line in the report for each individual server can be omitted from or included in the report for each facility using the following statements:

Prototype: `void facility::server_stats_on(void)`

Prototype: `void facility::server_stats_off(void)`

4.5 Resetting a Facility

In some cases, it is necessary to reset the statistics counters for a specific facility.

Prototype: `void facility::reset(void)`

Example: `f->reset();`

Executing this statement does not affect the state of the facility or its servers. The reset()and the reset_facilities statements each call facility::reset() for all facilities in the model.

4.6 Releasing a Specific Server at a Facility

Sometimes, it is necessary for one process to reserve a facility and then for another process to release the server obtained by the first process. In this case, the first process has to save the index of the server it obtained, and then give this server index to the second process, so that it can specify that index in the *release_server* statement, as follows:

Dynamic Example:
```
server_index = fd->reserve();
```

Prototype:
```
void facility::release_server (long
        server_index)
```

Dynamic Example:
```
fd->release_server(server_index);
```

This operates in the same way as the *release* statement except that the ownership of the server is not checked; thus, a process which did not reserve the facility may release it by executing the *release_server* statement with a server index.

4.7 Declaring and Initializing a Multiserver Facility

In some cases, a facility has multiple servers, and each of these servers is indistinguishable from the other servers. A pointer to a mutliserver facility is declared:

Dynamic Example:
```
facility_ms *cpud;
```

A multiserver facility is constructed:

Prototype:
```
facility_ms::facility_ms(char *name,
                         long ns);
```

Static Example:
```
facility_ms cpus("dual cpu", 2);
```

Dynamic Example:
```
cpud = new facility_ms("dual cpu", 2);
```

A process can either execute a *use* statement or the *reserve-release* pair of statements at a multiserver facility. In either case, the process gains access to any server that is free; a process is suspended and put in the single queue at the facility only when all of the servers are busy.

4.8 Facility Sets

A facility set is an array of facilities. A facility set is initialized as follows:

Prototype:
```
facility_set::facility_set(char *name,
 long nm_facs);
```

Static Example:
```
facility_set disks("disks", 10);
```

Dynamic Example:
```
facility_set *diskd;
diskd = new facility_set("disk",10);
```

In a facility set, each element of the set is an independent, single server facility, with its own queue. Each of these facilities is given a constructed name which shows its position in the set. In the above example, the name for the first element of the set is "disk[0]". Facility sets are used to model cases where each server has its own queue of waiting processes.

An individual element of a facility is accessed as an array element. In C++ this is non-obvious.

Static Example: `disks[i].use(exponential(1.0));`

Dynamic Example: `(*diskd)[i].use(exponential(1.0));`

4.9 Reserving a Facility with a Time-out

Sometimes a process must not wait indefinitely to gain access to a server. If a process executes the timed_reserve function, it will be suspended until either it gains use of a server or the specified time-out interval expires.

Prototype:
```
long facility::timed_reserve(double
                             timeout)
```

Dynamic Example:
```
result = fd->timed_reserve (100.0);
if (result ! = TIMED_OUT) . . .
```

The process must check the functional value, to determine whether or not it obtained a server. If the value TIMED_OUT is returned, the process did not obtain a server. If this is not returned (EVENT_OCCURRED will in fact be returned), then the process did obtain a server and should eventually release the server.

4.10 Renaming a Facility

The name of a facility can be changed at any time, as follows:

Prototype: `void facility::set_name(char *new_name);`

Dynamic Example: `fd->set_name("cpu");`

Only the first ten characters of the facility's name are stored.

4.11 Changing the Service Discipline at a Facility

The service discipline for a facility determines the order in which processes at the facility are given access to that facility. If not otherwise specified, the service discipline for a facility is fcfs. When the priorities differ, processes gain access to the server in priority order (higher priority processes before lower priority processes). When processes have the same priority, the processes gain access in the order of their arrival at the facility (first come, first served). This default service discipline can be changed.

Prototype: `void facility::set_servicefunc(void (*service_func)())`

Dynamic Example: `fd->servicefunc(pre_res);`

Set_servicefunc() refers to a service function which is invoked when the *use* method (described above) references this facility. This service function can be any of the following pre-defined service discipline functions:

- **fcfs** – first come, first served

  This is the default service discipline and is described in the introduction to this section. If the *synchronous_facility*

statement (see below) is used for this facility, this will behave like a *fcfs_sy* (clock synchronized fcfs) facility. In other words, there are two ways for a facility to become synchronized: specifying the service discipline of *fcfs_sy* or specifying (or defaulting) fcfs for the service discipline and using the *synchronous* method

- **fcfs_sy** – first come, first served, clock synchronized

  This is the same as fcfs except that requests can be satisfied only at the beginning of a clock cycle. If not otherwise specified (via *synchronous* below), the clock phase (time to onset of first clock cycle) will be 0.0, and the period (length of a clock cycle) will be 1.0.

- **inf_srv** – infinite servers

  There is no queueing delay at all since there is always a server available at the facility.

- **lcfs_pr** – last come, first served, preempt

  Arriving processes are always serviced immediately, preempting a process that is currently being served if necessary. Priority is not a consideration with this service discipline.

- **prc_shr** – processor sharing

  This is load-dependent processor sharing. Service times for each process are determined based on the number of processes at the facility. If not otherwise specified (see *set_loaddep* below), it will be assumed that the rate that applies when there are *n* processes at the facility is *n* (in other words, if there are *n* processes at the facility, the service time will be multiplied by *n*). The altered service times are recomputed as tasks that arrive at and leave the

facility. There is no queueing delay with processor sharing since the assumption is that the server works faster and faster as necessary to service all processes that request it.

There can be a maximum of 100 processes sharing a *prc_shr* facility.

- **pre_res** – preempt resume

  Higher priority processes will preempt lower priority processes, so that the highest priority process at the facility will always finish using it first. Where the priorities are the same, processes will be served on a first come, first served basis. Preempted processes will eventually resume and complete their service time interval.

- **pre_rst** – preempt-restart

  Higher priority processes will preempt lower priority processes, so that the highest priority process at the facility will always finish using it first. Where the priorities are the same, processes will be served on a first come, first served basis. Preempted processes will eventually resume and will restart their respective service time intervals each time they resume.

- **rnd_pri** – round robin with priority

  Higher priority processes will be served first. When there are multiple processes with the same priority, they will be serviced on a round robin basis, with each getting the amount of time specified in *set_timeslice* (see below) before being preempted by the next process of the same priority.

- **rnd_rob** – round robin

Processes will be serviced on a round robin basis, with each getting the amount of time specified in *set_timeslice* (see below) before being preempted by the next process requiring service. Process priority is not a consideration with this service discipline.

Caution: The use statement (as opposed to the reserve) statement must be used for most of these service disciplines to be effective. Only fcfs and *fcfs_sy* will operate properly with *reserve.*

To set the clock information for the *fcfs_sy* service discipline:

Prototype: `void facility::synchronous(double phase,, double period)`

Dynamic Example: `fd->synchronous (0.0, 1.0);`

To set the load dependent service rate for the *prc_shr* (see above) service discipline:

Prototype: `void facility::set_loaddep(double rate[], long n)`

Dynamic Example: `fd->set_loaddep(rate, 10);`

The "rate" array is an array of length *n*, where each element specifies the service rate for the corresponding number of processes using the server. *Rate[i]* is the amount by which the service time is multiplied when there are *i* processes at the facility. If *n* is less than the 100 (the maximum number of processes allowed to share use of a *prc_shr* facility), then the value of the last specified rate is replicated until 100 values are available. Also, if *n* is greater than 99, only 100 values will be used. It should be remembered that the altered service times are recomputed as tasks arrive at and leave the facility.

To set the time slice for the round robin service disciplines, *rnd_pri* and *rnd_rob* (see above):

Prototype: `void facility::set_timeslice(double slice_length)`

Dynamic Example: `fd->set_timeslice(0.01);`

4.12 Deleting a Facility or a Facility Set

To delete a dynamic facility:

Example: `delete fd;`

To delete a dynamic facility set:

Example: `delete f_set;`

Caution: Deleting a facility or facility set is an extreme action and should be done only when necessary.

4.13 Collecting Class-Related Statistics

Information about usage of a facility by processes belonging to different process classes can be collected for all facilities or for a specific facility. To collect class-based usage information for a specific facility:

Prototype: `void facility::collect_class`

Dynamic Example: `fd->collect_class();`

Usage of this facility by all process classes (see section 17, ”Process Classes”) will be reported in the facilities report. Also, it is an error to change the maximum number of classes allowed after this statement has been executed.

To collect usage information for all facilities, use this function:

Prototype: `void collect_class_facility_all(void)`

Example: `collect_class_facility_all();`

The function collects class usage information for all facilities that are in existence when this statement is executed. Usage of the facilities by all process classes (see section 17, ”Process Classes”) will be reported in the facilities report. It is an error to change the maximum number of classes allowed after this statement has been executed.

4.14 Inspector Methods

All statistics and information maintained by a facility can be retrieved during execution of a model or upon its completion.

Prototype:	Functional Value:
`char *facility::name()`	pointer to name of facility
`long facility::num_servers()`	number of servers at facility
`char *facility::service_disp()`	pointer to name of service discipline at facility
`double facility::timeslice()`	time in each time-slice for facility (which has a round robin service discipline)
`long facility::num_busy()`	number of servers currently busy at facility
`long facility::qlength()`	number of processes currently waiting at facility
`long facility::status()`	current status of facility BUSY if all servers are in use

	FREE if at least one server is not in use
`long facility::completions()`	number of completions at facility
`long facility::preempts()`	number of preempted requests at facility
`double facility::qlen()`	mean queue length at facility
`double facility::resp()`	mean response time at facility
`double facility::serv()`	mean service time at facility
`double facility::tput()`	mean throughput rate at facility
`double facility::util()`	utilization (% of time busy) at facility

Additional data on servers and classes can be obtained as follows:

`long facility::server_completions(long sn)`
: number of completions for server *sn* at facility

`double facility::server_serv(long sn)`
: mean service time for server *sn* at facility

`double facility::server_tput(long sn)`
: mean throughput rate for server *sn* at facility

`double facility::server_util(long sn)`
: utilization for server *sn* at facility

`double facility::server_serv_time_remaining(long sn)`
: time remaining on service interval server sn

`long facility::class_qlength(CLASS c)`
: number of process from class at facility

`long facility::class_completions(CLASS c)`
: number of completions for class at facility

`double facility::class_qlen(CLASS c)`
: mean queue length for class at facility

`double facility::class_resp(CLASS c)`
: mean response time for class at facility

`double facility::class_serv(CLASS c)`
: mean service time for class at facility

`double facility::class_tput(CLASS c)`
: mean throughput rate for class at facility

`double facility::class_util(CLASS c)`
: utilization for class at facility

4.15 Status Report

To obtain a report on the status of all of the facilities in a model:

Prototype: `void status_facilities (void)`

Example: `status_facilities ();`

This report lists each facility along with the number of servers, the number of servers which are busy, the number of processes waiting, the name and id of each process at a server, and the name and id of each process in the queue.

5 Storages

A CSIM storage is a resource which can be partially allocated to a requesting process. A storage consists of a counter (to indicate the amount of available storage) and a queue for processes waiting to receive their requested allocation. A storage set is an array of these basic storages.

A storage can be designated to be synchronous. In a synchronous storage, each allocate is delayed until the onset of the next clock cycle.

Usage and queueing statistics are automatically maintained for each storage unit. These are "printed" whenever a *report* or a *report_storages* statement is executed (see section 19.2, “CSIM Report Output,” for details about the reports that are generated).

5.1 Declaring and Initializing Storage

A pointer to a dynamic storage object is as follows:

Dynamic Example: `storage *sd;`

Before a storage can be used, the constructor must be invoked:

Prototype: `storage::storage(char *name, long size)`

Static Example: `storage ss("mem", 1000);`

Dynamic Example: `sd = new storage(“mem”, 1000);`

A newly created storage is created with all of the “storage” available. Storages should be declared with global variables in the sim (main) process, prior to the beginning of the simulation part of the model. A dynamic storage must be initialized via the *new storage* statement before it can be used in any other statement.

5.2 Allocating from a Storage

The elements of a storage can be allocated to a requesting process.

Prototype: `void storage::allocate(long amount)`

Dynamic Example: `sd->allocate(10);`

The amount of storage requested is compared with the amount of storage available at *sd.* If the amount of available storage is sufficient, the amount available is decreased by the requested amount and the requesting process continues. If the amount of available storage is not sufficient, the requesting process is suspended. When some of the storage elements are deallocated by some other process, the highest priority waiting processes are automatically allocated their requested storage amounts (as they can be accommodated), and they are allowed to continue. The list of waiting processes is searched in priority order until a request cannot be satisfied. In order to preserve priority order, a new request which would fit but which would get in front of higher priority waiting requests will be queued.

Note: the *alloc* method is equivalent to *allocate:*

Prototype: `void storage::alloc(long amount)`

Dynamic Example: `sd->alloc(10);`

5.3 Deallocating from a Storage Unit

To return storage elements to a storage, the deallocate procedure is used.

Prototype: `void storage::deallocate(long amount)`

Dynamic Example: `sd->deallocate(10);`

If there are processes waiting, the highest priority processes that are waiting are examined. Those that will now fit have their requests satisfied and are allowed to continue. Executing a deallocate statement causes no simulated time to pass.

Caution: There is no check to insure that a process returns only the amount of storage that it had been previously allocated.

Note: the *dealloc* method is equivalent to the *deallocate* method:

Prototype: `void storage::dealloc(long amount)`

Dynamic Example: `sd->dealloc(10);`

5.4 Producing Reports

Reports for storages are most often produced by calling the *report* function, which reports for all CSIM objects. Reports can be produced for all existing storages by calling the *report_storages* function. The report for a storage, as illustrated below, gives the name of the storage, the size (initial amount), the average allocation request, the utilization, the average time each request is "in" the storage, the average queue length, the average response time and the number of completed requests.

STORAGE SUMMARY

storage name	size	alloc amount	alloc count	dealloc amount	dealloc count	util	in-que length	in-queue time
store	120	33.6	100335	30.0	100333	0.751	7.94470	7.91818

5.5 Resetting a Storage Unit

In some cases, it is necessary to reset the statistics counters for a specific storage unit.

Prototype: `void storage::reset(void)`

Example: `s->reset();`

Executing this statement does not affect the state of the storage. The “reset” and the reset_storages statements each call the storage::reset () statement for all storage units in the model.

5.6 Storage Sets

A storage set is an array of storages. Each element of the array is an individual storage.

Dynamic Example: `storage set *d_set`

A storage set must be constructed before the elements of the set can be used.

Prototype:
```
void storage_set::storage_set(char
  *name,long amount, long number_in_set);
```

Static Example: `storage_set s_set(“set”, 100, 5);`

Dynamic Example: `d_set = new storage_set(“set”, 100, 5);`

The example initializes a set of five storages, each with 100 elements of storage available at the onset of operation. The name is the name of the set. Each individual unit of storage is given a unique (indexed) name. In the example, the first storage in the set is named ”set[0]”, the second is named “set[1]”, and so on. The last storage is named “set[99]”. Similarly, the individual units of storage are accessed as elements of an array. All of the operations which apply to a storage also apply to the individual units of a storage set.

Accessing individual elements of a storage set are accessed as shown in these examples:

Static Example: `s.set[i].allocate(10);`

Dynamic Example: `(*sd)[i])allocate(10);`

5.7 Allocating Storage with a Time-out

Sometimes, processes cannot wait indefinitely to allocate the needed amount of storage. If such a process executes the *timed_allocate* function, then, if the requested amount of storage is not available, the process will be suspended until either the requested amount of storage becomes available or the time-out interval expires.

Prototype:
```
long storage::timed_allocate(long
      amount, double timeout)
```

Dynamic Example:
```
result = sd->timed_allocate(10,100.0);
   if(result ! = TIMED_OUT) . . .
```

The process must check the function value *(result)* to determine whether or not the requested storage was obtained. If the value TIME_OUT is returned, the process did not obtain any of the requested storage. If this value is not returned

(EVENT_OCCURRED will in fact be returned), then the process did obtain the requested storage.

5.8 Making a Storage Unit Clock Synchronous

A storage unit can be designated to be a synchronous storage unit.

Prototype: `void storage::synchronous(double.phase, double period)`

Dynamic Example: `s->synchronous(0.0, 1.0);`

A synchronous storage unit is similar to a normal storage unit except that allocation requests are always delayed until the beginning of the next clock cycle. The clock phase specifies the interval before the onset of the first clock cycle, and the period specifies the interval between successive clock cycles.

5.9 Adding and Removing Storage Elements to a Storage

To increase the amount of storage (the number of storage elements) in a storage:

Prototype: `void storage::add_store(long amount)`

Dynamic Example: `sd->add_store(100);`

Note: The add method does not check the queue of waiting processes to see if a process could allocate some storage after some storage is added.

To decrease the amount of storage (the number of storage elements) in a storage:

Prototype: `void storage::remove_store(long amount)`

Dynamic Example: `sd->remove_store(100);`

5.10 Renaming a Storage Unit:

The name of a storage can be changed at any time, as follows:

Prototype: `void storage::set_name(char *new_name)`

Dynamic Example: `sd->set_name("cache");`

Only the first ten characters of the storage's name are stored.

5.11 Deleting Storage or a Storage Set

To delete a dynamic storage:

Example: `delete s;`

To delete a dynamic storage set:

Example: `delete s_set`

Deleting a storage or storage set is an extreme action and should be done only when necessary.

5.12 Collecting Class-Related Statistics

Information about usage of a storage unit by processes belonging to different process classes can be collected for all storage units or for a specific storage unit. To collect class-based usage information for a specific storage, use the following function:

Prototype: `void storage::collect_class_storage()`

Example: `s->collect_class_storage();`

Usage of this storage unit by all process classes (see section 17, ”Process Classes”) will be reported in the storages report. Also, it is an error to change the maximum number of classes allowed after this statement has been executed.

To collect usage information for all storages, use the following function:

Prototype: `void collect_class_storage_all(void)`

Example: `collect_class_storage_all();`

The function collects class usage information for all storage units that are in existence when this statement is executed. Usage of the storages by all process classes (see section 17, ”Process Classes”) will be reported in the storages report. It is an error to change the maximum number of classes allowed after this statement has been executed.

5.13 Inspector Methods

These functions each return a statistic which describes some aspect of the usage of the specified storage.

5. Storages

Prototype:	Functional Value:
`char *storage::name()`	pointer to name of store
`long storage::capacity()`	number of storages defined for store
`long storage::avail()`	number of storages currently available at store
`long storage::qlength()`	number of processes currently waiting at store
`double storage::request_total()`	sum of requested amounts from store
`double storage::release_total()`	sum of released amounts from store
`long storage::number_amt()`	time-weighted sum of requesters for store
`double storage::busy_amt()`	busy time-weighted sum of amounts for store
`double storage::waiting_amt()`	waiting time weighted sum of amounts for store
`long storage::request_cnt()`	total number of requests for store
`long storage::release_cnt()`	total number of completed requests for store
`long storage::queue_cnt()`	number of queued requests at store
`double storage::time()`	time at store that is spanned by report

Additional data on classes can be obtained as follows:

`double storage_class_busy_amt(STORE s, CLASS c)`
busy time-weighted sum of amounts for class at store

```
double storage_class_waiting_amt(STORE s, CLASS c)
```
waiting time-weighted sum of amounts for class at store

```
long storage_class_request_cnt(STORE s, CLASS c)
```
total number of requests for class at store

```
long storage_class_release_cnt(STORE s, CLASS c)
```
total number of completed requests for class at store

```
double storage_class_request_total(STORE s, CLASS c)
```
sum of requested amounts for class from store

```
double storage_class_release_total(STORE s, CLASS c)
```
sum of released amounts for class from store

5.14 Reporting Storage Status

Prototype: `void status_storages(void)`

Example: `status_storages();`

The report will be written to the default output location or to that specified by *set_output_file* (see section 21.6, “Output File Selection”).

5. Storages

6 Buffers

A CSIM buffer is a resource which can store (hold) a number of tokens. The primary operations for a buffer are *put*, which places a number of tokens into the buffer, and *put*, which removes a number of tokens from the buffer. A buffer has a maximum capacity for holding tokens. A *get* operation stalls if there are too few tokens in the buffer, and a *put* operation stalls if there is not enough space (unused capacity) in the buffer.

A buffer consists of a counter (indicating the number of tokens in the buffer), and two queues: a put-queue, for processes waiting to complete a put operation and a get-queue, for processes waiting to complete a get operation.

Usage and queueing statistics are automatically maintained for each buffer. These are “printed” whenever a *report* or *report_buffers* statement is executed (see section 19.2, “CSIM Report Output,” for details about the reports that are generated).

6.1 Declaring and Initializing a Buffer

A pointer to a dynamic buffer object is as follows.

Dynamic Example: `buffer *b;`

Before a buffer can be used, it must be initialized by calling the buffer function.

Prototype: `buffer::buffer(char *name, long size);`

Static Example: `buffer bs("bs", 10);`

Dynamic Example: `b = new buffer("b", 10);`

A newly created buffer is empty. Buffers should be declared with global variables in the sim (main) process, prior to beginning the simulation part of the model. A dynamic buffer must be initialized via the *new buffer* statement before it can be used in any other statement.

6.2 Putting Tokens into a Buffer

Tokens can be added to a buffer using the *put* operation.

Prototype: `void buffer::put(long amt);`

Dynamic Example: `b->put(5);`

The number of tokens being put (the amount) is compared with the space remaining in the buffer (the maximum size minus the current amount). If the available space is less than or equal to space remaining, the amount of the put is added to the current

amount and the process doing the buffer_put continues. If the amount of the put exceeds the space remaining, the process is placed in the put-queue and is then suspended. When some other process (or processes) removes (gets) tokens, the highest priority process in the put-queue is checked; if its put request can be accommodated, the buffer_put is done and the process resumes at the statement following the buffer_put statement. If other processes in the put-queue can be accommodated, they too are processed and allowed to proceed.

6.3 Getting Tokens from a Buffer

Tokens can be removed a buffer using the buffer-get statement.

Prototype: `void buffer::get(long amt);`

Dynamic Example: `b->get(4);`

The number of tokens being requested in a get (the amount) is compared to the number in the buffer. If the amount is less than or equal to the number in the buffer, the amount is subtracted and the process doing the buffer_get proceeds. If not, the process doing the buffer_get is placed in the get-queue and then suspended. When another process (or processes) puts (adds) tokens to the buffer, the highest priority process in the get-queue is checked; if its get request can be satisfied, the buffer_get is done, and the process resumes at the statement following the buffer_get. If other processes in the get-queue can be accommodated, they too are processed and allowed to proceed.

6.4 Producing Reports

Reports for buffers are most often produced by calling the *report* function which reports for all CSIM objects. Reports can be produced for all existing buffers by calling *report_buffers* function. The report for a buffer gives the name of the buffer,

BUFFER SUMMARY

buffer name	size	get amt	get qlen	get resp	get count	put amt	put qlen	put resp	put count
buff	20	2.4	0.00000	0.00000	32	3.5	1.54545	0.60714	28

6.5 Resetting a Buffer

In some cases, it is necessary to reset the statistics counters for a specificbuffer.

Prototype: `void buffer:: reset(void)`

Example: `b->reset();`

Executing this statement does not affect the state of the buffer or its servers. The "reset" and the "reset_buffers" statements each call buffer::reset () for all buffers in the model.

6.6 Timed Operations for Buffers

Sometimes, processes cannot wait indefinitely to either get tokens form or putting tokens into a buffer.

Prototype:
```
long buffer::timed_get(long amt,
      double timeout);
```

Dynamic Example:
```
result = b->timed_get(5, 100.0);
       if(result != TIMED_OUT) . . .
```

and

Prototype:
```
long buffer::timed_put(long amt,
      double timeout);
```

Dynaimc Example:
```
result = b->timed_put(5, 100.0);
       if(result != TIMED_OUT) . . .
```

The process must check the function value *(result)* to determine whether or not the operation time-out or completed. If the value TIME_OUT is returned, the process did not get or put the amount. If this value is not returned (EVENT_OCCURRED will in fact be returned), then the process did complete successfully.

6.7 Adding Capacity to a Buffer

To increase the space (capacity) for a buffer,

Prototype:
```
void buffer::add_space(long amt);
```

Dynamic Example:
```
b->add_space(25);
```

6.8 Removing Capacity from a Buffer

To remove space (capacity) from a buffer:

Prototype: `void buffer::remove_space(long amt);`

Dynamic Example: `b->remove_space(2);`

6.9 Deleting a Buffer

To delete a buffer:

Dynamic Example: `delete b`

Deleting a buffer is an extreme action and should be done only when necessary.

6.10 Collecting Class-Related Statistics

Information about usage of a buffer by processes belonging to different process classes can be collected for all buffers or for a specific buffer. To collect class-based usage information for a specific buffer, use the following function:

Prototype: `void buffer::collect_class_buffer()`

Example: `b->collect_class_buffer();`

Usage of this buffer by all process classes (see section 17, ”Process Classes”) will be reported in the buffers report. Also, it is an error to change the maximum number of classes allowed after this statement has been executed.

To collect usage information for all buffers, use the following function:

Prototype: `void collect_class_ buffer _all(void)`

Example: `collect_class_ buffer _all();`

The function collects class usage information for all buffers that are in existence when this statement is executed. Usage of the buffers by all process classes (see section 17, ”Process Classes”) will be reported in the buffers report. It is an error to change the maximum number of classes allowed after this statement has been executed.

6.11 Inspector Functions

These functions each return a statistic or counter value which describes some aspect of the operation of a buffer:

Prototype:	Function Value:
`long buffer::current()`	current number of tokens
`long buffer::size()`	capacity of buffer
`long buffer::get_total()`	total amount retrieved
`long buffer::put_total()`	total amount put
`long buffer::get_count()`	number of get’s
`long buffer::put_count()`	number of put’s

`double buffer::get_timeQueue()` sum of get-queue lengths

`double buffer::put_timeQueue()` sum of get-queue lengths

`char* buffer::name()` name of buffer

`long buffer::get_current_count()` current get-queue length

`long buffer::put_current_count()` current put-queue length

Additional data on classes can be obtained as follows:

`long buffer_class_get_total(BUFFER b, CLASS c)`
total amount retrieved for a class

`long buffer_class_put_total(BUFFER b, CLASS c)`
total amout put for a class

`long buffer_class_get_count(BUFFER b, CLASS c)`
number of gets for a class

`long buffer_class_put_count(BUFFER b, CLASS c)`
number of puts for a class

`double buffer_class_get_timeQueue(BUFFER b, CLASS c)`
sum of get-queue lengths for a class

`double buffer_class_put_timeQueue(BUFFER b, CLASS c)`
sum of put-queue lengths for a class

6.12 Reporting Buffer Status

Prototype: `void buffer_status(void);`

Example: `status_buffers();`

The buffer status report will be written to the default output location or to the location specified by *set_output_file* (see section 21.6, “Output File Selection”).

7 Events

Events are used to synchronize the operations of CSIM processes. An event exists in one of two states: *occurred or not occurred*. A process can change the state of an event or it can suspend its execution until an event has occurred. When a process is suspended it can join a set of processes, all of which will be resumed when the event occurs. Or, it can join an ordered queue from which only one process is resumed for each occurrence of the event. An event is automatically reset to the *not occurred* state when all of the suspended processes that can proceed have done so.

Advanced features of events include the ability to create sets of events for which processes can wait and the ability for a process to bound its waiting time by specifying a time-out. Events can also be used to construct other synchronization mechanisms such as semaphores.

7.1 Declaring and Initializing an Event

A pointer to a dynamic event is declared in a CSIM program as follows:

Dynamic Example:
```
event *ed;
```

Before a dyanmic event can be used, it must be initialized by invoking the *new event* constructor.

Prototype: `event:::event(char *name)`

Static Example: `event es(“done”);`

Dynamic Example: `ed = new event(“done”);`

An event is initialized in the *not occurred* state. The event name is used only to identify the event in output reports and trace messages.

An event that is initialized in the first CSIM process (sim) exists during the entire simulation run and is called a global event. An event initialized in any other process is called a local event. A local event is deleted when the process in which it was initialized terminates. To make such an event exist for the entire run, it must be initialized by calling the *global_event* function.

Prototype: `global event::global_event (char *name)`

Static Example: `global_event es(“done”);`

Dynamic Example: `ed = new global_event(“done”);`

7.2 Waiting for an Event to Occur

A process waits for an event to occur by calling the *wait* function.

Prototype: `void event::wait(void)`

Dynamic Example: `e->wait();`

If the event is in the *occurred* state, control returns from the *wait* function immediately and the event is changed to the *not occurred* state. If the event is in the *not occurred* state, the calling process is suspended from further execution and control will not return from the *wait* function until some other process sets this event.

When the event is set, all waiting processes will be resumed and the event will be placed in the *not occurred* state.

7.3 Waiting with a Time-Out

Sometimes a process must not be suspended indefinitely waiting for an event to occur. If a process calls the *timed_wait* function, it will be suspended until either the event is set or the specified amount of time has passed.

Prototype: `long event::timed_wait(double timeout)`

Dynamic Example:
```
result = ed->timed_wait(100.0);
  if (result ! = TIMED_OUT)
```

The calling process should check the functional value to determine the circumstances under which it was resumed. If the value EVENT_OCCURRED is returned, the process was activated because the event has occurred; if the value TIMED_OUT is returned, the specified amount of time passed without the event being set.

7.4 Queueing for an Event to Occur

A process joins the ordered queue for an event by calling the *queue* function.

Prototype: `void event::queue(void)`

Dynamic Example:
```
ed->queue();
```

This function behaves similarly to the *wait* function, except that each time the event is set only one queued process is resumed. The queue is maintained in order of process priority, with processes having the same priority being ordered by time of insertion into the queue.

7.5 Queueing with a Time-out

If a process calls the *timed_queue* function, it will be suspended until either the event is set a sufficient number of times for the process to be activated or the specified amount of time has passed.

Prototype: `long event::timed_queue(double timeout)`

Dynamic Example:
```
result = ed->timed_queue(100.0);
  if (result ! = TIMED_OUT) ...
```

The calling process should check the functional value to determine the circumstances under which it was resumed. If the value EVENT_OCCURRED is returned, the process was activated because the event occurred; if the value

TIMED_OUT is returned, the specified amount of time passed without the process being activated by the event being set.

7.6 Setting an Event

A process can put an event into the *occurred* state by calling the *set* function.

Prototype: `void event::set(void)`

Dynamic Example: `ed->set();`

Calling this function causes all waiting processes and one queued process to be resumed. If there are no waiting or queued processes, the event will be in the *occurred* state upon return from the *set* function. If there are waiting or queued processes, the event will be in the *not occurred* state upon return. No simulation time passes during these activities. Setting an event that is already in the *occurred* state has no effect.

7.7 Clearing an Event

A process can put an event into the *not occurred* state by calling the *clear* function.

Prototype: `void event::clear(void)`

Dynamic Example: `ed->clear();`

Clearing an event happens in zero simulation time and no processes are in any way affected. Clearing an event that is already in the *not occurred* state has no effect.

7.8 Renaming an Event

The name of an event can be changed at any time using the *set_name_event* function.

Prototype: `void event::set_name(char *new_name)`

Dynamic Example: `es->set_name_event("finished");`

Only the first ten characters of the event's name are stored.

7.9 Deleting an Event

When a dynamic event is no longer needed, its storage can be reclaimed using the *delete* function.

Dynamic Example: `delete ed;`

If an event is local, only the process that created the event can delete it. Once an event has been deleted, it must not be further referenced. It is an error to attempt to delete an event on which processes are waiting or queued.

7.10 Collecting and Reporting Statistics for Events

A set of statistics on the usage can be collected for specified events. Statistics collection for an event is initiated by executing the monitor() method.

Prototype: `void event::monitor(void )`

Dynamic Example: `e->monitor();`

The standard report function automatically proceeds to “print” a report for each monitored event. This report is as follows:

EVENT SUMMARY

event name	number of queue vst	avg que length	avg time queued	number of wait vsts	avg wait length	avg time waiting	number of set ops
ev	10	0.99010	1.00000	50	4.95050	1.00000	10

A separate for all of the monitored events is produced using the report_events() procedure, as follows:

Prototype: `void report_events(void )`

Example: `report_events();`

All of the events in an event_set (see below) can be monitored as well.

Prototype: `void event_set::monitor(void)`

Dynamic Example: `es->monitor();`

7.11 Resetting an Event

In some cases, it is necessary to reset the statistics counters for a specific event.

Prototype: `void event::reset(void)`

Example: `e->reset();`

Executing this statement does not affect the state of the event. The “reset” and the reset_events statements each call reset_event() for all events in the model.

7.12 Event Sets

An event set is an array of related events for which some special operations are provided. A pointer to a dynamic event set is declared as follows:

Example: `event_set *ed_set;`

All events in an event set are initialized by invoking the *event_set* constructor.

Prototype: `event_set::event_set(char *name, long number_of_events)`

Static Example: `event_set es_set("events", 10);`

Dynamic Example: `ed_set = new event_set("events",10);`

As with any C array, the events in an event set are indexed from *0* to *num_events - 1*. Individual events in the event set can be manipulated using any of normal event functions (*e.g.*, set, clear, wait, queue).

Static Example: `es_set[3].set();`

Dynamic Example: `(*ed_set).[3].set();`

A process can wait for the occurrence of any event in an event set by calling the *wait_any* function.

Prototype: `long event_set::wait_any(void)`

Dynamic Example: `event_index = ed_set->wait_any();`

This function returns the index of the event that caused the calling process to proceed. If multiple events in the set are in the *occurred* state, the lowest numbered event is the one recognized by the calling process. All processes that have called *wait_any* are activated by the next event that occurs, and these processes all receive the same index value.

A process can join an ordered queue for an event set by calling the *queue_any* method.

Prototype: `long event_set::queue_any(void)`

Dynamic Example: `event_index = ed_set->queue_any();`

Each time any event in the event set occurs, one process in the queue is activated. The functional value is the same as that of the *wait_any* function. It is not currently possible to specify a time-out for the *wait_any* or *queue_any* methods.

An entire dynamic event set is deleted as follows:

Example: `delete e_set;`

It is an error to delete individual elements of an event set

A process can also do a timed_wait_any() operation at an event_set.

Prototype: `long event_set::timed_wait_any(double time_out)`

Dynamic Example: `result = ed_set->timed_wait_any(double time_out);`

If the result is not equal to TIME_OUT, then it is index of the event which caused the process to continue.

A process can also do a timed_queue_any() operation at an event_set:

Prototype: `long event_set::timed_queue_any(double time_out)`

Dynamic Example: `result = ed_set->timed_queue_any(double time_out);`

If the result is not equal to TIMED_OUT, then it is the index of the event which caused the process to continue.

7.13 Inspector Methods

The following methods return information about the specified event at the time they are called.

Prototype:	Functional value:
char *event::name()	pointer to name of event
long event::wait_cnt()	number of processes waiting for event
long event::queue_cnt()	number of processes queued of event
long event::qlen()	sum of wait_cnt and queue_cnt
long event::state()	state of event: OCC if occurred or NOT_OCC if not occurred
long event::set_count()	number of set operations
double event::queue_sum()	sum of queue queue lengths
double event::wait_sum()	sum of wait queue lengths
long event::queue_delay_count()	number of queue delays
long event::wait_delay_count()	number of wait delays
long event::queue_length()	average queue queue length
long event::wait_length()	average wait queue length
long event::queue_time()	average queue queue time
long event::wait_time()	average wait queue time
long event::queue_count()	average queue queue count
long event::wait_count()	average wait queue count
long event_set::num_events()	number of events in the event set

7.14 Status Report

The *status_events* function prints a report of the status of all events in the model.

Prototype: `void status_events(void)`

Example: `status_events();`

For each event, the report includes its state, the number of processes waiting for it, the number of processes queued for it, the name and id of all waiting processes, and the name and id of all queued processes. The report is written to the default output stream or the stream specified in the last call to *set_output_file*.

7.15 Built-In Events

A process can suspend itself until there are no other active processes by waiting on the built-in event *event_list_empty*.

Example: `event_list_empty.wait();`

This event is automatically set by CSIM when all processes have terminated or are waiting for something (*e.g.*, a facility or storage). Modelers sometimes use this to force the initial (sim) process to wait until all work in the system being modeled has completed. Upon being reactivated, the initial process might then produce reports.

If run length control is involved for a table, qtable, meter or box, (see 14.3), a process can suspend itself until the run length control mechanism signals the end of a run. This is done by waiting for the built-in event *converged*.

Example: `converged.wait()`

7. Events

8 Mailboxes

A mailbox allows for the synchronous exchange of data between CSIM processes. Any process may send a message to any mailbox, and any process may attempt to receive a message from any mailbox.

A mailbox is comprised of two FIFO queues: a queue of unreceived messages and a queue of waiting processes. At least one of the queues will be empty at any time. When a process sends a message, the message is given to a waiting process (if one exists) or it is placed in the message queue. When a process attempts to receive a message, it is either given a message from the message queue (if one exists) or it is added to the queue of waiting processes.

A message can be either a single long integer or a pointer to some other data object. If a process sends a pointer, it is the responsibility of that process to maintain the integrity of the referenced data until it is received and processed.

8.1 Declaring and Initializing a Mailbox

A pointer to a dynamic mailbox is declared in a CSIM program as follows:

Dynamic Example:
```
mailbox *md;
```

Before a dynamic mailbox can be used, it must be initialized by invoking the *new mailbox* constructor.

Prototype: `mailbox::mailbox(char*name)`

Static Example:
```
mailbox ms("request");
```

Dynamic Example:
```
md = new mailbox("requests");
```

A newly created mailbox contains no messages. The mailbox name is used only to identify the mailbox in output reports and trace messages.

A mailbox that is initialized in the first CSIM process (sim) exists during the entire simulation run and is called a global mailbox. A mailbox initialized in any other process is called a local mailbox. A local mailbox is deleted when the process in which it was constructed terminates.

8.2 Sending a Message

A process sends a message by calling the *send* function.

Prototype: `void mailbox::send(long message)`

Dynamic Example:
```
md->send((long) buffer);
```

If one or more processes are waiting on this mailbox, the process at the head of the process queue will resume execution and will be given this message. If no processes are waiting, this message will be appended to the tail of the message queue. No simulation time passes during this method call.

8.3 Receiving a Message

A process receives a message by calling the *receive* function.

Prototype: `void mailbox::receive(long* message)`

Dynamic Example: `md->receive((long*) &ptr);`

If one or more messages are queued at this mailbox, the calling process is given the message at the head of the queue and continues executing. If no messages are queued, the process is suspended from further execution and is added to the tail of the process queue for this mailbox.

8.4 Receiving a Message with a Time-out

Sometimes a process must not wait indefinitely to receive a message. If a process calls the *timed_receive* function, it will be suspended until either a message is received or the specified amount of time has passed.

Prototype:
```
long mailbox::timed_receive(long*
 message, double timeout)
```

Example:
```
result = md->timed_receive((long*)&ptr,
                           100.0);
if (result ! = TIMED_OUT) ...
```

The calling process can check the functional value to determine the circumstances under which it was resumed. If the value EVENT_OCCURRED is returned, the process was activated because a message was received; if the value TIMED_OUT is returned, the specified amount of time passed without the process being activated by the receipt of a message.

8.5 Synchronous_send

A process can also send a message with a synchronous send operation (as opposed to a normal send operation as described above). With a synchronous send, the send process waits until the sent message has been received by another process.

Prototype: `void mailbox::synchronous_send(long message)`

Dynamic Example: `mb->synchronous_send(msg,);`

8.6 Synchronous_send with a Time-out

A process can also do a synchronous_send operation with a time-out interval:

Prototype:
```
long mailbox::timed_synchronous_send(long
                    message, double timeout)
```

Dynamic Example:
```
result = mb->timed_synchronous_send(
                msg, 100.0);
if (result ! = TIMED_OUT)
```

The calling process can check the function value to determine the circumstances under which it was resumed. If the value EVENT_OCCURRED is returned, the process was activated because a message was received; if the value TIMED_OUT is returned, the specified amount of time passed without the process being activated by the receipt of a message by another process.

8.7 Collecting and Reporting Statistics for Mailboxes

A set of statistics on the usage can be collected for specified mailboxes. Statistics collection for a mailbox is initiated by executing the mailbox_monitor statement.

Prototype: `void mailbox::monitor()`

Dynamic Example: `mb->monitor();`

The standard report function automatically proceeds to “print” a report for each monitored mailbox. This report is as follows:

MAILBOX SUMMARY

mailbox name	number of proc visits	process qlength	process rspTime	number of messages	message qlength	message rspTime
mb	100	0.18737	0.32295	100	1.03091	1.79461

A separate report for all monitored mailboxes is produced by the report_mailboxes() procedure:

Prototype: `void report_mailboxes()`

Example: `report_mailboxes();`

8.8 Resetting a Mailbox

In some cases, it is necessary to reset the statistics counters for a specific mailbox.

Prototype: `void mailbox::reset()`

Example: `mb->reset();`

Executing this statement does not affect the state of the mailbox. The “reset()” and the reset_mailboxes statements each call the reset() method for all mailboxes in the model.

8.9 Mailbox Sets

A mailbox set is an array of related mailboxes for which some special operations are provided. A mailbox set is declared using the C array construct.

Example: `mailbox_set *mb_set;`

All mailboxes in a mailbox set are initialized with a single call to the *mailbox_set* function.

Prototype: `mailbox_set(char *name, long number_of_mailboxes)`

Dynamic Example: `mb_set = new mailbox_set( “mboxes”, 10);`

As with any C++ array, the mailboxes in a mailbox set are indexed from *0* to *num_mailboxes - 1*. Individual mailboxes in the mailbox set can be manipulated using any of normal mailbox functions (*e.g.*, send, and receive).

Dynamic Example: `(*mb_set).[3].send(1);`

A process can wait for the occurrence of any mailbox in an mailbox set by calling the *recieve_any* function.

Prototype: `long mailbox_set::receive_any((long*) msg)`

Example: `mb_index = receive_any(mb_set, &m);`

This function returns the index of the mailbox that caused the calling process to proceed. If multiple mailboxes in the set have received messages, the lowest numbered mailbox is the one recognized by the calling process.

A process can also do a timed_receive_any() operation at an mailbox_set.

Prototype:
```
long mailbox_set::timed_receive_any((long*)&ptr,
double time_out)
```

Example:
```
result = mb_set->timed_receive_any(&m, 10.0)
if(result != TIMED_OUT) . . .
```

If the result is not equal to TIMED_OUT, then it is the index of the mailbox which received the message and caused the process to continue.

All of the events in a mailbox_set can be monitored as well.

Prototype: `void mailbox_set::monitor()`

Example: `mbs->monitor();`

8.10 Renaming a Mailbox

The name of a mailbox can be changed at any time using the *set_name_mailbox* function.

Prototype: `void mailbox::set_name(char *new_name)`

Static Example: `mb.set_name("responses");`

Only the first ten characters of the mailbox's name are stored.

8.11 Deleting a Mailbox

When a dynamic mailbox is no longer needed, its storage can be reclaimed as follows:

Example: `delete md;`

If a mailbox is local, only the process that created the mailbox can delete it. Once a mailbox has been deleted, it must not be further referenced. Deleting a mailbox causes any unreceived messages to be lost. It is an error to attempt to delete a mailbox on which processes are waiting.

8.12 Inspector Methods

The following methods return information about the specified mailbox at the time they are called.

Prototype:	Functional value:
`char *mailbox::name()`	pointer to name of mailbox
`long mailbox::msg_cnt()`	if positive, number of unreceived messages if negative, magnitude is number of waiting processes

`long mailbox::queue_cnt()` number of processes queued at mailbox

`long mailbox_set::num_msgs()` number of messages in a mailbox set

`double mailbox::proc_sum()` sum of process queue lengths

`long mailbox::proc_delay_count()` number of processes delayed

`long mailbox::proc_count()` number of process receiving messages

`double mailbox::proc_length()` average process queue length

`double mailbox::proc_time()` average process delay time

`double mailbox::msg_sum()` sum of message queue lengths

`long mailbox::msg_delay_count()` number of messages delayed

`long mailbox::msg_count()` number of messages sent

`double mailbox::msg_length()` average message queue length

`double mailbox::msg_time()` average message delay time

8.13 Status Report

The *status_mailboxes* function prints a report of the status of all mailboxes in the model.

Prototype: `void status_mailboxes(void)`

Example: `status_mailboxes();`

For each mailbox, the report includes the number of unreceived messages, the number of waiting processes, and the name and id of all waiting processes. The report is written to the default output stream or the stream specified in the last call to *set_output_file.*

9 Managing Queues

Each of the CSIM objects (facilities, storages, etc.) consists of one or more queues (for suspended processes) and some other structure (servers in a facility, a list of messages in a mailbox, etc.). In some models, it is necessary to be able to manipulate the processes in one of these queues. The queue management features are used to accomplish this.

9.1 Process Pointers and Process Structures

The class for manipulating processes is *process*: A typedef defines a pointer to an instance of a process class.

Example: `process *pPtr;`

Example: `process_t pPtr;`

This is the object returned when a process is requested from a CSIM object.

A process pointer can be used to query the state of a process and to change attributes of a process.

Prototype: `long process::priority(void)`

Example: `priority = pptr->priority();`

Prototype: `void process::set_priorty(long prty)`

Example: `pptr->set_priority(5);`

Prototype: `long process::identity(void)`

Example: `id = pptr->identity();`

Prototype: `char* process::name(void)`

Example: `nm = pptr->name();`

Prototype: `process_class* process::this_process_class()`

Example: `c = pptr->this_pprocess_class()`

Prototype: `process_class* process_process_class(process *pptr)`

Example: `c = process_process_class(pptr)`

Note: Process classes are described in Section 17.

A process can retrieve its own process class.

Prototype: `process_class* current_process_class()`

Example: `c = current_process_class()`

A process can retrieve the next process from a list of processes

Prototype: `process_t process::next(void)`

Example: `p = pptr->next();`

A process can have its own user-defined structure. This structure can be established by a process and be retrieved and examined by this process and other processes using a process pointer.

Prototype: `void* get_this_struct(void)`

Example: `strctr = get_this_struct();`

A process can establish its own private structure.

Prototype: `void set_this_struct(void *strct)`

Example: `set_this_struct(strct);`

A process can retrieve a pointer to the private structure for another process

Prototype: `void* process::get_struct(void)`

Example: `strctr = pptr->get_struct();`

A process can set a private structure for another process.

Prototype: `void process::set_struct(void* strctr)`

Example: `strctr = pptr->set_struct(strctr);`

9.2 Process Queues at Facilities

The queue of processes at a facility can be managed by a set of routines.

The pointer to the first process (the process at the head of the queue) can be retrieved as follows:

Prototype: `process_t facility::first_process(void)`

Example: `pptr = f->first_process();`

The pointer to the last process at the end of the queue (the process at the tail of the queue) can be retrieved as follows:

Prototype: `process_t facility::last_process(void)`

Example: `pptr = f->last_process();`

A specific process can be removed from a facility queue as follows:

Prototype: `process_t facility::remove_process(process_t pptr)`

Example: `pptr = f->remove_process(ppptr);`

A process can be placed in a facility queue; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void facility::insert_process(process_t pptr)`

Example: `f->insert_process(pptr);`

9.3 Process Queues at Storages

The routines for managing processes in a storage queue are as follows:

The pointer to the first process (the process at the head of the queue) can be retrieved as follows:

Prototype: `process_t storage::first_process(void)`

Example: `pptr = s->first_process();`

The pointer to the last process at the end of the queue (the process at the tail of the queue) can be retrieved as follows:

Prototype: `process_t storage::last_process(void)`

Example: `pptr = s->last_process();`

A specific process can be removed from a storage queue as follows:

Prototype: `process_t storage::remove_process(process_t pptr)`

Example: `pptr = s->remove_process(ppptr);`

A process can be placed in a storage queue; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void storage::insert_process(process_t pptr)`

Example: `s->insert_process(pptr);`

9.4 Process Queues at Buffers

A buffer has two queues of processes: the queue of processes waiting for a put operation to complete and the queue of processes waiting for a get operation to complete. These two queues are referred to as the put-queue and the get-queue respectively.

The pointer to the first process of the put-queue (the process at the head of the put-queue) can be retrieved as follows:

Prototype: `process_t buffer::put_first_process(void)`

Example: `pptr = b->put_first_process();`

The pointer to the last process at the end of the put-queue (the process at the tail of the put-queue) can be retrieved as follows:

Prototype: `process_t buffer::put_last_process(void)`

Example: `pptr = b->put_last_process();`

A specific process can be removed from a put-queue queue as follows:

Prototype: `process_t buffer::put_remove_process(process_t pptr)`

Example: `pptr = b->put_remove_process(ppptr);`

A process can be placed in a buffer put-queue; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void buffer::put_insert_process(process_t pptr)`

Example: `buffer_put_insert_process(b, pptr);`

The pointer to the first process of the get-queue (the process at the head of the put-queue) can be retrieved as follows:

Prototype: `process_t buffer::get_first_process(void)`

Example: `pptr = b->get_first_process();`

The pointer to the last process at the end of the get-queue (the process at the tail of the get-queue) can be retrieved as follows:

Prototype: `process_t buffer::get_last_process(void)`

Example: `pptr = b->get_last_process();`

A specific process can be removed from a get-queue as follows:

Prototype: `procees_t buffer::get_remove_process(process_t pptr)`

Example: `pptr = b->get_remove_process(ppptr);`

A process can be placed in a buffer get-queue; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void buffer::get_insert_process(process_t pptr)`

Example: `b->get_insert_process(pptr);`

9.5 Process Queues at Events

An event has two queues of processes: the queue of processes doing a wait operation and the queue of processes doing a queue operaton. These two queues are referred to as the wait-queue and the queue-queue respectively.

The pointer to the first process of the wait-queue (the process at the head of the wait-queue) can be retrieved as follows:

Prototype: `process_t event::first_wait_process(void)`

Example: `pptr = e->first_wait_process();`

The pointer to the last process at the end of the wait-queue (the process at the tail of the wait-queue) can be retrieved as follows:

Prototype: `process_t event::last_wait_process(void)`

Example: `pptr = e->last_wait_process();`

A specific process can be removed from a wait-queue queue as follows:

Prototype:
```
process_t event::remove_wait_process(process_t
                                     pptr)
```

Example: `pptr = e->remove_wait_process(ppptr);`

A process can be placed in an event wait-queue; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void e->insert_wait_process(process_t pptr)`

Example: `e->insert_wait_process(pptr);`

The pointer to the first process of the queue-queue (the process at the head of the queue-queue) can be retrieved as follows:

Prototype: `process_t event:first_queuue_process()`

Example: `pptr = e->first_queuue_process();`

The pointer to the last process at the end of the queue-queue (the process at the tail of the queue-queue) can be retrieved as follows:

Prototype: `process_t event_last_queue_process(void)`

Example: `pptr = e->last_queue_process();`

A specific process can be removed from a queue-queue as follows:

Prototype: `process_t event:remove_queue_process(process_t pptr)`

Example: `pptr = event:remove_queue_process(ppptr);`

A process can be placed in a queue-queue; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void event:insert_queue_process(pptr)`

Example: `e->insert_queue_process(pptr);`

9.6 Process Queues and Message Lists at Mailboxes

A CSIM mailbox consists of queue of processes waiting to receive messages and a list of messages waiting to be received by processes.

The pointer to the first process of the process queue at a mailbox (the process at the head of the process-queue) can be retrieved as follows:

Prototype: `process_t mailbox::first_process(void)`

Example: `pptr = mb->first_process();`

The pointer to the last process at the end of the process queue at a mailbox (the process at the tail of the process queue) can be retrieved as follows:

Prototype: `process_t mailbox::last_process(void)`

Example: `pptr = mb->last_process();`

A specific process can be removed from the process-queue at a mailbox as follows:

Prototype: `process_t mailbox::remove_process(pptr)`

Example: `pptr = mb->remove_process(pptr);`

A process can be placed in the process queue at a mailbox; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void mailbox::insert_process(process_t pptr)`

Example: `mb->insert_process(pptr);`

There is a pointer to a message pointer:

Example: `message *msg_ptr;`

Example: `message_t msg_ptr;`

The pointer to the first message of the message list at a mailbox can be retrieved as follows:

Prototype: `message_t mailbox::first_msg()`

Example: `msg_ptr = mb->first_msg();`

The pointer to the last message at the end of the message list at a mailbox can be retrieved as follows:

Prototype: `message_t mailbox::last_msg()`

Example: `msg_ptr = mb->last_msg();`

A specific message can be removed from the message list at a mailbox as follows:

Prototype: `message_t mailbox::remove_message(message_t mptr)`

Example: `msg_ptr = mb->remove_message(mptr);`

A message can be placed in the message list at a mailbox; the position of this message is at the end of the message list.

Prototype: `void mailbox:insert_message(message_t mptr)`

Example: `mb->insert_message(mptr);`

An instance of a message class is a message object. A pointer to a message object is defined as follows:

Example: `message *mptr;`

Example: `message_t mptr;`

The content (the integer payload for the message) can be retrieved from a message object as follows:

Prototype: `long message::content(void);`

Example: `msg = msg_ptr->content();`

The next message in a list of messages can be accessed as follows:

Prototype: `message_t message:: next_msg(void)`

Example: `msg_ptr = msg_ptr–>next_msg();`

10 Introduction to Statistics Gathering

CSIM automatically gathers and reports performance statistics for certain types of model components, including facilities and storages. CSIM also provides four general-purpose statistics gathering tools: *tables, qtables, meters*, and *boxes.* These tools can be used for the following purposes:

- To obtain statistics other than mean values for facilities and storages
- To obtain statistics for other model components, such as mailboxes and events
- To obtain statistics for selected submodels or for the model considered as a whole
- To employ the run length control algorithms provided with CSIM (see section 16.3, “Run Length Control”)

Any statistics can of course be gathered by declaring and updating variables in a CSIM program. But, the statistics gathering tools are powerful and comprehensive, and their use will decrease the likelihood of programming errors that lead to incorrect statistics. Formatted reports of the statistics gathered with these tools can easily be included in the model output.

The following steps are suggested for adding statistics gathering to a model:

- Identify what statistics are of interest and which statistics gathering tools are appropriate.
- Declare a global pointer (variable) for each statistics gathering tool that will be used.

- Initialize each statistics gathering tool, usually at the beginning of the *sim* function.
- Add instrumentation (*i.e*., function calls) to the model to feed data to the tools.
- Generate reports by calling the *report* function.

The magnitudes of the performance statistics obviously depend on the time unit that is chosen for the model. Most of the reports produced by the statistics gathering tools will accommodate floating point numbers with six digits to the left of the decimal point and six digits to the right of the decimal point. Up to nine digits can be displayed for integer values. The time unit should be chosen to avoid performance values so far from unity that digits of interest are not displayed.

11 Tables

A table is used to gather statistics on a sequence of discrete values such as interarrival times, service times, or response times. Data values are “recorded” in a table to include them in the statistics. A table does not actually store the recorded values; it simply updates the statistics each time a value is included. (See section 11.6, “Moving Windows,“ for the only exception to this rule.)

The statistics maintained by a table include the minimum, maximum, range, mean, variance, standard deviation, and coefficient of variation. Optional features for a table allow the creation of a histogram, the calculation of confidence intervals, and the computation of statistics for values in a moving window.

First-time users of tables should focus on the following three sections, which explain how to set up tables, record values, and produce reports. Subsequent sections describe the more advanced features of tables.

11.1 Declaring and Initializing a Table

A pointer to a dynamic table is declared in a CSIM program as follows:

Dynamic Example: `table *td;`

Before a table can be used, it must be initialized by invoking the *table* constructor:

Prototype: `table::table(char* name);`

Static Example: `table ts("response times");`

Dynamic Example: `td = new table("response times");`

The table name is used only to identify the table in the output reports. Up to 80 characters in the name will be stored by CSIM. A newly created table contains no values and all the statistics are zero.

A table can be initialized as a permanent table using the *permanent_table* function.

Prototype: `permanent_table::permanent_table(char* name)`

Dynamic Example: `td = new permanent_table("response times");`

The information in a permanent table is not cleared when the *reset* function is called, and a permanent table is not deleted when *rerun* is called. In all other ways, a permanent table is exactly like any other table. Permanent tables are often used to gather data across multiple runs of a model. As a general rule, do not make a table permanent unless you have a specific reason for doing so.

11.2 Tabulating Values

A value is included in a table using the *record* function.

Prototype: `void table::tabulate(double value)`

Dynamic Example: `td->tabulate(1.0);`

Tables are designed to maintain statistics on data of type double. Data of other types, such as integer, must be cast to type double in the call to record.

The record() method is equivalent to the tabulate() methods:

Prototype: `void table::record(double value)`

Dynamic Example: `td->record(1.0);`

11.3 Producing Reports

Reports for tables are most often produced by calling the *report* procedure, which prints reports for all statistics gathering objects. A report can be generated for a specified table at any time by calling the *report_table* method.

Prototype: `void table::report(void)`

Static Example: `ts.report();`

Reports can be produced for all existing tables by calling the *report_tables* function.

Prototype: `void report_tables(void)`

Example: `report_tables();`

The report for a table will include the table name and all statistics, as illustrated below. If the table is empty, a message to that effect is printed instead of the statistics.

TABLE 1: response times			
minimum	0.009880	mean	2.881970
maximum	13.702809	variance	7.002668
range	13.692929	standard deviation	2.646255
observations	962	coefficient of var	0.918211

A summary report for all tables can be generated by calling the *table_summary* function.

Prototype: `void table_summary(void)`

Example: `table_summary();`

The report that is produced contains one line for each table and includes only a subset of the statistics. If a table is empty, no statistics will appear in the last three columns.

TABLE SUMMARY

name	observations	mean	maximum	standard deviation
response times	962	2.881970	13.702809	2.646255

11.4 Histograms

A histogram can be specified for a table in order to obtain more detailed information about the recorded values. The mode and other percentiles can often be estimated

from a histogram. A histogram is specified for a table by calling the *table_histogram* function.

Prototype: `void table::add_histogram(long nbucket, double min, double max)`

Static Example: `ts.add_histogram(10, 0.0, 10.0);`

The number of buckets in the histogram will be *nbucket*. The smallest value in the first bucket will be *min*; the largest value in the last bucket will be *max*. All buckets will have the same width of (*max*-*min*)/*nbucket*. An underflow bucket and an overflow bucket will automatically be created if needed to hold values less than *min* or greater than *max.*

Usually, a histogram is specified for a table immediately after the table is initialized. Additional calls can be made to *add_histogram* to change the characteristics of the histogram, but only if the table is empty.

A report for a table having a histogram will include an additional section as illustrated below. For each bucket in the histogram, the following information will be displayed: the smallest value the bucket can hold, the number of values in the bucket, the proportion of all values that are in the bucket, the proportion of all values in the bucket and all preceding buckets, and a bar whose length corresponds to the proportion of values in the bucket.

lower limit	frequency	proportion	cumulative proportion	
0.00000	265	0.275468	0.275468	**********
1.00000	219	0.227651	0.503119	********
2.00000	125	0.129938	0.633056	****
3.00000	92	0.095634	0.728690	***
4.00000	74	0.076923	0.805613	***
5.00000	54	0.056133	0.861746	**
6.00000	53	0.055094	0.916840	**
7.00000	38	0.039501	0.956341	*
8.00000	8	0.008316	0.964657	*
9.00000	8	0.008316	0.972973	*
>=10.00000	26	0.027027	1.000000	*

If leading or trailing buckets contain no values, the lines in the report for these buckets will not be printed. This allows the histogram to be output as compactly as possible without losing any information.

CSIM must save information for each bucket in a histogram. Consequently, the storage requirements for a table that has a histogram are proportional to the number of buckets.

11.5 Confidence Intervals

CSIM can automatically compute confidence intervals for the mean of the data in any table. The confidence interval calculations are enabled by calling the *table_confidence* function.

Prototype: `void table::confidence(void)`

Static Example: `ts.confidence();`

If confidence intervals have been requested, the report for a table will have an additional section, as illustrated below.

confidence intervals for the mean after 50000 observations

level	confidence interval	rel. error
90 %	4.114119 +/- 0.296434 = [3.817684, 4.410553]	0.077648
95 %	4.114119 +/- 0.354041 = [3.760078, 4.468159]	0.078837
98 %	4.114119 +/- 0.421555 = [3.692563, 4.535674]	0.080279

Chapter 16, “Confidence Intervals and Run Length Control,” describes confidence intervals in detail and explains how to interpret the information in this report.

11.6 Moving Windows

By default, all values recorded in a table are included in the statistics. If a moving window is specified for a table, only the last *n* values are used in computing the statistics, where *n* is called the window size. A moving window is specified for a table using the *moving_window* method.

Prototype: `void table::moving_window(long n)`

Static Example: `ts.moving_window(1000);`

Usually, a table’s moving window is specified immediately after the table is initialized. Additional calls can be made to *table_moving_window* to change the table’s window size. It is an error to specify a moving window for a table that is not empty.

If a table has a window size of *n*, the last *n* values recorded in the table must be saved by CSIM. Consequently, the storage requirements for a table having a moving window are proportional to its window size.

11.7 Inspector Methods

All statistics maintained by a table can be retrieved during the execution of a model or upon its completion. The attributes of a table (*i.e.*, its name and moving window size) can also be retrieved.

Prototype:	Functional value:
`char *table::name()`	pointer to name of table
`long table::size()`	size of moving window
`long table::cnt()`	number of values recorded
`double table::min()`	minimum value
`double table::max()`	maximum value
`double table::sum()`	sum of values
`double table::sum_square()`	sum of squares of values
`double table::mean()`	mean of values
`double table::range()`	range of values
`double table::var()`	variance of values
`double table::stddev()`	standard deviation of values
`double table::cv()`	coefficient of variation of values

The following inspector methods retrieve information about the confidence interval associated with a table:

Prototype:	Functional value:
`double table::conf_halfwidth (double level)`	half width
`double table::conf_lower (double level)`	lower end
`double table::conf_upper(double level)`	upper end

```
void table::compute_confidence_statistics(double
                 conf_level, double *mean_of_means,
                 double *half_width, *rel_error)
```

confidence interval values

The compute_confidence_statistics() function computes the confidence interval at the specified confidence level for the data in the table. The values returned are the estimate of the mean, the half-width of the interval and the relative error of the estimate. Note that the addresses of the returned values are inputs to the function.

The following inspector methods retrieve information about the run length control associated with a table:

Prototype:	Functional value:
`long table::batch_size()`	current size of batch
`long table::batch_count()`	number of batches used
`long table::converged()`	TRUE or FALSE
`double table::conf_mean()`	mid point of conf. int.
`double table::conf_accuracy (double level)`	accuracy achieved

Although most statistics are mathematically undefined if there is no data, the corresponding inspector methods return a value of zero if the table is empty.

The following inspector methods retrieve information about the histogram associated with a table.

Prototype:	Functional value:
`long table::hist_num()`	number of buckets
`double table::hist_low()`	smallest value that is not underflow
`double table::hist_high()`	largest value that is not overflow
`double table::hist_width()`	width of each bucket
`long table::hist_bucket (long i)`	number of values in bucket i

The number of buckets in a histogram does not include the underflow or overflow buckets. Bucket number *0* is the underflow bucket; bucket number *1+histogram_num*( ) is the overflow bucket. If a histogram has not been specified for a table, the above inspector methods all return zero values.

The inspector methods that retrieve information about the results of run-length control are described in section 16.3.

11.8 Renaming a Table

The name of a table can be changed at any time using the *set_name* method.

Prototype: `void table::set_name(char *new_name)`

Static Example: `ts.set_name("elapsed time");`

Only the first 80 characters of the table's name are stored.

11.9 Resetting a Table

Resetting a table causes all information maintained by the table to be reinitialized. All optional features selected for the table (*e.g.*, histogram, confidence intervals, moving window) remain in effect and are also reinitialized.

The *reset* function is usually used to reset all statistics gathering tools at once. A specific table can be reset using the *reset_table* function.

Prototype: `void table::reset(void)`

Static Example: `ts.reset();`

Although permanent tables are not reset by the *reset* function, they can be reset explicitly by calling *reset_table*.

11.10 Deleting a Table

When a dynamic table is no longer needed, its storage can be reclaimed using the *delete_table* function.

Example: `delete td;`

Once a table has been deleted, it must not be further referenced. If enhancements (either histogram, confidence intervals, or moving window) have been defined for a table, the each of these enhancements is also deleted when the corresponding table is deleted.

11. Tables

12 Qtables

A qtable is used to gather statistics on an integer-valued function of time, such as the length of a queue, the population of a subsystem, or the number of available resources. Every change in the value of the function must be “noted” by calling a CSIM function. A qtable does not actually save the functional values; it simply updates the statistics each time the value changes. (See section 12.6 for the only exception to this rule.)

A qtable is also used to gather statistics on a real-valued (double) function of time. Every change in the value of the function is noted by calling a CSIM function. A qtable for real-valued entries is a qtable object; values of the function are entered using the record_value() method.

The statistics maintained by a qtable include the minimum, maximum, range, mean, variance, standard deviation, and coefficient of variation. The number of changes in the functional value is maintained, as well as the initial and final values. Optional features for a qtable allow the creation of a histogram, the calculation of confidence intervals, and the computation of statistics for values in a moving window.

First-time users of qtables should focus on the following three sections, which explain how to set up qtables, note changes in their values, and produce reports. Subsequent sections describe the more advanced features of qtables.

12.1 Declaring and Initializing a Qtable

A pointer to a dynamic qtable is declared in a CSIM program as follows:

Dynamic Example: `qtable *qtd;`

Before a qtable can be used, it must be initialized by invoking the *qtable* constructor.

Prototype: `qtable::qtable(char *name)`

Static Example: `qtable qts("queue length");`

Dynamic Example: `qtd = new qtable("queue length");`

The qtable name is used only to identify the qtable in the output reports. Up to 80 characters in the name will be stored by CSIM. A newly created qtable has an initial value of zero. To create a qtable with a non-zero initial value, call the *note_state* method (described below) immediately after creating the qtable.

A qtable can be initialized as a permanent qtable using the *permanent_qtable* class.

Prototype: `permanent_qtable::permanent_qtable(char *name)`

Dynamic Example: `qtd = new permanent_qtable("queue length");`

12.2 Noting a Change in Value

The most common way for the value of a qtable to change is for it to increase or decrease by one. Such a change would occur when a customer joins a queue or a

resource is allocated. The value of a qtable is increased by one using the *note_entry* method.

Prototype: `void qtable::note_entry(void)`

Dynamic Example: `qtd -> note_entry();`

The value of a qtable is decreased by one using the *note_exit* method.

Prototype: `void qtable::note_exit(void)`

Dynamic Example: `qtd->note_exit();`

The value of a qtable can be changed to an arbitrary number using the *note_value* method.

Prototype: `void qtable::note_value(long value)`

Example: `qtd->note_value(12);`

The value of a qtable can be changed to an arbitrary floating (double) number using the *record_value* method.

Prototype: `void qtable::record_value(double value)`

Example: `qtd->note_value(1.75);`

The value of a qtable can be initialized to an arbitrary floating (double) number using the *set_initial_value* method.

Prototype: `void qtable::set_initial_value(double value)`

Example: `qtd->set_initial_value(-1.0);`

Note: using the record_value() method and the note methods on the same qtable is incorrect.

12.3 Producing Reports

Reports for qtables are most often produced by calling the *report* function, which prints reports for all statistics gathering objects. A report can be generated for a specified qtable at any time by calling the *report_qtable* method.

Prototype: `void qtable::report(void)`

Dynamic Example: `qtd->report();`

Reports can be produced for all existing qtables by calling the *report_qtables* function.

Prototype: `void report_qtables(void)`

Example: `report_qtables();`

The report for a qtable will include the qtable name and all statistics, as illustrated below. If no time has passed since the creation or reset of the qtable, a message to that effect is printed instead of the statistics.

QTABLE 1:	queue length				
initial	0	minimum	0	mean	2.788416
final	4	maximum	14	variance	8.529951
entries	966	range	14	standard deviation	2.920608
exits	962			coeff of variation	1.047408

A summary report for all qtables can be generated by calling the *qtable_summary* function.

Prototype: `void qtable_summary (void)`

Example: `qtable_summary ();`

The report that is produced contains one line for each qtable and includes only a subset of the statistics. If no time has passed, no statistics will appear in the last three columns.

```
QTABLE SUMMARY
                                                              standard
name          entries  exits    mean     maximum      deviation
-------------------------------------------------------------------
queue length   966     962   2.788416     14          2.920608
```

Note: The report and histogram for a qtable with floating values are similar to reports for a qtable with integer values, except that double values are printed where appropriate.

12.4 Histograms

A histogram can be specified for a qtable in order to obtain more detailed information about the functional values. Depending on how the qtable is being used, its histogram might give the distribution of the queue lengths, the subsystem population, or the number of available resources. A histogram is specified for a table by calling the *add_histogram* method.

Prototype: `void qtable::add_histogram(long nbucket, long min, long max)`

Example: `qtd->add_histogram(11, 0, 10);`

The number of buckets in the histogram will be (no greater than) *nbucket*. The smallest value in the first bucket will be *min*; the largest value in the last bucket will be *max.* All buckets will have the same width, which will be rounded up to an integer

if necessary. An underflow bucket and an overflow bucket will automatically be created if needed to hold values less than *min* or greater than *max.*

Caution: The *min* and *max* parameters of *qtable_histogram* are of type long, whereas the analogous parameters of *table_histogram* are of type double.

Usually, a histogram is specified for a qtable immediately after the qtable is initialized. Additional calls can be made to *qtable_histogram* to change the characteristics of the histogram, but only if the qtable is empty.

A report for a qtable having a histogram will include an additional section as illustrated below. For each bucket in the histogram, the following information will be displayed: the smallest value the bucket can hold, the total time the functional value was in the bucket, the proportion of time that the functional value was in the bucket, the proportion of all functional values in the bucket and all preceding buckets, and a bar whose length corresponds to the proportion of time the functional value was in the bucket.

number	total time	proportion	cumulative proportion	
0	248.74145	0.249003	0.249003	**********
1	185.45534	0.185651	0.434654	********
2	157.13503	0.157300	0.591954	********
3	100.01937	0.100125	0.692079	********
4	78.14196	0.078224	0.770303	******
5	62.59210	0.062658	0.832961	*****
6	44.38455	0.044431	0.877392	****
7	35.33308	0.035370	0.912762	***
8	25.94494	0.025972	0.938735	**
9	21.48465	0.021507	0.960242	**
>= 10	39.71625	0.039758	1.000000	***

If leading or trailing buckets contain no values, the lines in the report for these buckets will not be printed. This allows the histogram to be output as compactly as possible without losing any information.

CSIM must save information for each bucket in a histogram. Consequently, the storage requirements for a qtable that has a histogram are proportional to the number of buckets.

12.5 Confidence Intervals

CSIM can automatically compute confidence intervals for the mean value of any qtable. The confidence interval calculations are enabled by calling the *confidence* method.

Prototype: `void qtable::confidence(void)`

Dynamic Example: `qtd->confidence();`

If confidence intervals have been requested, the report for a qtable will include an additional section, as illustrated below.

confidence intervals for the mean after 29600.000000 time units		
level	confidence interval	rel. error
90 %	4.319412 +/- 0.491696 = [3.827715, 4.811108]	0.128457
95 %	4.319412 +/- 0.588209 = [3.731203, 4.907621]	0.157646
98 %	4.319412 +/- 0.701971 = [3.617441, 5.021382]	0.194052

Section 16.1, “Confidence Intervals,” describes confidence intervals in detail and explains how to interpret the information in this report.

12.6 Moving Windows

By default, all changes to the value of a qtable are included in the statistics. If a moving window is specified for a qtable, only the last *n* changes are used in computing the statistics, where *n* is called the window size. A moving window is specified for a qtable using the *moving_window* method.

Prototype: `void qtable::moving_window(long n)`

Dynamic Example: `qtd->moving_window (1000);`

Usually, a qtable's moving window is specified immediately after the qtable is initialized. Additional calls can be made to *qtable_moving_window* to change the qtable's window size. It is an error to specify a moving window for a qtable that is not empty.

If a qtable has a window size of *n*, the last *n* changes noted for the qtable must be saved by CSIM. Consequently, the storage requirements for a qtable having a moving window are proportional to its window size.

Note: In an alternate implementation of moving windows, the window size would be specified as an amount of time. The storage requirements of such an implementation would be non-constant and potentially prohibitive.

12.7 Inspector Methods

All statistics maintained by a qtable can be retrieved during the execution of a model or upon its completion. The attributes of a qtable (*i.e.*, its name and moving window size) can also be retrieved.

Prototype:	Functional value:
`char *qtable::name()`	pointer to name of qtable
`long qtable::size()`	moving window size
`long qtable::entries()`	number of note_entry's
`long qtable::exits()`	number of note_exit's
`long qtable::min()`	minimum value
`long qtable::max()`	maximum value

`long qtable::initial()`	initial value
`long qtable::current()`	current value
`double qtable::sum()`	sum of values weighted by time
`double qtable::sum_square()`	sum of squared weighted
`double qtable::mean()`	mean value
`long qtable::range()`	range of values
`double qtable::var()`	variance of values
`double qtable::stddev()`	standard deviation of values
`double qtable::cv()`	coefficient of variation of values

When a qtable has recorded double values, the following inspector methods should be used (instead of the equivalent methods used listed above)

Prototype:	**Functional Values:**
`double qtable::entries_dbl()`	sum of positive changes
`double qtable::exits_dbl()`	sum of negative changes
`double qtable::range_dbl()`	range of values
`double qtable::min_dbl()`	minimum value
`double qtable::max_dbl()`	maximum value
`double qtable::initial_dlb()`	initial value
`double qtable::current_dbl()`	current value

The following inspector methods retrieve information about the confidence interval associated with a table:

Prototype:	**Functional value:**
`double qtable::conf_halfwidth (double level)`	half width

`double qtable::conf_lower (double level)`	lower end
`double qtable::conf_upper (double level)`	upper end

The following inspector methods retrieve information about the run length control associated with a table:

Prototype:	**Functional value:**
`long qtable::batch_size()`	current size of batch
`long qtable::batch_count()`	number of batches used
`long qtable::converged()`	TRUE or FALSE
`double qtable::conf_mean()`	mid point of conf. int.
`double qtable::conf_accuracy (double level)`	accuracy achieved

Many statistics are mathematically undefined if zero time has passed since the creation or reset of a qtable. The corresponding inspector methods return a value of zero in this case.

The following inspector methods retrieve information about the histogram associated with a qtable.

Prototype:	**Functional value:**
`long qtable::hist_num()`	number of buckets
`double qtable::hist_low()`	smallest value that is not underflow
`double qtable::hist_high()`	largest value that is not overflow
`double qtable:hist_width()`	width of each bucket
`qtable::hist_bucket (long i)`	total time value is in bucket

The number of buckets in a histogram does not include the underflow or overflow buckets. Bucket number *0* is the underflow bucket; bucket number

1+histogram_num( ) is the overflow bucket. If a histogram has not been specified for a qtable, the above inspector methods all return zero values.

The inspector methods that retrieve information about the results of run-length control are described in section 16.3, “Run Length Control.”

12.8 Renaming a Qtable

The name of a qtable can be changed at any time using the *set_name* method.

Prototype: `void qtable::set_name(char *new_name)`

Dynamic Example: `qtd-> set_name (“number in queue”);`

Only the first 80 characters of the qtable’s name are stored.

12.9 Resetting a Qtable

Resetting a qtable causes all information maintained by the qtable to be reinitialized, except that the current value is saved for use in computing future values. All optional features selected for the qtable (*e.g.,* histogram, confidence intervals, moving window) remain in effect and are also reinitialized.

The *reset* function is usually used to reset all statistics gathering tools at once. A specific qtable can be reset using the *reset* method.

Prototype: `void qtable::reset(void)`

Example: `qtd->reset_qtable ();`

Although permanent qtables are not reset by the *reset* function, they can be reset explicitly by calling *reset_qtable*.

12.10 Deleting a Qtable

When a dynamic qtable is no longer needed, its storage can be reclaimed as follows:

Example: `delete qtd;`

Once a qtable has been deleted, it must not be further referenced.

13 Meters

A meter is used to gather statistics on the flow of entities such as customers or resources past a specific point in a model. Meters can be used to measure arrival rates, completion rates, and allocation rates. A meter can be thought of as a probe that is inserted at some point in a model.

While a meter primarily measures the rate at which entities flow past it, a meter also keeps statistics on the times between passages. These interpassage times are recorded in a table, which is an integral part of every meter.

First-time users of meters should focus on the following three sections, which explain how to set up meters, update meters, and produce reports. Subsequent sections describe the more advanced features of meters.

13.1 Declaring and Initializing a Meter

A pointer to a dynamic meter is declared in a CSIM program as follows:

Dynamic Example:
```
meter *md;
```

Before a meter can be used, it must be initialized by invoking the *meter* constructor.

Prototype: `meter::meter(char *name)`

Static Example: `meter ms("system completions");`

Dynamic Example: `md = new meter("system completions");`

The meter name is used only to identify the meter in the output reports. Up to 80 characters in the name will be stored by CSIM.

13.2 Instrumenting a Model

An entity notes its passage by a meter using the *note_passage* method.

Prototype: `void meter::note_passage(void)`

Dynamic Example: `md-> note_passage();`

For the statistics to be accurate, every entity of interest must note its passage and do so at the correct time.

13.3 Producing Reports

Reports for meters are most often produced by calling the *report* function, which prints reports for all statistics gathering objects. A report can be generated for a specified meter at any time by calling the *report* method.

Prototype: `void meter::report(void)`

Example: `md->report();`

Reports can be produced for all existing meters by calling the *report_meters* function.

Prototype: `void report_meters(void)`

Example: `report_meters();`

The report for a meter, as illustrated below, will include the meter name, the number of passages, the passage rate, and statistics on the interpassage times. If no time has elapsed, a message to that effect is printed instead of the statistics.

```
METER 2:  System completions

 count                494       rate                  0.988000

interpassage time statistics

 minimum         0.001258       mean                  1.008764
 maximum         6.533026       variance              0.994894
 range           6.531768       standard deviation    0.997444
 observations         494       coefficient of var    0.988778
```

A summary report for all meters can be generated by calling the *meter_summary* function.

Prototype: `void meter_summary(void)`

Example: `meter_summary();`

The report that is produced contains one line for each meter and includes only a subset of the statistics. If no time has passed, undefined statistics will be omitted.

METER SUMMARY

name	passages	rate	mean ip time	max ip time
System arrivals	501	1.002000	0.997048	6.679665
System completions	494	0.988000	1.008764	6.533026

13.4 Histograms

A histogram can be specified for the interpassage times of a meter. This is accomplished using the *add_histogram* method.

Prototype:
```
void meter::add_histogram(long nbucket,
    double min, double max)
```

Dynamic Example: `md -> add_histogram(10, 0.0, 10.0);`

The histogram for a meter is exactly the same as the histogram for a table. See section 11.4, “Histograms,” for details.

13.5 Confidence Intervals

CSIM can automatically compute confidence intervals for the mean interpassage time at a meter. The confidence interval calculations are enabled by calling the *confidence* method.

Prototype: `void meter::confidence(void)`

Dynamic Example: `md -> confidence();`

The confidence intervals for a meter are the same as the confidence intervals for a table. See section 16.1, “Confidence Intervals,” for details.

13.6 Moving Windows

Moving windows are not supported by meters.

13.7 Inspector Methods

All statistics maintained by a meter can be retrieved during the execution of a model or upon its completion. The name of a meter can also be retrieved.

Prototype:	Functional value:
`char* meter::name()`	pointer to name of meter
`double meter::start_time()`	time at which recording began
`long meter::cnt()`	number of passages noted
`double meter::rate()`	rate of passages
`table* meter::ip_table()`	pointer to interpassage time table

Although the passage rate is mathematically undefined if no time has passed, the *rate* method returns the value zero in this case.

The pointer to a meter's interpassage time table can be passed to the inspector functions for a table in order to obtain interpassage time statistics.

Example:
```
max_ip_time = m->ip_table()->max();
```

If no passages have occurred, the interpassage time table is empty. The interpassage time contributed by the first passage is the time from the beginning of the observation period to that first passage.

13.8 Renaming a Meter

The name of a meter can be changed at any time using the *set_name_meter* function.

Prototype: `void meter::set_name(char *new_name)`

Dynamic Example: `md -> set_name("system departures");`

Only the first 80 characters of the meter's name are stored.

13.9 Resetting a Meter

Resetting a meter causes all information maintained by the meter to be reinitialized, except that the time of the last passage is saved for use in computing the next interpassage time. All optional features selected for the meter (*e.g.*, histogram, confidence intervals, moving window) remain in effect and are also reinitialized.

The *reset* function is usually used to reset all statistics gathering tools at once. A specific meter can be reset using the *reset_meter* function.

Prototype: `void meter::reset(void)`

Dynamic Example: `md->reset();`

13.10 Deleting a Meter

When a dynamic meter is no longer needed, its storage can be reclaimed using the *delete_meter* function.

Example: `delete md;`

Once a meter has been deleted, it must not be further referenced.

14 Boxes

A box conceptually encloses part or all of a model. The box gathers statistics on the number of entities in the box (*i.e*., the population) and the amount of time entities spend in the box (*i.e*., the elapsed time). An entity might be a customer, a message, or a resource. Boxes are usually used to gather statistics on queue lengths, response times, and populations. Instrumenting a model involves inserting function calls at the places that entities enter and exit the box.

A table and a qtable are invisible but integral parts of every box. Statistics on the elapsed times are kept in the table, statistics on the population are kept in the qtable.

First-time users of boxes should focus on the following three sections, which explain how to set up boxes, instrument a model, and produce reports. Subsequent sections describe the more advanced features of boxes.

14.1 Declaring and Initializing a Box

A pointer to a dynamic box is declared in a CSIM program as follows:

Dynamic Example:
```
box *bd;
```

Before a box can be used, it must be initialized by invoking the *box* constructor.

Prototype: `box::box(char *name)`

Static Example: `box bs("box");`

Dynamic Example: `bd = new box("system");`

The box name is used only to identify the box in the output reports. Up to 80 characters in the name will be stored by CSIM. A newly created box is always empty. To create a non-empty box, call the *enter_box* function (described in the following section) the appropriate number of times immediately after creating the box.

A box can be initialized as a permanent box using the *box* class.

Prototype: `box::permanent_box(char * name)`

Dynamic Example: `b = new permanent_box("system");`

The information in a permanent box is not cleared when the *reset* function is called, and a permanent box is not deleted when *rerun* is called. In all other ways, a permanent box is exactly like a box. As a general rule, do not make a box permanent unless you have a specific reason for doing so.

14.2 Instrumenting a Model

An entity enters a box by calling the *enter* method.

Prototype: `double box::enter(void)`

Dynamic Example: `timestamp = bd->enter ();`

This function returns a timestamp that must be saved by the entity that entered the box. The entity exits the box by calling the *exit* method and passing to it the timestamp that it received upon entry.

Prototype: `void box::exit(double entry_time)`

Dynamic Example: `bd->exit(timestamp);`

It is the responsibility of the programmer to ensure that the integrity of the timestamp is maintained while the entity is in the box. Because boxes may be nested or may overlap, it is advisable to make the timestamp local to the CSIM process and to use a separate timestamp variable for each box. An invalid timestamp (*i.e*., one that is less than zero or greater than the current time) will cause an error.

14.3 Producing Reports

Reports for boxes are most often produced by calling the *report* function, which prints reports for all statistics gathering objects. A report can be generated for a specified box at any time by calling the *report* method.

Prototype: `void box::report(void)`

Dynamic Example: `bd->report();`

Reports can be produced for all existing boxes by calling the *report_boxes* function.

Prototype: `void report_boxes(void)`

Example: `report_boxes();`

The report for a box, as illustrated below, will include the box name, statistics on the elapsed times, and statistics on the population of the box. If the box is empty or no time has passed since its creation or reset, messages to that effect are printed instead of the statistics. Note that statistics on the elapsed times reflect only those entities that have exited the box. Entities still in the box when the report is produced contribute to the population statistics but not to the elapsed time statistics.

14. Boxes

BOX 1: Queue statistics

statistics on elapsed times

minimum	0.009880	mean	2.088345
maximum	7.943915	variance	3.211423
range	7.934035	standard deviation	1.792044
observations	494	coefficient of var	0.858117

statistics on population

initial	0	minimum	0	mean	
final	7	maximum	10	variance	
entries	501	range	10	standard deviation	
exits	494			coeff of variation	

A summary report for all boxes can be generated by calling the *box_summary* function.

Prototype: `void box_summary(void)`

Example: `box_summary();`

The report that is produced contains one line for each box and includes only a subset of the statistics. If a box is empty or no time has passed since its creation or reset, some statistics will not appear.

BOX SUMMARY

name	mean elapsed-time	maximum elapsed-time	mean population	maximum population
Queue statistic	2.088345	7.943915	2.093697	10

14.4 Histograms

A histogram can be specified for the elapsed times in a box and for the population of a box using the following functions:

Prototype: `void box::add_time_histogram(long nbucket, double min, double max)`

Dynamic Example: `bd->add_time_histogram (10, 0.0, 10.0);`

Prototype: `void box::add_number_histogram(long nbucket, long min, long max)`

Dynamic Example: `bd->add_number_histogram(10, 0, 10);`

The histogram for the elapsed times is the same as the histogram for a table. See section 9.4, “Histograms,” for details. The histogram for the population of a box is the same as the histogram for a qtable. See section 10.4, “Histograms,” for details.

Caution: The *min* and *max* parameters of *box_time_histogram* are of type double, whereas the corresponding parameters of *box_number_histogram* are of type long.

14.5 Confidence Intervals

Confidence intervals can be requested for the mean of the elapsed times in a box and for the mean population of a box using the following functions:

Prototype: `void box::time_confidence(void)`

Dynamic Example: `bd->time_confidence();`

Prototype: `void box::number_confidence(void)`

Dynamic Example: `bd->number_confidence();`

These two types of confidence intervals are identical to the confidence intervals for a table and qtable, respectively. See sections 16.1, “Confidence Intervals,” for details.

14.6 Moving Windows

Moving windows can be specified for the elapsed times in a box and for the population of a box using the following functions:

Prototype: `void box::time_moving_window(long n)`

Dynamic Example: `bd->time_moving_window(1000);`

Prototype: `void box::number_moving_window(long n)`

Dynamic Example: `bd->number_moving_window(1000);`

The window for the elapsed times specifies the number of entities whose elapsed times will be included in the statistics. The window for the population specifies the number of changes in the population that will be included in the statistics. Consequently, the simulation time covered by these two windows may not be the same.

14.7 Inspector Methods

All statistics maintained by a box can be retrieved during the execution of a model or upon its completion. The name of a box can also be retrieved.

Prototype:	Functional value:
`char* box::name()`	pointer to name of box
`table* box::time_table()`	pointer to elapsed time table
`qtable* box::number_qtable()`	pointer to population qtable

The pointer to a box's elapsed time table can be passed to the inspector methods for a table in order to obtain statistics on the times that entities have spent in the box.

Dynamic Example:
```
max_time_in_box =
    bd->time_table()->max();
```

If no entities have exited the box, the table will be empty and zeros will be returned for the undefined statistics.

The pointer to a box's population qtable can be passed to the inspector methods for a qtable in order to obtain statistics on the population.

Dynamic Example:
```
max_population =
    bd->number_qtable->max();
```

If no time has passed, zero values will be returned for the undefined statistics.

14.8 Renaming a Box

The name of a box can be changed at any time using the *set_name* method.

Prototype: `void box::set_name(char *new_name)`

Dynamic Example: `bd->set_name("system");`

Only the first 80 characters of the box's name are stored.

14.9 Resetting a Box

Resetting a box causes all information maintained by the box to be reinitialized, except that the number currently present in the box is saved for use in computing future populations. All optional features selected for the box (*e.g.*, histogram, confidence intervals, moving window) remain in effect and are also reinitialized.

The *reset* function is usually used to reset all statistics gathering tools at once. A specific box can be reset using the *reset* method.

Prototype: `void box::reset(void)`

Dynamic Example: `bd->reset();`

Although permanent boxes are not reset by the *reset* function, they can be reset explicitly by calling the *reset* method.

14.10 Deleting a Box

When a dynamic box is no longer needed, its storage can be reclaimed as follows:

Example: `delete bd;`

Once a box has been deleted, it must not be further referenced.

15 Advanced Statistics Gathering

15.1 Example: Instrumenting a Facility

For each facility, CSIM automatically gathers and reports the following statistics:

- mean service time
- mean queue length
- utilization
- mean response time
- throughput
- number of completions

Meters and boxes can easily be used to gather more detailed statistics. The following statements show the declaration of the needed variables:

```
facility *f;
meter *arrivals;
meter *departures;
box *queue_box;
box *service_box;
```

The following statements, which would appear in the *sim* function, show the initialization of the variables:

```
f = new facility("center");
arrivals = new meter("arrivals");
departures = new meter("completions");
queue_box = new box("queue");
service_box = new box("in service");
```

The following code shows the instrumentation of the facility:

```
customer()
{
   double timestamp1;
   double timestamp2;

   create("customer");
   arrivals->note_passage();
   timestamp1 = queue_box->enter();
   f->reserve();
   timestamp2 = service_box->enter();
   hold (exponential(0.8));
   f release();
   service_box->exit(timestamp2);
   queue_box->exit(timestamp1);
   departures->note_passage();
}
```

The report for box “*queue_box*” would give statistics on response times (under the heading “statistics on elapsed times”) and queue lengths (under the heading “statistics on population”). The report for box “*service_box*” would give statistics on service times (under the heading “statistics on elapsed times”) and utilization (under the heading “statistics on population”). The report for meter “*arrivals*” would give statistics on the arrival rate and inter-arrival times. The report for meter “*departures*” would give statistics on the completion rate and inter-completion times. If the arrival and completion rates were sufficiently similar, this quantity would be called the throughput.

Obviously, histograms could be added to any of these meters and boxes to obtain information on the various distributions.

15.2 The Report Function

Although reports can be produced at any time for individual statistics gathering tools, it is most common to generate reports for all tools at the same time, usually when the simulation has converged. This can be done by calling the *report* function.

Prototype: `void report(void)`

Example: `report();`

The *report* function produces reports for all facilities, storages, and classes, followed by reports for all tables, qtables, meters, and boxes. The sequence of reports begins with a header that includes the model name, the date and time, the current simulation time, and the cpu time used.

15.3 Resetting Statistics

CSIM provides a single function that will clear all accumulated statistics without affecting the state of the system being modeled in any way. This *reset* function is most often used when warming up a simulation. The simulation is begun with the system in an empty state, simply as a matter of convenience. A small number of customers are allowed to pass through the system, hopefully taking the system closer to its equilibrium state. Then, the statistics are reset and the simulation is run until convergence is achieved.

The *reset* function has a simple interface.

Prototype: `void reset(void)`

Example: `reset();`

Reset clears the statistics that are automatically gathered for facilities, storages, events, and process classes. It also resets the statistics in all non-permanent tables, qtables, meters, and boxes being used in the program. Permanent tables are not affected by calling *reset*.

In general, resetting statistics returns all the statistical counters and timers maintained by CSIM to their initial values, which are usually zero. But, there are a few subtle and important exceptions to this rule. When a qtable is reset, it remembers the current value for use in computing future values from the relative changes specified by *note_entry* and *note_exit*. When a meter is reset, it remembers the time of the last passage for use in computing the next interpassage time. When a box is reset, it remembers the number present for use in computing future populations.

Calling *reset* in no way changes the state of the system being modeled. It does not change the simulation clock; it does not affect the streams of random numbers being used in the simulation; and it does not affect the states of processes, facilities, storages, events, and mailboxes. The *reset* function is normally called during a simulation run, whereas the *rerun* function (see section 21.4.1, “To rerun a CSIM model”) is called between successive runs.

16 Confidence Intervals and Run Length Control

Most simulations are designed so they converge to what might be called the "true solution" of the model. But, because a simulation can only be run for a finite amount of time, this true solution can never be known. This gives rise to two important questions: What is the accuracy in the results of a simulation's output? How long should a simulation be run in order to obtain a given accuracy? These questions can be answered using confidence intervals and run-length control algorithms.

Using an ad hoc technique instead of the methods described in this section can be dangerous as well as wasteful. Running a simulation for too short an amount of time will result in performance statistics that are highly inaccurate. Running a simulation for an unnecessarily long amount of time wastes computing resources and delays the completion of the simulation study. Without some type of formal analysis, the errors in simulation results cannot be quantified.

16.1 Confidence Intervals

A *confidence interval* is a range of values in which the true answer is believed to lie with a high probability. The interval can be specified in two equivalent ways, either by specifying the midpoint of the interval (which could be considered the "best guess" for the true answer) and the half-width of the interval, or by specifying the lower and upper bounds of the interval. CSIM reports the confidence interval in both formats, as illustrated below:

4.114119 +/- 0.296434 = [3.817684, 4.410553]

The probability that the true answer lies within the interval is called the *confidence level*. Since a confidence level of 100% would result in an infinitely wide confidence interval, confidence levels from 90% to 99% are most often used. Be aware that there is always a small probability (dictated by the confidence level) that the true answer lies outside the confidence interval.

Confidence intervals can be automatically generated for the mean values in any table, qtable, meter, or box simply by calling one of the following functions immediately after the statistics object has been initialized.

Prototype: `void table::confidence(void)`

Prototype: `void qtable::confidence(void)`

Prototype: `void meter::confidence(void)`

Prototype: `void box::time_confidence(void)`

Prototype: `void box::number_confidence(void)`

The technique used to calculate confidence intervals is called *batch means analysis*. It is beyond the scope of this manual to describe the mathematics underlying this technique, but any good simulation text should provide details.

If confidence intervals have been requested for a table, qtable, meter, or box, the statistics report will include a section like the following.

confidence intervals for the mean after 50000 observations

level	confidence interval	rel. error
90 %	4.114119 +/- 0.296434 = [3.817684, 4.410553]	0.077648
95 %	4.114119 +/- 0.354041 = [3.760078, 4.468159]	0.078837
98 %	4.114119 +/- 0.421555 = [3.692563, 4.535674]	0.080279

Notice that confidence intervals are calculated for three commonly used confidence levels: 90%, 95%, and 98%. The confidence intervals are reported in both of the formats described previously. The relative error measures the accuracy in the midpoint of the interval as an estimate of the true answer. It is defined to be the half-width divided by the lower bound of the interval. Like any relative error, its value suggests how many accurate digits there are in the estimate.

The algorithm for computing confidence intervals groups the observations into fixed size batches and uses only complete batches. For this reason, the number of observations used in the calculation of the confidence intervals may be slightly less than the number of observations used in computing the other performance statistics. For example, in the above report 50,000 observations were used to calculate the confidence intervals. The part of the report not shown may give the mean, variance, standard deviation, *etc.* based on 50,472 observations.

The algorithm also requires a minimum number of observations for its results to be valid. This minimum number cannot be known before running the simulation because it depends on the amount of correlation found in the statistic. If a report is produced before sufficient observations have been obtained, the message

```
> insufficient observations to compute confidence intervals
```

will appear in place of the confidence intervals. To obtain confidence intervals, run the simulation longer or use the run length control algorithm.

16.2 Inspector Functions

All values calculated by the confidence interval algorithm can be retrieved during the execution of a model or upon its completion.

Prototype:	Functional value:
`long table::batch_size()`	size of batch
`long table::batch_count()`	number of batches
`double table::conf_mean()`	midpoint of interval
`double table::conf_halfwidth(lv)`	half-width of interval
`double table::conf_lower(lv)`	lower bound of interval
`double table::conf_upper(lv)`	upper bound of interval
`double table::conf_accuracy(lv)`	accuracy achieved

```
void table::compute_confidence_statistics(double
                conf_level, double *mean_of_means,
                double *half_width, *rel_error)
```

confidence interval values

The compute_confidence_statistics() function computes the confidence interval at the specified confidence level for the data in the table. The values returned are the estimate of the mean, the half-width of the interval and the relative error of the estimate. Note that the addresses of the returned values are inputs to the function.

Prototype:	Functional value:
`long qtable::batch_size`	size of batch
`long qtable::batch_count`	number of batches
`double qtable::conf_mean`	midpoint of interval
`double qtable::conf_halfwidth`	half-width of interval
`double qtable::conf_lower`	lower bound of interval

`double qtable::conf_upper`	upper bound of interval
`double qtable::conf_accuracy`	accuracy achieved

If confidence intervals have not been requested or if there have not been sufficient observations to calculate confidence intervals, all of the above functions return zero values.

To inspect confidence interval information for meters and boxes, pass to the appropriate function listed above a pointer returned by one of the following functions: *meter::ip_table, box::time_table,* or *box::number_qtable.*

16.3 Run Length Control

If the reported confidence intervals show that the needed accuracy has not been achieved, a simulation could be run again for a longer amount of time. This has two disadvantages: repeating part of the simulation is wasteful, and it may not be clear how much longer to run the simulation the second time.

A better method is to use the run length control algorithm that is built into CSIM. This algorithm monitors the confidence interval as it narrows and automatically terminates the simulation when the desired accuracy has been achieved.

To use run length control, choose a performance measure that will be used to decide when the simulation should terminate. Instrument the model to gather statistics on this performance measure using a table, qtable, meter, or box. Immediately after the statistics gathering object has been initialized, call the appropriate function below.

Prototype:
```
void table::run_length(double accuracy,
  double conf_level, double max_time)
```

Dynamic Example:
```
t->run_length (0.01, 0.95, 10000.0);
```

16. Confidence Intervals and Run Length Control

Prototype: `void qtable::run_length(double accuracy, double conf_level, double max_time)`

Prototype: `void meter::run_length(double accuracy, double conf_level, double max_time)`

Prototype: `void box::time_run_length(double accuracy, double conf_level, double max_time)`

Prototype: `void box::number_run_length(double accuracy, double conf_level, double max_time)`

The accuracy parameter specifies the maximum relative error that will be allowed in the mean value of this performance measure. A value of 0.1 is usually used to request one digit of accuracy, 0.01 is used to request two digits of accuracy, and so forth. The *conf_level* parameter is the confidence level and usually has a value between 0.90 and 0.99. The *max_time* parameter places an upper bound on how long the simulation will run. If the specified accuracy cannot be achieved within this time, the simulation will terminate and a warning message will appear in the report.

In the main CSIM process, place the following call to the *wait* function.

```
converged.wait();
```

"Converged" is a built-in event that does not need to be declared or initialized. This event is set when the run length control algorithm determines that the requested accuracy has been achieved or when the maximum time has passed.

If run length control has been enabled, the statistics report will include a section like the following.

```
results of run length control using confidence intervals

   cpu time limit      10.0        accuracy requested      0.005000
   cpu time used        1.8        accuracy achieved       0.005000

   95.0% confidence interval: 0.998735 +/- 0.004969 = [0.993767, 1.003704]
```

The confidence interval is reported in both formats for the confidence level that was specified. If the requested accuracy was not achieved or if there were not enough observations to calculate confidence intervals, a warning message will appear in the report.

The mechanics for running a simulation until multiple performance measures have been obtained to desired accuracy are simple. Call the appropriate run length function for several statistics gathering objects and then wait on the “converged” event as many times as there are statistics to converge. However, there are some subtleties in the theory underlying this procedure. Persons interested in this topic should read section 9.7 of *Simulation Modeling and Analysis* by Law and Kelton.

16.4 Caveats

Confidence intervals attempt to bound the errors in performance statistics caused by running a simulation for a finite amount of time. They in no way measure the errors caused by the model being an unfaithful representation of the actual system.

All known techniques for computing confidence intervals are heuristics. Detecting and removing correlation from performance data is a mathematically difficult problem. Confidence intervals should always be considered to be estimates.

In spite of these limitations, it is our belief that confidence intervals and run length control play an essential role in any simulation study. Simply running a simulation for a “long time” and hoping that the performance measures will be highly accurate is an unprofessional and dangerous approach

16. Confidence Intervals and Run Length Control

.

17 Process Classes

Process classes are used to segregate data for reporting purposes. A set of usage statistics is automatically maintained for each process class. These are "printed" whenever a *report* or a *report_classes* statement is executed. In addition, facility, storage and buffer information (from *report_facilities)* is kept by process class, when process classes exist. See section 19.2, “CSIM Report Output,” for details about the reports that are generated.

17.1 Declaring and Initializing Process Classes

To declare a pointer to a dynamic process class:

Example: `process_class *c;`

A process class must be initialized via the *process_class* statement before it can be used in any other statement.

Prototype: `process_class::process_class(char *name)`

Static Example: `process_class cs("low priority");`

Dynamic Example: `c = new process_class(“low priority”)`

17.2 Using Process Classes

To have the executing process join a process class:

Prototype: `void process_class::set_process_class(void)`

Dynamic Example: `c->set_process_class();`

To retrieve a pointer to the process_class for an active process:

Prototype: `process_class *current_class(void)`

Example: `c = current_class();`

If no *set_process_class* statement is executed for a process, that process is automatically a member of the "default" class. A *report* statement will not print process class statistics for the default process class. A *report_classes* statement will print process class statistics for the default process class, but ONLY if it is the only process class. If any other process class is defined, *report_classes* will only report on non-default process classes.

17.3 Producing Reports

Reports for process classes are most often produced by calling the *report* function, which prints reports for all of the CSIM objects. Reports can be produced for all existing process classes by calling the *report_classes* function. The report for a process class gives the class id, the class name, the number of entries into the class, the average lifetime for a process in this class, the average number of hold operations executed by jobs in this class, the average time per hold and the average wait time per job in this class.

PROCESS CLASS SUMMARY

id	name	number	lifetime	hold count	hold time	wait time
0	default	493	4.05680	0.99594	4.05680	0.00000
1	low priority	293	229.66986	0.54266	2.27873	227.39113
2	high priority	198	2.18412	1.00000	1.67845	0.50567

17.4 Changing the Name of a Process Class

Prototype: `void process_class::set_name(char *new_name)`

Dynamic Example: `c->set_name("high priority");`

17.5 Resetting Process Classes

The statistics associated with a process class can be reset as follows:

Prototype: `void process_class::reset(void)`

Example: `c ->reset();`

The statistics associated with all of the process classes can be reset as follows:

Prototype: `void reset_process_classes(void)`

Example: `reset_process_classes();`

17.6 Deleting Process Classes

To delete a dynamic process class:

Dynamic Example: `delete c;`

If a facility is collecting statistics for the deleted class, this collection will continue.

17.7 Inspector Methods

These methods each return an attribute or a statistic that describes some aspect of the usage of the specified process class. The type of the returned value for each of these methods is as indicated.

Prototype:	Functional Value:
`long process_class::id()`	id of process class
`char *process_class::name()`	pointer to name of process class
`long process_class::cnt()`	number of processes in process class
`double process_class::lifetime()`	total time for all processes in process class
`long process_class::holdcnt()`	total number of holds for all processes in process class
`double process_class::holdtime()`	total hold time for all processes in process class

18 Random Numbers

Most simulations are random number driven. In such simulations, random numbers are used for interarrival times, service times, allocation amounts, and routing probabilities. For each application of random numbers in a simulation, a distribution must be chosen. The distribution determines the likelihood of different values occurring. A distribution is uniquely specified by the name of its family (such as uniform, exponential, or normal) and its parameter values (such as the mean and standard deviation). Discussions of distributions and their uses in models can be found in texts [Lake 96].

Random numbers generated by computers are actually *pseudo-random*. A sequence of values is generated using a recurrence relation that calculates the next value in the sequence from the previous value. The sequence is begun by specifying a starting value called a *seed*. A good random number generator has the property that the numbers it produces have no discernible patterns that distinguish them from truly random numbers.

Most CSIM users need only read the following two sections, which describe single stream random number generation. Those interested in building multiple-stream simulations should read the remaining sections as well.

18.1 Single Stream Random Number Generation

CSIM includes a library of functions for generating random numbers from more than 20 different distributions. Continuous distributions have values that are floating-point numbers; values from these distributions are most often used for amounts of time.

Discrete distributions have values that are integers; values from these distributions are often used for quantities of resources.

The following prototypes are for the functions that generate values from continuous distributions. The parameters *min* and *max* specify the minimum and maximum values that will be generated. The parameters *mean*, *var*, *stddev*, and *mode* specify respectively the mean, variance, standard deviation, and mode of the distribution. The parameters *shape1*, *shape2*, *shape*, *alpha*, and *beta* are all shape parameters whose meaning can be found in any text that describes these distributions.

Prototype: `double uniform(double min, double max)`

Prototype: `double uniform01( )`

Prototype: `double triangular(double min, double max, double mode)`

Prototype: `double beta(double min, double max, double shape1, double shape2)`

Prototype: `double exponential(double mean)`

Prototype: `double gamma(double mean, double stddev)`

Prototype: `double erlang(double mean, double var)`

Prototype: `double hyperx(double mean, double var)`

Prototype: `double weibull(double shape, double scale)`

Prototype: `double normal(double mean, double stddev)`

Prototype: `double lognormal(double mean, double stddev)`

Prototype: `double cauchy(double alpha, double beta)`

Prototype: `double hypoexponential(double mn, double var)`

Prototype: `double pareto(double a)`

Prototype: `double zipf(long n)`

Prototype: `double zipf_sum(long n, double *sum)`

The following prototypes are for the functions that generate values from discrete distributions. The parameters *min* and *max* specify the minimum and maximum values that will be generated. The parameter *mean* specifies the mean of the distribution. The parameters *prob_success*, *num_trials*, and *success_num* are respectively the probability of success, the number of trials, and the success number. A text that describes theses distributions should be consulted for the detailed meaning of these parameters.

Prototype: `long uniform_int(long min, long max)`

Prototype: `long bernoulli(double prob_success)`

Prototype:
```
long binomial(double prob_success, long
  num_trials)
```

Prototype: `long geometric(double prob_success)`

Prototype:
```
long negative_binomial(long success_num,
  double prob_success)
```

Prototype: `long poisson(double mean)`

Two functions must be used to efficiently generate values from an empirical distribution.

Their prototypes are shown below.

Prototype:
```
void setup_empirical(long n, double prob[],
  double cutoff[], long alias[])
```

Prototype:
```
double empirical(long n, double cutoff[],
  long alias[], double value[])
```

The *setup_empirical* function must be called once, prior to any calls to function *empirical*. It takes as input the number of values, *n*, in the distribution and an array, *prob*, that specifies the probability of generating each value. It calculates two sets of values and stores them in the arrays *cutoff* and *alias*. The contents of these arrays need not be understood to use this distribution. All arrays must be of size at least n+1. Function *empirical* is called to generate a value from an empirical distribution that has already been set-up. The function takes as input the same parameters *n*, *cut-off*, and *alias* as the *setup_empirical* function. It also takes an array, *value*, that contains the values to be generated with the probabilities that were specified in array *prob*. Each call returns one of the values in the array *value.*

18.2 Changing the Seed of the Single Stream

By default, the single stream from which all random numbers are generated is seeded with the value of 1. Unless the seed is changed, every execution of every CSIM program will use the same sequence of random numbers. The seed can be changed by calling the *reseed* function of the default stream.

Prototype: `void reseed(void*, long n)`

Example: `reseed(NIL, 13579);`

In simulations that use a single random number stream, the value of the first parameter in the function call should always be NIL. The second parameter is the positive integer that is to be used as the seed. The choice of the seed value will not affect the randomness of the numbers that are produced. Although it is most common to call reseed once at the beginning of a CSIM program, the *reseed* function can be called any number of times and from any place within a program.

The current state of the stream can be retrieved by calling the *stream_state* function.

Prototype: `long stream::state()`

Example: `i = stream_state(NIL);`

If *stream_state* is called immediately after reseeding the stream, the seed value will be returned. Otherwise, the positive integer used to produce the most recently generated random number will be returned.

18.3 Single Versus Multiple Streams

In a single stream simulation, all random numbers are produced from a single stream of pseudo-random integers. The random numbers used for a particular purpose (for example, interarrival times) are generated from a subsequence of these random integers. It is of concern to some people that the subsequence of integers may not be “as random” as the stream from which they were extracted. This concern can be alleviated by using a separate stream of pseudo-random integers for each application of random numbers in the model. So, separate streams would be used for the service times at each facility, for the allocation amounts of each storage, and so forth.

Multiple streams are also used to guarantee that exactly the same sequence of random numbers is used for the interarrival times (for example) in two different models. This technique is called common random numbers and is described in simulation texts.

There is virtually no difference in the time required to generate random number from a single stream or from multiple streams. Multiple stream simulations require slightly more programming: the multiple streams must be declared, initialized, and (perhaps) seeded, and each call to a function that generates random numbers must specify the stream to be used.

18.4 Managing Multiple Streams

A stream is declared in a CSIM program using the class *stream*.

Dynamic Example: `stream *s;`

Before a stream can be used, it must be initialized by calling the *create_stream* constructor.

Prototype: `stream::stream(void)`

Example: `s = new stream();`

By default, streams are created with seeds that are spaced 100,000 values apart. CSIM contains a table of 100 such seed values; if more than 100 streams are created, the seed values are reused.

The seed value for any stream can be changed by calling the *reseed* function.

Prototype: `void stream::reseed(long n)`

Dynamic Example: `s->reseed(24680);`

The second parameter is a positive integer that is to be used as the new seed. Although it is most common to call reseed once for each stream at the beginning of a CSIM program, streams can be reseeded any number of times and at any place in the program.

The current state of a stream can be retrieved by calling the *stream_state* function.

Prototype: `long stream::state()`

Dynamic Example: `i = s->state();`

If *state* is called immediately after reseeding a stream, the seed value will be returned. Otherwise, the positive integer used to produce the random number most recently generated from the stream will be returned.

If a dynamic stream is no longer needed, its storage can be reclaimed as follows:

Dynamic Example: `delete s;`

Once a stream has been deleted, it must not be further referenced.

18.5 Multiple Stream Random Number Generation

The same 18 distributions are available for generating random numbers from multiple streams as are available for generating random numbers from a single stream. The following are two examples:

Multiple Stream Prototype:
```
double stream::uniform
  (double min, double max)
```

Multiple Stream Prototype:
```
double stream::triangular
  (double min, double max,
   double mode)
```

In all other ways, the functions and their parameters are exactly the same. It is the programmer's responsibility to ensure that a stream is used for only one purpose and that a separate stream is used for each application of random numbers in the model.

18.6 Changing the Random Number Generator Function

The default random number generator (RNG) function is a linear congruential generator. A different RNG can be invoked by creating a stream of type stream_prob2.

Example: `set_stream_prob(stream_prob2);`

This stream uses two seeds and is considered to be a “better” RNG. All of the probability distributions will use this RNG, if they are called with str2 as the stream argument.

A stream_prob2 RNG can be reseeded and its state querried as follows:

Example: `void stream::reseed2(long n1, long n2);`

Example: `void stream::state2(long *n1, long *n2);`

A user-provided RNG can be instantiated. This topic is beyond the scope of this manual; additional information can be obtained from Mesquite Software, Inc.

19 Output from CSIM

In order for a simulation model to be useful, output indicating what occurred has to be produced so that it can be analyzed. The following kinds of output can be produced from CSIM:

- Reports

CSIM always collects usage and queueing information on facilities and storage units. In addition, it will collect summary information from tables, qtables, histograms and qhistograms, if any were created by the user. All of this information can be printed via various report statements.

- Model statistics

CSIM collects statistics on the model itself. This information will be printed upon request.

- Status reports

Throughout the execution of the model, CSIM collects information on current status. This information will be printed via various status statements.

If no report statement is specified, CSIM will not generate any output (although the user can generate customized output by gathering data through the various information retrieval statements, doing calculations on it, if desired, and printing it).

19.1 Generating Reports

19.1.1 Partial Reports

A partial report can contain information on just one type of object or just the header.

Prototype: `void report_hdr(void)`

Prototype: `void report_facilities(void)`

Prototype: `void report_storages(void)`

Prototype: `void report_buffers(void)`

Prototype: `void report_classes(void)`

Prototype: `void report_events(void)`

Prototype: `void report_mailboxes(void)`

Prototype: `void report_tables(void)`

Prototype: `void report_qtables(void)`

Prototype: `void report_meters(void)`

Prototype: `void report_boxes(void)`

Where:

- *report_hdr* prints the header of the report
- *report_facilities* prints the usage statistics for all facilities defined in the model
- *report_storages* prints the usage statistics for all storage units defined in the model

- *report_buffers* prints the usage statistics for all buffers defined in the model
- *report_classes* prints the process usage statistics for all process classes defined in the model
- *report_events* prints the usage statistics for all events defined in the model and for which the monitor method has been invoked
- *report_mailboxes* prints the usage statistics for all mailboxes defined in the model and for which the monitor method has been invoked
- *report_tables* prints the summary information for all tables (with histograms and confidence intervals)
- *report_qtables* prints the summary information for all qtables (with histograms and confidence intervals)
- *report_meters* prints the summary information for all meters (with histograms and confidence intervals)
- *report_boxes* prints the summary information for all boxes (with histograms and confidence intervals)

Notes:

- Details of the contents of these reports are in the section 19.2, ”CSIM Report Output.”

19.1.2 Complete Reports

A complete report contains all of the sub-reports.

Prototype: `void report(void)`

Notes:

- The sub-reports appear in the order:
  - report_hdr

- report_facilities
- report_storages
- report_buffers
- report_classes
- report_events
- report_mailboxes
- report_tables
- report_qtables
- report_meters
- report_boxes

- Details of the contents of these reports can be found in section 19.2, ”CSIM Report Output.”

19.1.3 To change the model name:

Prototype: `void set_model_name(char* new_name)`

Example: `set_model_name("prototype system");`

Where:

- *name* - is the new name for the simulation model (quoted string or type char*)

Notes:

- *name* appears as the model name in the report header (in *report_hdr* and *report*).
- Unless changed by this statement, the model name will be “CSIM”.

19.2 CSIM Report Output

The output generated by the *report* statements present information on the simulation run as it has progressed so far. The sub-reports, comprising the overall report are:

- Header
- Report on facility usage (if any facilities were declared)
- Report on storage usage (if any storage units were declared)
- Report on buffer usage (if any buffers were declared)
- Report on the process classes (if more than one process class (the default process class) has been declared)
- Report on the monitored events
- Report on the monitored mailboxes
- Summary for each table (with histogram and confidence interval) declared
- Summary for each qtable (with histogram and confidence interval) declared
- Summary for each meter (with histogram and confidence interval) declared
- Summary for each box (with histogram and confidence interval) declared

The following tables give a complete description of each of these sub-reports.

19.2.1 Report_Hdr Output

Output Heading	Meaning
Revision	CSIM version number
System	System simulation was run on, e.g. SUN Sparc
Model	Model name (see *set_model_name*) statement
Date and time	Date and time that report was printed
Ending Simulation Time	Total simulated time
Elapsed Simulation Time	Simulated time since last reset
CPU Time	Real CPU time used since last report

19.2.2 Report_Facilities Output

Output Heading	Meaning
Facility Summary	
facility name	Name (for a facility set, the index is appended)
service discipline	Service discipline (when one was defined)
service time	Mean service time per request
util	Mean utilization (busy time divided by elapsed time)
throughput	Mean throughput rate (completions per unit time)
queue length	Mean number of requests waiting or in service
response time	Mean time at facility (both waiting and in service)
Counts	
completion count	Number of requests completed

Notes:

- When computing averages based on the number of requests for facilities, the number of completed requests is used. Thus, any requests waiting or in progress when the report is printed do not contribute to these statistics.
- If collection of process class statistics is specified, then the above items are repeated on a separate line for each process class which uses the facility.

19.2.3 Report_Storages Output

Output Heading	Meaning
Storage Summary	
storage name	Name of storage unit
size	Size of storage unit
Means (see note below)	
alloc amount	Mean amount of storage per allocate request
alloc count	Number of allocates
dealloc amount	Mean amount of storage per deallocate requests
util	Average percentage of storage allocated over time
in-que length	Mean time requests are waiting
in-queue time	Average time a request waits

Notes:

- When computing averages based on the number of requests for storage, the number of completed requests is used. Thus, any requests waiting or in progress when the report is printed do not contribute to these statistics.
- If collection of process class statistics is specified, then the above items are repeated on a separate line for each process class which uses the storage.

19.2.4 Report Buffers Output

Output Heading	Meaning
Buffer Summary	
buffer name	Name of storage unit
size	Size of storage unit
Means (see note below)	
get amt	Mean amount of space per get
get qlen	Average queue length for gets
get resp	Average response time for gets
get count	Number of gets
put amt	Mean amount of space per put
put qlen	Average queue length for puts
put resp	Average response time for puts
put count	Number of puts

Note:

- If collection of process class statistics is specified, then the above items are repeated on a separate line for each process class which uses the buffer.

19.2.5 Report_Classes Output

Output Heading	Meaning
id	Process class id
name	Process class name
number	Number of processes belonging to this class
lifetime	Mean simulated time per process in this class
hold ct	Mean number of hold statements per process in this class
hold time	Mean hold time per process in this class
wait time	Mean wait time per process in this class (lifetime - holdtime)

Notes:

- If no process classes are specified, the report for the "default" class (every process begins as a member of this class) is not provided. If any process classes are specified, then the report includes the default class.

19.2.6 Report_Events Output

Output Heading	Meaning
Event Summary	
Event name	Name of event
Numver of queue vst	Total number of entries to queue queue
Avg queue length	Average length of queue queue
Avg time queued	Average time in queue queue
Number of wait vsts	Total number of entries to wait queue
Avg wait length	Average length of wait queue
Avg time waiting	Average time in wait queue
Number of set ops	Number of set operations at event

19.2.7 Report_Mailboxes Output

Output Heading	Meaning
Mailbox Summary	
Mailbox name	Name of mailbox
Number of proc visits	Number of process doing receives
Process qlength	Average process queue length
Process rspTime	Average process response time
Number of messages	Number of message sent
Message qlength	Average number of messages in the mailbox
Message rspTime	Average time for messages in the mailbox

19.2.8 Report_Tables Output

Output Heading	Meaning
Tables (also output by *report_table(t);*)	
minimum	Minimum value recorded
maximum	Maximum value recorded
range	Maximum - minimum
observations	Number of entries in table
mean	Average of values recorded
variance	Variance of values recorded
standard deviation	Square root of variance
coefficient of var.	Standard deviation divided by the mean
Confidence Intervals (also output by *report_table(t*);)	
Observations	Number of observations used to compute interval
Level	Probability that interval contains true mean
Confidence interval	Two forms: Mid-point +/- half-width Lower limit - upper limit
Re. error	Relative error: half-width divided by lower limit

Histograms (also output by *report_table(t);*)	
Lower limit	Low value for this bucket
Frequency	Number of entries in this bucket
Proportion	Fraction of total number of entries that are in this bucket
Cumulative proportion	Fraction of total number of entries that are in this bucket and all lower buckets

19.2.9 Report_Qtables Output

Qtables and Qhistograms (also output by *report_qtable(qt);)*	
Initial	Initial state value
Final	Final state value
Entries	Number of entries to states
Exits	Number of exits from states
Minimum	Minimum state value
Maximum	Maximum state value
Range	Range of state values
Mean	Mean state value (Time-weighted)
Variance	Variance of state values
Standard deviation	Square root of variance
Coeff. of variation	Coefficient of variation: standard deviation divided by mean

Confidence Intervals (also output by *report_qtable(qt)*;)	
Observations	Number of observation used to comput interval
Level	Probability that interval contains true mean
Confidence Interval	Two forms: Mid-point +/- half-width Lower limit - upper limit
Rel. error	Relative error: half-width divided by lower limit
Histograms (also output by *report_qtable(qt);)*	
Lower limit	Low value for this bucket
Frequency	Number of entries in this bucket
Proportion	Fraction of total number of entries that are in this bucket
Cumulative proportion	Fraction of total number of entries that are in this bucket and all lower buckets

Notes:

- All histogram output for qtables is grouped by state value, where each interval except the last includes only one state value. The last bucket contains all state values greater than the value covered by the penultimate value.

19.2.10 Report_Meters Output

Meters (also output by *report_meter(m)*;)	
Count Rate Interpassage time statistics	(see Tables)
Confidence Intervals	(see Tables)
Histograms	(see Tables)

19.2.11 Report_Boxes Output

Boxes (also output by *report_box(b)*;)	
Statistics on elapsed times (see Tables)	
Confidence Intervals	(see Tables)
Histograms	(see Tables)
Statistics on population (see Qtables)	
Confidence Intervals	(see Qtables)
Histograms	(see Qtables)

19.3 Printing Model Statistics

19.3.1 To generate a report on the model statistics:

Example: `mdlstat();`

Notes:

- This report lists:
  - CPU time used
  - Number of events processed
  - Main memory obtained via malloc calls
  - Number of malloc calls
  - Process information:
    - Number of processes started
    - Number of processes saved
    - Number of processes terminated
    - Maximum number of processes active at one time
  - Information about storage for run-time stacks

19.3.2 Events Processed

The number of events (events interval to the operation of the model) can be retrieved as follows:

Prototype: `void events_processed(void)`

19.4 Generating Status Reports

19.4.1 Partial Reports

Prototype: `void status_processes (void)`

Prototype: `void status_next_event_list (void)`

Prototype: `void status_events (void)`

Prototype: `void status_mailboxes (void)`

Prototype: `void status_facilities (void)`

Prototype: `void status_storages (void)`

Prototype: `void status_buffers (void)`

Where:

- *status_processes* prints the status of all processes defined in the model
- *status_next_event_list* prints the pending state changes for processes
- *status_events* prints the status of all events defined in the model
- *status_mailboxes* prints the status for all mailboxes defined in the model
- *status_facilities* prints the status of all facilities defined in the model

- *status_storages* prints the status of all storages defined in the model
- *status_bufferss* prints the status of all bufferss defined in the model
- Details of the contents of these reports are in the sections of this document that discuss their related objects.

19.4.2 Complete Reports

Prototype: `void dump_status (void)`

Notes:

- The sub-reports appear in the order:
  - status_processes
  - status_next_event_list
  - status_events
  - status_mailboxes
  - status_facilities
  - status_storages

Each of the above status statements is callable, so a "customized" status report can be created.

19. Output from CSIM

20 Tracing Simulation Execution

A simulation program, like any other complex software, can be difficult to debug and verify correct. To aid in this, CSIM can produce a log of trace messages during the execution of a simulation. A one-line trace message is produced each time an interesting change in the state of the simulation occurs.

An enormous number of trace messages can be generated by even a short simulation run. For this reason you should try to be selective when enabling different tracing options.

20.1 Tracing All State Changes

The generation of trace messages for all state changes is enabled using the *trace_on* function. The tracing is disabled using the *trace_off* function.

Prototype: `void trace_on (void)`

Example: `trace_on ();`

Prototype: `void trace_off (void)`

Example: `trace_off ();`

Trace messages can be turned on and off as desired during a simulation. Logic can even be added to a simulation to turn on trace messages when a specific condition is detected.

Trace messages can also be enabled by specifying the switch "-T" in the command line that executes the simulation. This feature allows trace messages to be enabled without modifying or recompiling the program. See the documentation for your operating system or programming environment for details on specifying command line switches.

20.2 Tracing a Specific Process

Trace messages that pertain to one specific process or one type of process can be produced using the *trace_process* function. A specific process is identified by a character string consisting of the name that was specified in the call to function *create*, followed by a period and the sequence number of the process. If the period and sequence number are omitted, trace messages for all processes created with that name will be generated.

Prototype: `void trace_process (char* name)`

Example: `trace_process ("customer.100");`

Example: `trace_process ("customer");`

Note that in the first example above there is no guarantee that the 100th process that is created will be an instance of customer. If it is not, no trace messages will be produced. The tracing of a specific process can be disabled by calling function *trace_off*. Successive calls to *trace_process* will change which process is being traced. There is currently no way to specify a list of processes to trace.

20.3 Tracing a Specific Object

Trace messages that pertain to one specific object (*i.e*., a facility, storage, event, or mailbox) can be produced using the *trace_object* function. The object is identified by the character string that was specified when the object was initialized.

Prototype: `void trace_object (char* name)`

Example: `trace_object ("memory");`

Note that the type of the object is not specified. If there is more than one object with the specified name, trace messages for all such objects will be produced. The tracing of a specific object can be disabled by calling function *trace_off*. Successive calls to *trace_object* will change which object is being traced. There is currently no way to specify a list of objects to trace.

20.4 Format of Trace Messages

Each trace message contains the current simulation time, the name and sequence number of the process that caused the state change, and a description of the state change. Sample trace messages are shown below.

```
0.716   customer  4   1   use facility cpu for 0.070
0.716   customer  4   1   reserve facility cpu
0.716   customer  4   1   hold for 0.070
0.716   customer  4   1   sched proc: t = 0.070, id = 4
0.787   customer  4   1   release facility cpu
```

20.5 Program Generated Trace Messages

Any CSIM program can add its own trace messages to the sequence by calling the *trace_msg* function.

Prototype: `void trace_msg (char* string)`

Example: `trace_msg ("entering procedure for");`

Trace messages containing any mixture of text and numeric values can be constructed using the *C sprintf* function. CSIM will prefix the provided string with the current simulation time and the name and sequence number of the process that produced the message.

20.6 What Is and Is Not Traced

Ideally, every occurrence that changes the state of a CSIM object will generate a trace message. In particular, any occurrence that causes time to pass should be traced.

Occurrences that do not produce trace messages include 1) the generation of random numbers, 2) the updating of performance statistics, and 3) the production of reports. Obviously, non-CSIM operations such as updates of local variables can not produce trace messages.

20.7 Redirecting Trace Output

By default, trace messages are written to file *stdout*. Trace messages can be redirected to a different file using the function *set_trace_file*.

Prototype: `void set_trace_file (FILE * file_pointer)`

Example: `*fp = fopen ("trace", "w"); set_trace_file (fp);`

21 MISCELLANEOUS

21.1 Real Time

Although, internally, the model only deals with simulated time, the running of the model takes place in real time.

21.1.1 Retrieve the current real time

Prototype: `char* time_of_day (void)`

Example: `tod = time_of_day ();`

Where:

- tod - is the actual time of day (type char*)

Notes:

- The format of the returned string is:

  day mm dd hh:mm:ss yyyy, for example, Sun Jun 05 13:22:43 1994 for Sunday, June 5, 1994 at 1:22:43 PM

21.1.2 Retrieve the amount of CPU time used by the model

Prototype: `double cputime (void)`

Example: `t = cputime ();`

Where:

- *t* - is the amount of CPU time, in seconds, that has been consumed by the model thus far (type double)

21.2 Retrieving and Setting Limits

There is a maximum number of each kind of CSIM data object in a CSIM program. These maximums can be interrogated and/or changed. The maximums serve as limits on the number of structures of a particular type which exist simultaneously.

21.2.1 Retrieve or change a CSIM maximum

The argument for each of the following statement is interpreted as follows

- Zero – the function returns the current value of the maximum
- Non-zero - the maximum will be changed to the new value

Prototype: `long max_buffers(long new_max)`

Prototype: `long max_classes(long new_max)`

Prototype: `long max_events(long new_max)`

Prototype: `long max_facilities(long new_max)`

Prototype: `long max_histogramslong new_max)`

Prototype: `long max_mailboxes(long new_max)`

Prototype: `long max_messages(long new_max)`

Prototype: `long max_processes(long new_max)`

Prototype: `long max_qtables(long new_max)`

Prototype: `long max_servers(long new_max)`

Prototype: `long max_sizehist(long new_max)`

Prototype: `long max_storages(long new_max)`

Prototype: `long max_tables(long new_max)`

Notes:

- The maximums apply to objects which have been both declared and initialized (and not deleted).
- Since a histogram creates a table, the number of active histograms + active tables cannot exceed the limit for tables.
- Because each mailbox includes an event, the maximum number of events must include at least one event per mailbox. Therefore, if the maximum number of mailboxes is increased, it is likely that the maximum number of events must also be increased.
- It is an error to change the maximum number of classes after a *collect_class_*... statement has been executed.

21.3 Creating a CSIM Program

There are two distinct ways of writing CSIM programs:

- Write a routine named sim() (the standard approach). This will cause CSIM to do the following:
  - Generate the main() routine "under the covers"
  - Perform necessary initialization
  - Process the command line
  - Call sim() with *argc* and *argv* repositioned to point to the non-CSIM arguments
- Provide the main() routine yourself. This allows you to imbed the CSIM model in a surrounding tool. To do this:
  - Call sim() (or any routine) which becomes the first (base) CSIM process when it executes a *create* statement
  - Call *proc_csim_args* to process the CSIM command line arguments (if desired)
  - Call *conclude_csim* when the simulation model part of the program is complete

21.3.1 Process CSIM input parameters from a user-provided main() routine

Prototype: `void proc_csim_args (int * argc, char *** argv)`

Where:

- *argc* and *argv* are the standard C arguments.

Notes:

- On return, any CSIM arguments have been processed (currently the only CSIM argument is *-T* (to turn on tracing) and *argc* and *argv* have been modified to point to any remaining arguments.

21.3.2 Cause CSIM to perform its necessary cleanup when using a user-provided main() routine

Prototype: `void conclude_csim (void)`

Notes:

- If a model is to be rerun, then the *rerun* statement should be executed.

21.4 Rerunning or Resetting a CSIM Model

It may be useful to run a model multiple times with different values, or run multiple models in the same program.

21.4.1 Rerun a CSIM model

Prototype: `void rerun (void)`

Notes:

- *rerun* will cause the following to occur:
  - All non-permanent tables structures are cleared.
  - All processes are eliminated
  - All facilities, events, mailboxes, process classes, storage units, tables and qtables established before the first *create*

statement (the *create* for the first ("sim") process) are reinitialized
- All remaining facilities, storage units, events, etc., are eliminated
- The clock is set to zero

- The following are NOT reset or cleared:
  - The random number generator (issue a *reset_prob(1)* to reset the random number stream)
  - Permanent tables structures
- Special provisions are required for C++ programs with static objects which are constructed before the program begins execution.

21.4.2 Clear statistics without rerunning the model

Prototype: `void reset (void)`

Notes:

- *reset* will cause the following to occur:
  - All statistics for facilities, storage units and buffers are cleared.
  - All non-permanent table structures are cleared
  - The global variable *_start_tm* is set to the current time and is used as the starting point for calculations
- The variable *clock* (the simulation clock) is not altered.
- Time intervals for facilities, storage units and qtables which began before the *reset* are tabulated in their entirety if they end after the *reset*.
- This feature can be used to eliminate the effects of start-up transients.

21.4.3 Conclude_flag

Sometimes, it is necessary to delete some or all of the resources in a model prior to ending a run. In this case, an error can be caused when a resource is deleted – an example would be deleting an event when there are processes in one of the queues. In some cases, the resource should be deleted without causing this error. If the *_conclude_flag* variable is non-zero, such errors will be ignored.

Prototype: `void set_conclude_flag(void)`

21.5 Error Handling

When CSIM detects an error, its default action is to send a message to the error file and then perform a *dump_status*. If this is not satisfactory, the programmer can instead intercept CSIM errors and handle them as desired.

21.5.1 Request that CSIM call a user-specific error handler

Prototype: `void set_err_handler (void (*handler)(long))`

Prototype: `void errHandler(long n);`

Example: `set_err_handler(errHandler);`

Notes:

- The function is called with one argument: the index of the error that was detected (see section 22, “Error Messages,” for a list of errors and their indices).

21.5.2 Request that CSIM revert to the default method of handling errors

Prototype: `void clear_err_handler (void)`

21.5.3 Print the error message corresponding to the index passed to the error handler

Prototype: `void print_csim_error (long error_number)`

Where:

- *index* - is the error index for which the error message should be printed (type long)

Notes:

- The error messages and their indices are listed in section 22, “Error Messages.“

Prototype: `char* csim_eror_msg (long n);`

Example: `printf (“%d: %s/n”, n, csim_err_msg (n);`

Gets string which is error message corresponding to the CSIM error. The error number is made available as the argument to the CSIM error handler procedure.

21.6 Output File Selection

CSIM allows the user to select where various types of output should be sent. The default file for all of these is “stdout”. The following are the files that can be specified:

- Output file - for reports and status dumps
- Error file - for error messages
- Trace file - for traces

21.6.1 Change the file to which a given type of output is sent

Prototype: `void set_error_file (FILE * f)`

Prototype: `void set_output_file (FILE * f)`

Prototype: `void set_trace_file (FILE * f)`

Where:

- *fp* - is a file pointer of the file to which the indicated type of output will be sent (type FILE*)

Notes:

- Type FILE is normally declared in the standard header file <stdio.h>.
- The user is responsible for opening and closing the file.

21.7 Compiling and Running CSIM Programs

A CSIM program has to be compiled referencing the CSIM library to process the required (the "cpp.h" header file) and using the CSIM library (archive file) to satisfy calls to the CSIM library routines.

For information on installing and using CSIM 20 on specific platforms, please see the appropriate installation guide.

21.8 Reminders and Common Errors

When writing a CSIM program, the following things are important:

- Be aware of the maximum allowed number of concurrently active processes. In the current version, there is a limit of 1000 concurrently active processes (this can be changed by using the function *max_processes)*.
- When a process (a procedure containing a *create* statement) is called with parameters, these should be either parameters passed as values (the default in C) or addresses of variables in global (or static) storage. Beware of local arrays and strings which are parameters for processes...they are likely to cause problems. THIS IS VERY IMPORTANT!!

CSIM manages processes by copying the runtime stack to a save area when the process is suspended and then back to the stack when the process resumes. Thus, if a process receives a parameter which is a local address in the initiating process (i.e. in that process's stack frame), the address will not point to the desired value when the called process is executing.

- All entities (facilities, storage units, etc.) must be declared using variables of the correct type.
- All entities (facilities, storage units, etc.) must be initialized before being referenced.
- An array of length n is indexed 0,1,...,n-1 (standard C indexing).

21. Miscellaneous

22 Error Messages

The following error messages can be printed by a CSIM program which detects a problem. With each error message is its index (see section 21.5, “Error Handling,” for the usage of indexes), and a brief interpretation:

1	NEGATIVE EVENT TIME
You tried to schedule an event to occur at a negative time. The probable cause is either a negative hold interval or a program which has truly run away.	
2	EMPTY EVENT LIST
Every active process is waiting for an event to occur, and there is no process which can cause an event to happen (this is a common error) Possible causes for this error are: • A *create* statement was left out of a process • There is a deadlock • There is a subtle error in process synchronization If it is none of these, use the debugging switch(es), to try to find out what was going on when disaster struck.	
3	RELEASE OF IDLE/UNOWNED FACILITY
A process has attempted to release a facility which it did not own.	
4	(not used)

5	PROCESS SHARING TASK LIMIT EXCEEDED
An attempt was made to have more than 100 processes at a facility declared with the prc_shr service function.	
6	NOTE FOUND CURRENT STATE LESS THAN ZERO
You issued either a *note_entry* or a *note_exit* to store a value in a qtable or qhistogram, and the current state (current queue length) was less than zero. One cause of this error is that more *note_exit* statements than *note_entry* statements to have been executed.	
7	ERROR IN DELETE EVENT
The *delete_event* procedure was called and one of the following failures occurred: • The argument was NIL	
8	ERROR IN DELETE MAILBOX
The *delete_mailbox* procedure was called and one of the following failures occurred: • The argument was NIL	
9	MALLOC FAILURE
The UNIX routine named malloc was unable to allocate more memory to the program. Malloc is used to allocate space for process control units, so this usually occurs when many processes are simultaneously active. The only cures are to either have fewer processes or to have the UNIX limits on virtual memory changed on your system.	

10	IN PREEMPT, ERROR IN CANCEL EVENT FOR PROCESS (INTERNAL ERROR)
The processor sharing or last-come, first-served service disciplines have tried to preempt a process which does not hold the facility. This is a CSIM error and should not occur.	
11	ILLEGAL EVENT TYPE (INTERNAL ERROR)
The procedure for creating events has been called with a mode (type) parameter which is not recognizable. This is a CSIM error and should not occur.	
12	TOO MANY EVENTS
The limit on the number of events which can be simultaneously in existence is being exceeded. Either: The program needs more events (see the *max_events* function) or You've created more events than you intended in your program	
13	TOO MANY FACILITIES
The limit on the number of facilities which can be simultaneously in existence is being exceeded. Either: The program needs more facilities (see the *max_facilities* function) You've created more facilities than you intended in your program	
14	TOO MANY HISTOGRAMS
The limit on the number of histograms which can be simultaneously in existence is being exceeded. Either: The program needs more histograms (see the *max_histograms* function) or You've created more histograms than you intended in your program	

15	TOO MANY MAILBOXES
The limit on the number of mailboxes which can be simultaneously in existence is being exceeded. Either: The program needs more mailboxes (see the *max_mailboxes* function) or You've created more mailboxes than you intended in your program	
16	TOO MANY MESSAGES
The limit on the number of messages which can be simultaneously in existence is being exceeded. Either: The program needs more messages (see the *max_messages* function) or You've created more messages than you intended in your program	
17	TOO MANY PROCESSES
The limit on the number of processes which can be simultaneously in existence is being exceeded. Either: The program needs more processes (see the *max_processes* function) or You've created more processes than you intended in your program	
18	CONFIDENCE LEVEL MUST BE BETWEEN 0.0 AND 1.0
In a call to run_length (table or qtable), an illegal value for the confidence level was specified	

19	TOO MANY STORAGES
The limit on the number of storage units which can be simultaneously in existence is being exceeded. Either: The program needs more storage units (see the *max_storages* function) or You've created more storage units than you intended in your program	
20	TOO MANY SERVERS
The limit on the number of servers which can be simultaneously in existence is being exceeded. Either: The program needs more servers (see the *max_servers* function) You've created more servers than you intended in your program	
21	INVALID TIMESTAMP IN CALL TO EXIT_BOX
The value of the parameter in a call to exit_box is either less than zero or greater than the value of the simulated clock	
22	not used
23	not used
24	TRIED TO RETURN AN UNALLOCATED PCB
This is a CSIM error and should not occur.	
25	TRIED TO CHANGE MAXIMUM CLASSES AFTER COLLECT
You cannot change the limit on process classes after a *collect_class_facility [all]* statement.	

26	TOO MANY CLASSES
The limit on the number of classes which can be simultaneously in existence is being exceeded. Either: The program needs more process classes (see the *max_classes* function) or You've created more classes than you intended in your program	
27	IN RETURN EVENT, FOUND WAITING PROCESS
An attempt was made to delete a local event, but a process is waiting for that event. A local event is deleted either by use of a *delete_event* statement or when the process which initialized that event terminates.	
28	TRIED TO DELETE EMPTY EVENT SET
An attempt was made to delete an event_set structure which is not initialized.	
29	TRIED TO WAIT ON NIL EVENT SET
The *wait_any* or *queue_any* function was passed a NIL pointer (argument).	
30	WAIT_ANY ERROR, NIL EVENT
This is an internal error in the *wait_any* or *queue_any* function. The function thinks that an event in the set occurred, but it did not find one. This is a CSIM error and should not occur.	

31	STORAGE DEALLOCATE ERROR: CURRENT COUNT < 0
The *deallocate* procedure has detected a negative value for the current number of users at a storage unit (more *allocate*s than *deallocate*s were done). This is probably the result of having some processes doing a *deallocate* without a prior *allocate* operation. Note that this error can result regardless of the amount of storage allocated and deallocated.	
32	TIMED_RECEIVE ERROR - MSG WAS LOST
There was a failure in *timed_receive*. This is a CSIM error and should not occur.	
33	MULTISERVER FACILITY- ZERO OR NEG. NUMBER OF SERVERS
A multi-server facility was defined with the number of servers less than or equal to zero.	
34	TRIED TO CHANGE MAX_CLASSES AFTER CREATING PROCESS CLASSES
You can't change the maximum number of process classes after a *collect_class_facility* or *collect_class_facility_all* has been executed.	
35	ASKED FOR STATS ON NON-EXISTENT SERVER
You called a function that retrieves information about a server and specified an out-of-range server number.	
36	ERROR IN CALENDAR QUEUE INIT
This is a CSIM error and should not occur.	

37	ERROR IN DELETE FACILITY
The *delete_facility* procedure was called and one of the following failures occurred: The argument was NIL The argument did not point to a facility	
38	ERROR IN DELETE PROCESS CLASS
The *delete_process_class* procedure was called and one of the following failures occurred: The argument was NIL The argument did not point to a process class	
39	ERROR IN DELETE QTABLE
The *delete_qtable* procedure was called and one of the following failures occurred: The argument was NIL The argument did not point to a qtable or qhistogram	
40	ERROR IN DELETE STORAGE
The *delete_storage* procedure was called and one of the following failures occurred: The argument was NIL The argument did not point to a storage unit	
41	ERROR IN DELETE TABLE
The *delete_table* procedure was called and one of the following failures occurred: The argument was NIL The argument did not point to a table or histogram	

42	IN TIMED-, ERROR IN CANCEL EVENT FOR PROCESS (INTERNAL ERROR)
Either *timed_queue*, *timed_receive* or *timed_wait* has tried to cancel a *hold* for a process, and the process cannot be found in the next_event_list. This is a CSIM error and should not occur.	
43	STACK UNWIND FAILURE - HPPA (INTERNAL ERROR)
This is a CSIM error and should not occur.	
44	ODD OR SMALL STACK LENGTH - HPPA (INTERNAL ERROR)
This is a CSIM error and should not occur.	
45	SET_STACK ROUTINES MAY NOT BE INVOKED AFTER CALLING CREATE - HPPA
This is a CSIM error and should not occur.	
46	UNRECOVERABLE STACK OVERFLOW - HPPA
This is a CSIM error and should not occur.	
47	INITIAL STACK SIZE TOO SMALL - HPPA
This is a CSIM error and should not occur.	
48	EMPIRICAL DISTRIBUTION: n EXCEEDS 100
For an empirical distribution, you cannot have more than 100 values.	
49	FACILITY CREATED WITH NUM SERVERS <= 0
Attempted to create a multi-server facility with no servers	
50	(NOT USED)

51	(NOT USED)
52	UNIFORM DISTRIBUTION: MAX < MIN
You called the uniform(min, max) function with max < min	
53	TRIANGULAR DISTRIBUTION: PARAMETER ERROR
In a call to triangular(min, max, mode) either min < max or mode < min or mode > max	
54	GAMMA DISTRIBUTION: PARAMETER ERROR
In a call to gamma(mean, stddev), either mean <= 0.0 or stddev <= 0.0.	
55	EXPONENTIAL DISTRIBUTION: MEAN < 0.0
In a call to exponential(mean), mean <= 0.0	
56	ERLANG2 DISTRIBUTION: PARAMETER ERROR
In a call to erlang2(mean, stddev), either mean <= 0.0 or stddev <= 0.0 or stddev >= mean	
57	HYPERX DISTRIBUTION: VARIANCE <= MEAN*MEAN
In a call to hyperx(mean, var), var <= $mean^2$	
58	BETA DISTRIBUTION: PARAMETER ERROR
In a call to beta(min, max, shape1, shape2), either max <= min or shape1 <= 0.0 or shape2 <= 0.0	
59	WEIBULL DISTRIBUTION: PARAMETER ERROR
In a call to weibull(shape, scale), either shape <= 0.0 or scale <= 0.0	

60	LOGNORMAL DISTRIBUTION: MEAN <= 0.0
In a call to lognormal(mean, stddev), mean <= 0.0	
61	CAUCHY DISTRIBUTION: XBETA <= 0.0
In a call to cauchy(alpha, beta), beta <= 0.0	
62	UNIFORM_INT DISTRIBUTION: MAX < MIN
In a call to uniform_int(min, max), max < min	
63	BERNOULLI DISTR.: PROB. SUCCESS < 0.0 OR > 1.0
In a call to bernoulli(probSuccess), either probSuccess < 0.0 or probSuccess > 1.0	
64	BINOMIAL DISTRIBUTION: PARAMETER ERROR
In a call to binomial(probSuccess, numTrials), either probSuccess <= 0.0 or probSuccess >= 1.0 or numTrials < 1	
65	GEOMETRIC DISTR.: PROB. SUCCESS <= 0.0 OR >= 1.0
In a call to geometric(probSuccess), either probSuccess <= 0.0 or probSuccess >= 1.0	
66	NEG. BINOMIAL DISTRIBUTION: PARAMETER ERROR
In a call to negative_binomial(successNum, probSuccess) either successNum <= 0.0 or probSuccess <= 0.0 or probSuccess >= 1.0	
67	POISSON DISTRIBUTION: MEAN <= 0.0
In a call to poisson(mean), mean <= 0.0	
68	(NOT USED)

69	(NOT USED)
70	BUFFER: ERROR IN DELETE_BUFFER
In a call to delete_buffer(bfr), bfr is NIL	
71	BUFFER: REMOVE_SPACE ERROR
In a call to buffer_remove_space(bft, amt), amt is greater than the amount of space in the buffer	
72	(NOT USED)
73	TOO MANY BUFFERS
Tried to too many buffers; either reduce the number of buffers created or increase max_buffers	
74	STORAGE: REMOVE ERROR
In a call to remove_store(store, size), either size exceeds the amount of storage currently available or the original amount of storage	
75	PARETO DISTRIBUTION: PARAMETER <= 0.0
In a call to pareto(a), a <= 0.0	
76	SETUP_EMPIRICAL – ERROR IN PROB. ARRAY
In a call to setup_empirical, the sum of the probabilities specified is not equal to 1.0	
77	RUN_LENGTH: ACCURACY MUST BE > 0.0 AND < 1.0
In a call to table_run_length or qtable_run_length, either accuracy <= 0.0 or accuracy >= 1.0	

78	RUN_LENGTH: CPU MAX MUST BE > 0.0
In a call to table_run_length or qtable_run_length, the maximum cpu time specification is <= 0.0	
79	NEXT-EVENT-LIST ERROR: TIME GOING BACKWARDS
An error has occurred in the routines that manage the next_event_list; please contact Mesquite Software, Inc.	
80	TRIED TO DELETE EMPTY MAILBOX SET
In a call to delete_mailbox_set, the pointer is NIL	
81	TRIED TO RECEIVE ON EMPTY MAILBOX SET
In a call to receive_any, the pointer is invalid	
82	TIMED_PUT (BUFFER): NEGATIVE TIME-OUT
In a call to buffer_timed_put, the timeout interval is negative	
83	TIMED_GET (BUFFER): NEGATIVE TIME OUT
In a call to buffer_timed_get, the timeout interval is negative	
84	STORAGE: ALLOCATE OR DEALLOCATE NEG. AMT
In a call to allocate, alloc, deallocate, dealloc, timed_allocate or timed_alloc, passed an amount which is negative	

22. Error Messages

23 Acknowledgments

- Teemu Kerola of MCC assisted in the initial implementation of CSIM. He also designed and implemented the MONIT event logging feature and the post-run analysis program for the SUN.
- Bill Alexander of MCC has provided consultation on the wisdom of many proposed features.
- Leonard Cohn of MCC suggested using mailboxes.
- Ed Rafalko, of Eastman Kodak, provided the changes required to have CSIM available on the VMS operating system.
- Rich Lary and Harry Siegler of DEC have provided code for the VMS version of CSIM. They also suggested a number of modifications which have improved the performance of CSIM programs.
- Geoff Brown of Cornell University did most of the work for the HP-300 version. He also provided the note on CSIM on the NeXT System.
- Jeff Brumfield, of The University of Texas at Austin, critiqued many aspects CSIM. He and Kerola suggested process classes.
- Connie Smith, of L & S Systems, did much of the work on the Macintosh version.
- Kevin Wilkinson, of HP Labs, did most of the work on the HP Prism support.

23. Acknowledgments

- Murthy Devarakonda, of IBM T.J. Watson Research Labs, did most of the work on the IBM RS/6000 support.
- Jeff Brumfield provided the ideas, code, and documentation on meters, boxes, confidence intervals, and run length control. He also improved the format of the output reports and added the additional probability distributions.
- Beth Tobias rewrote the CSIM manual.
- Jorge Gonzales helped test and debug CSIM18.
- Dawn Childress revised and reformatted the CSIM18 manuals.
- Randall Alexander provided valuable assistance in managing configurations and in getting CSIM 20 ready to ship.

24 List of References

[Brow88] Brown, R., Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem, *Communications of the ACM*, (31, 10), October, 1988, pp. 1220 - 1227.

[KeSc87] Kerola, T. and H. Schwetman, Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs, *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ACM/SIGMETRICS, May, 1987, pp. 163-174.

[Lake91] Law, A. and D. Kelton, *Simulation Modeling and Analysis,* second edition, (McGraw-Hill, 1991).

[MaMc73] MacDougall, M.H. and J.S. McAlpine, Computer System Simulation with ASPOL, *Symposium on the Simulation of Computer Systems*, ACM/SIGSIM, June, 1973, pp. 93-103.

[MacD74] MacDougall, M.H., Simulating the NASA Mass Data Storage Facility, *Symposium on the Simulation of Computer Systems*, ACM/SIGSIM, June 1974, pp. 33-43.

[MacD75] MacDougall, M.H., Process and Event Control in ASPOL, *Symposium on the Simulation of Computer Systems,* ACM/SIGSIM, August, 1975, pp. 39-51.

24. List of References

[Schw86] Schwetman, H.D., CSIM: A C-Based, Process-Oriented Simulation Language, *Proceedings of the 1986 Winter Simulation Conference,* December, 1986, pp. 387 - 396.

[Schw88] Schwetman, H.D., Using CSIM to Model Complex Systems, *Proceedings of the 1988 Winter Simulation Conference,* December, 1988, pp. 246 - 253; also available as Microelectronics and Computer Technology Corporation, Technical Report ACA-ST-154-88.

[Schw90b] Schwetman, H.D., Introduction to Process-Oriented Simulation and CSIM, *Proceedings of the 1990 Winter Simulation Conference*, December, 1990, pp. 154- 157.

[Schw94] Schwetman, H.D., CSIM17: A Simulation Model-Building Toolkit, Proceedings of the 1994 Winter Simulation Conference, December, 1994. pp. 464-470

[Schw95] Schwetman, H.D., Object-Oriented Simulation Modeling with C++/CSIM17, Proceeding of the 1995 Winter Simulation Conference, December, 1995.

[Schw96] Schwetman, H.D., CSIM18 - The Simulation Engine, Proceedings of the 1996 Winter Simulation Conference, December 1996.

[Schw97] Schwetman, H.D., Data Analysis and Automatic Run Length Control in CSIM18, Proceedings of the 1997 Winter Simulation Conference, Atlanta, GA, Dec. 1997

24. List of References

[Schw98] Schwetman, H.D., Model-Based Systems Analysis Using CSIM18, Proceedings of the 1998 Winter Simulation Conference, Washington, DC, Dec. 1998

[Schw99] Schwetman, H.D., Model, Then Build: A Modern Approach to Systems Development Using CSIM18, Proceedings of the 1999 Winter Simulation Conference, Phoenix, AZ, Dec., 1999

[Schw00a] Schwetman, H.D., Finding the Best System Configuration: An Application of Simulation and Optimization, Proceedings of the 2000 European Simulation Multiconference, Gent, Belgium, May 2000 (Society for Computer Simulation)

[Schw00b] Schwetman, H.D., Optimizing Simulations with CSIM18/OptQuest: Finding the Best Configuration, Proceedings 2000 Winter Simulation Conference, Orlando, FL, Dec., 2000

[Schw01] Schwetman, H.D., CSIM19: A Powerful Tool For Building Systems Models, Proceedings 2001 Winter Simulation Conference, Washington, DC, Dec. 2001

24. List of References

25 Sample Program

A sample CSIM program follows. This program is a model of an M/M/1 queueing system. The process *sim* includes a *for* loop, which generates, at appropriate intervals (exponentially distributed with mean IATM) arriving customers. These customers contend for the facility on a first-come-first-served basis. As each customer gains exclusive use of the facility, they delay for a service period (again exponentially distributed, but with mean SVTM) and then depart. The individual response times (time of arrival to time of departure) are collected in a table. The program also makes use of the histogram feature to collect the frequency distribution of the queue length.

Sample Program to Simulate Single Server Facility

```
// C++/CSIM Model of M/M/1 queue

#include "cpp.h"                 // class definitions
#include <stdio.h>

#define NARS 5000
#define IAR_TM 2.0
#define SRV_TM 1.0

event *done;              // pointer to event done
facility *f;              // pointer to facility f
table *tbl;               // pointer to table of response times
qtable *qtbl;             // pointer to qtable of number in system
int cnt;                        // count of remaining processes
FILE *fp;

void init();
void generateCustomers();
void customer();

extern "C" void sim(int argc, char *argv[])
{
      init();
      create("sim");
```

25. Sample Program

```
        generateCustomers();
        done->wait();// wait for last customer to depart
        report();           // model report
        mdlstat();          // model statistics
}

void generateCustomers()
{
        create("gen");
        for(int i = 1; i <= NARS; i++) {
                hold(exponential(IAR_TM));// interarrival interval
                customer();         // generate next customer
        }
}

void customer()                         // arriving customer
{
        double t1;

        create("cust");
        t1 = clock;         // record start time
        qtbl->note_entry(); // note arrival
        f->reserve();       // reserve facility
        hold(exponential(SRV_TM));// service interval
        f->release();       // release facility
        tbl->record(clock - t1);   // record response time
        qtbl->note_exit();         // note departure
        if(--cnt == 0)
           done->set();     // if last customer, set event done
}

void init()
{
        fp = fopen("csim.out", "w");
        set_output_file(fp);
        set_model_name("M/M/1 Queue");
        done = new event("done");         // instantiate event done
        f = new facility("facility");           // instantiate facility f
        tbl = new table("resp tms");      // instantiate table
        qtbl = new qtable("num in sys")  ;// instantiate qtable
        qtbl->add_histogram(10, 0, 10);  // add histogram to qtable
        cnt = NARS;                // initialize counter
}
```

26 Statements and Reserved Words

26.1 Statement and Reserved Words

Usage	Object	Section
bernoulli(double probSuccesss)	random	18.1
beta(mn,mx,sh1,sh2)	random	18.1
binomial(double probSuccess, long numTrials)	random	18.1
box(char* name)	box	14.1
box::exit (double enterTime);	box	14.2
box::name();	box	14.7
box::number_confidence();	box	14.7
box::number_histogram(long nbkt, long min, long max);	box	14.7
box::number_moving_window(long);	box	14.6
box::number_qtable();	box	14.7
box::number_run_length(double acc, double confLev, double maxTime);	box	14.5
box::report();	box	14.3
box::reset();	box	14.9
box::set_name (const char* name);	box	14.6
box::summary();	box	14.3
box::time_confidence();	box	14.7

box::time_histogram(long nbkt, double xmin, double xmax);	box	14.4
box::time_moving_window(long);	box	14.6
box::time_run_length(double acc, double confLev, double maxTime);	box	14.5
box::time_table();	box	14.7
buffer(char *name, long size)	buffer	6.1
buffer::add_space(const long amt);	buffer	6.7
buffer::class_get_count()	buffer	6.7
buffer::class_get_time()	buffer	6.11
buffer::class_get_total()	buffer	6.11
buffer::class_put_count()	buffer	6.11
buffer::class_put_time()	buffer	6.11
buffer::class_put_total()	buffer	6.11
buffer::collect_class_buffer	buffer	6.10
buffer::current();	buffer	6.1
buffer::get(const long amt);	buffer	6.3
buffer::get_count(BUFFER b);	buffer	6.7
buffer::get_current_count();	buffer	6.7
buffer::get_first_process();	buffer	9.4
buffer::get_insert_process(process_t p);	buffer	9.4
buffer::get_last_process();	buffer	9.4
buffer::get_remove_process(process_t p);	buffer	9.4
buffer::get_timeQueue();	buffer	6.11
buffer::get_total();	buffer	6.11
buffer::name();	buffer	6.11
buffer::put(const long amt);	buffer	6.2
buffer::put_count();	buffer	6.11
buffer::put_current_count();	buffer	6.11

buffer::put_first_process();	buffer	9.4
buffer::put_insert_process(process_t p);	buffer	9.4
buffer::put_last_process();	buffer	9.4
buffer::put_remove_process(process_t p);	buffer	9.4
buffer::put_total();	buffer	6.11
buffer::remove_space(const long amt);	buffer	6.8
buffer::reset();	buffer	6.5
buffer::size();	buffer	6.11
buffer::timed_get(const long amt, const double t);	buffer	6.6
buffer::timed_put(const long amt, const double t);	buffer	6.6
buffer_put_timeQueue(BUFFER b);	buffer	6.11
cauchy(double alpha, double beta)	random	18.1
clear_err_handler();	utility	21.5
clock()	utility	21.5
collect_class_buffer_all()	buffer	6.10
collect_class_facility_all();	facility	4.13
collect_class_storage_all()	storage	5.12
conclude_csim();	utility	21.3
cputime();	utility	21.1
create(const char*);	process	3.2
csim_error_msg(long msgNumber);	utility	21.5
csim_exit(long);	utility	21.3
csim_identity();	process	3.6
csim_priority();	process	3.6
csim_random(x,y)	random	18.1
csim_random_int(long mn, long mx)	random	18.1
csim_rerun();	utility	21.4

csim_reset();	utility	21.4
csim_terminate();	utility	3.4
current_class();	process_class	17.2
dump_status();	utility	19.4
empirical(n,cut,als,val)	random	18.1
enter_box(BOX);	box	14.2
erlang(mean,var)	random	18.1
erlang2(mean,var)	random	18.1
erlangn(mean,n)	random	18.1
event(name)	event	7.1
event::clear()	event	7.7
event::clear();	event	7.7
event::first_queue_process();	event	9.5
event::first_wait_process();	event	9.5
event::insert_queue_process(process_t p);	event	9.5
event::insert_wait_process(process_t p);	event	9.5
event::last_queue_process();	event	9.5
event::last_wait_process();	event	9.5
event::monitor();	event	7.1
event::name();	event	7.13
event::qlen();	event	7.13
event::queue()	event	7.4
event::queue();	event	7.4
event::queue_cnt();	event	7.13
event::queue_count();	event	7.13
event::queue_delay_count();	event	7.13
event::queue_length();	event	7.13
event::queue_sum();	event	7.13

event::queue_time();	event	7.13
event::remove_queue_process(process_t p);	event	9.5
event::remove_wait_process(process_t p);	event	9.5
event::reset();	event	7.11
event::set()	event	7.6
event::set();	event	7.6
event::set_name (const char* name);	event	7.8
event::state()	event	7.13
event::state();	event	7.13
event::timed_queue(double);	event	7.5
event::timed_queue_any(double);	event	7.12
event::timed_wait(double);	event	7.3
event::wait()	event	7.2
event::wait();	event	7.2
event::wait_cnt();	event	7.13
event::wait_count();	event	7.13
event::wait_delay_count();	event	7.13
event::wait_length();	event	7.13
event::wait_sum();	event	7.13
event::wait_time();	event	7.13
event_set(char* name, long numEvents)	event_set	7.1
event_set::count();	event_set_set	7.13
event_set::monitor();	event_set	7.1
event_set::name();	event_set	7.13
event_set::num_events(EVENT*);	event_set	7.13
event_set::queue_any();	event_set	7.12
event_set::timed_wait_any(double);	event_set	7.12
event_set::wait_any();	event_set	7.13

events_processed();	utility	19.3.2
exponential(double mean)	random	18.1
exponential(double mn)	random	18.5
facility(double name)	facility	4.1
facility::class_completions(process_class*);	facility	4.14
facility::class_qlen(PROCESS_CLASS);	facility	4.14
facility::class_qlength(process_class);	facility	4.14
facility::class_resp(process_class);	facility	4.14
facility::class_serv(process_class);	facility	4.14
facility::class_tput(process_class);	facility	4.14
facility::class_util(process_class);	facility	4.14
facility::collect_class_facility();	facility	4.13
facility::completions();	facility	4.14
facility::first_process();	facility	9.2
facility::insert_process(process_t p);	facility	9.2
facility::last_process();	facility	9.2
facility::name();	facility	4.14
facility::num_busy();	facility	4.14
facility::num_servers();	facility	4.14
facility::preempts();	facility	4.14
facility::qlen();	facility	4.14
facility::qlength();	facility	4.14
facility::release();	facility	4.3
facility::release_server(long srvrIndex);	facility	4.6
facility::remove_process(process_t p);	facility	9.2
facility::reserve();	facility	4.3
facility::resp();	facility	4.11
facility::serv();	facility	4.14

facility::server_completions(long srvrIdx);	facility	4.14
facility::server_serv(long srvrIdx);	facility	4.14
facility::server_service_time_remaining()	facility	4.14
facility::server_stats_off()	facility	4.4
facility::server_stats_on()	facility	4.4
facility::server_tput(long srvrIdx);	facility	4.14
facility::server_util(long srvrIdx);	facility	4.14
facility::service_disp();	facility	4.14
facility::set_loaddep(double* v, long n);	facility	4.11
facility::set_name(const char* name);	facility	4.1
facility::set_servicefunc(SF);	facility	4.11
facility::set_timeslice(double ts);	facility	4.11
facility::status();	facility	4.14
facility::synchronous(double, double);	facility	4.11
facility::timed_reserve(double);	facility	4.9
facility::timeslice();	facility	4.11
facility::tput();	facility	4.14
facility::use(double t);	facility	4.2
facility::util();	facility	4.14
facility_ms(char* name, long numSrvrs)	facility_ms	4.7
facility_set(char* name, long num)	facility_set	4.8
faciolity::reset_facility();	facility	4.5
gamma(double mean,double stddev)	random	18.1
geometric(probSuccess)	random	18.1
global_event(name)	global_event	7.1
global_mailbox(name)	global_mailbox	8.1
hold(double holdTime);	process	3.3
hyperx(double mean, double var)	random	18.1

hypoexponential(double mn, double var)	random	18.1
identity()	process	3.6
lognormal(double mean, double stddev)	random	18.1
mailbox(char* name)	mailbox	8.1
mailbox::first_msg();	mailbox	9.6
mailbox::first_process();	mailbox	9.6
mailbox::insert_msg(message_t m);	mailbox	9.6
mailbox::insert_process(process_t p);	mailbox	9.6
mailbox::last_msg(message_t);	mailbox	9.6
mailbox::last_process();	mailbox	9.6
mailbox::monitor();	mailbox	8.7
mailbox::msg_cnt();	mailbox	8.12
mailbox::msg_count();	mailbox	8.1
mailbox::msg_delay_count();	mailbox	8.12
mailbox::msg_length();	mailbox	8.12
mailbox::msg_sum();	mailbox	8.12
mailbox::msg_time();	mailbox	8.12
mailbox::proc_count();	mailbox	8.12
mailbox::proc_delay_count();	mailbox	8.12
mailbox::proc_length();	mailbox	8.12
mailbox::proc_sum();	mailbox	8.12
mailbox::proc_time();	mailbox	8.12
mailbox::queue_cnt();	mailbox	8.12
mailbox::receive(long msg)	mailbox	8.3
mailbox::receive(long* msg);	mailbox	8.3
mailbox::remove_msg(message_t msg);	mailbox	9.6
mailbox::remove_process(process_t p);	mailbox	9.6
mailbox::reset();	mailbox	8.8

mailbox::send(long msg)	mailbox	8.2
mailbox::send(long msg);	mailbox	6.2
mailbox::set_name(const char* name);	mailbox	8.10
mailbox::synchronous_send(long msg);	mailbox	8.5
mailbox::timed_receive(long*, double);	mailbox	8.4
mailbox::timed_synchronous_send(long, double);	mailbox	8.6
mailbox_::name();	mailbox	8.12
mailbox_set(arr,name,n)	mailbox_set	8.1
mailbox_set::monitor();	mailbox_set	8.7
mailbox_set::name();	mailbox_set	8.10
mailbox_set::num_msgs();	mailbox_set	8.12
mailbox_set::receive_any(long* msg);	mailbox_set	8.9
mailbox_set::timed_receive_any(long*, double);	mailbox_set	8.9
max_buffers(long);	utility	21.2
max_classes(long);	utility	21.2
max_events(long);	utility	21.2
max_facilities(long);	utility	21.2
max_histograms(long);	utility	21.2
max_mailboxes(long);	utility	21.2
max_messages(long);	utility	21.2
max_processes(long);	utility	21.2
max_qtables(long);	utility	21.2
max_servers(long);	utility	21.2
max_sizehist(long);	utility	21.2
max_storages(long);	utility	21.2
max_tables(long);	utility	21.2
mdlstat();	utility	19.3

message::content();	message	9.6
message::next_msg();	message	9.6
meter(char* name)	meter	13.1
meter::cnt(METER);	meter	13.7
meter::confidence(METER);	meter	13.5
meter::histogram(long nbkts, double min, double max);	meter	13.4
meter::ip_table();	meter	13.7
meter::name();	meter	13.7
meter::note_passage();	meter	13.2
meter::rate();	meter	13.7
meter::report();	meter	13.3
meter::reset();	meter	13.9
meter::run_length(double acc, double conLev, double maxTime);	meter	13.5
meter::set_name(const char*name);	meter	13.8
meter::start_time();	meter	13.7
meter::summary();	meter	13.7
negative_binomial(long sn, double ps)	random	18.1
normal(double mean, double stddev)	random	18.1
normal2(double mean, double stddev)	random	18.1
pareto(double a)	random	18.1
permanent_table(name)	permanent_table	11.1
poisson(double mean)	random	18.1
print_csim_error(long msgNum);	utility	21.5
priority()	process	3.6
proc_csim_args(int* argc, char** argv[]);	utility	21.3
process::get_struct();	process	9.1
process::identity();	process	9.1

process::name();	process	9.1
process::name();	process	3.6
process::next_process();	process	9.1
process::priority();	process	9.1
process::process_process_class()	process	9.1
process::restart();	process	9.1
process::set_priority(long pr);	process	9.1
process::set_struct(void* strct);	process	9.1
process::this_process_class()	process	9.1
process_class(name)	process_class	17.1
process_class::cnt();	process_class	17.7
process_class::holdcnt();	process_class	17.7
process_class::holdtime();	process_class	17.7
process_class::id();	process_class	17.7
process_class::lifetime();	process_class	17.7
process_class::name();	process_class	17.7
process_class::reset();	process_class	17.5
process_class::set_name(const char* name);	process_class	17.4
process_class::set_process();	process_class	17.2
qtable(name)	qtable	12.1
qtable::batch_count();	qtable	12.7
qtable::batch_size();	qtable	12.7
qtable::cnt();	qtable	12.7
qtable::conf_accuracy(double confLev);	qtable	12.7
qtable::conf_halfwidth(double confLev);	qtable	12.7
qtable::conf_lower(double confLev);	qtable	12.7
qtable::conf_mean();	qtable	12.7
qtable::confLev conf_upper(double confLev);	qtable	12.7

qtable::confLev confidence();	qtable	12.5
qtable::confLev converged();	qtable	12.5
qtable::cur();	qtable	12.7
qtable::current();	qtable	12.7
qtable::cv();	qtable	12.7
qtable::dbl_current();	qtable	12.7
qtable::dbl_entries();	qtable	12.7
qtable::dbl_exits(Q);	qtable	12.7
qtable::dbl_initial();	qtable	12.7
qtable::dbl_max();	qtable	12.7
qtable::dbl_min();	qtable	12.7
qtable::dbl_range();	qtable	12.7
qtable::entries();	qtable	12.7
qtable::exits();	qtable	12.7
qtable::histogram(long nbkts, long min, long max);	qtable	12.4
qtable::histogram_bucket(long i);	qtable	12.7
qtable::histogram_high();	qtable	12.7
qtable::histogram_low();	qtable	12.7
qtable::histogram_num();	qtable	12.7
qtable::histogram_total(Q);	qtable	12.7
qtable::histogram_width();	qtable	12.7
qtable::initial();	qtable	12.7
qtable::max(Q);	qtable	12.7
qtable::mean();	qtable	12.7
qtable::min();	qtable	12.7
qtable::moving_window(long numEntries);	qtable	12.6
qtable::name();	qtable	12.7

qtable::note_entry();	qtable	12.2
qtable::note_exit();	qtable	12.2
qtable::note_state(long st);	qtable	12.2
qtable::note_value(long v);	qtable	12.2
qtable::range();	qtable	12.7
qtable::report ();	qtable	12.3
qtable::report_dbl();	qtable	12.3
qtable::reset();	qtable	12.9
qtable::run_length(double accur, double confidLev, double maxTime);	qtable	12.7
qtable::set_name(const char* name);	qtable	12.8
qtable::state();	qtable	12.7
qtable::stddev();	qtable	12.7
qtable::sum();	qtable	12.7
qtable::sum_square();	qtable	12.7
qtable::summary();	qtable	12.3
qtable::var();	qtable	12.7
qtable::window_size();	qtable	12.6
qtabls::set_moving(long numEntries);	qtable	12.6
random(mn,mx)	random	18.1
report();	utility	19.1
report_boxes();	box	14.3
report_buffers();	buffer	6.4
report_classes();	process_class	17.3
report_events();	event	17.10
report_facilities();	facility	4.4
report_hdr();	utility	19.1
report_mailboxes());	mailbox	8.7

report_meters();	meter	13.3
report_qtables();	qtable	12.3
report_storages();	storage	5.4
report_table(TABLE);	table	11.3
report_tables();	table	11.3
rerun()	utility	21.4
reset()	utility	21.4
reset_boxes();	box	14.9
reset_buffers();	buffer	6.5
reset_events();	event	7.11
reset_facilities();	facility	4.5
reset_mailboxes();	mailbox	8.8
reset_meters();	meter	13.9
reset_process_classes();	process_class	17.5
reset_qtables();	qtable	12.9
reset_storages();	storage	5.5
reset_tables();	table	11.9
set_conclude_flag();	utility	21.4.3
set_err_handler(EH);	utility	21.5
set_error_file(FILE*);	utility	21.6
set_initial_value(QTABLE, double);	qtable	12.1
set_input_file(FILE*);	utility	21.6
set_model_name(const char*);	utility	19.1
set_output_file(FILE*);	utility	21.6
set_priority(long pr);	process	3.5
set_this_struct(void *);	process	9.1
set_trace_file(FILE*);	utility	21.6
setup_empirical(const long, const double*,	random	18.1

double*, long*);		
simtime();	utility	2.3
status()	facility	4.14
status_buffers();	buffer	6.1
status_events();	event	7.14
status_facilities();	facility	4.15
status_mailboxes();	mailbox	8.13
status_next_event_list();	utility	19.4
status_processes();	process	3.7
status_storages();	storage	5.13
storage(char *name, long amt)	storage	5.1
storage::busy_amt();	storage	5.12
storage::capacity();	storage	5.12
storage::class_busy_amt()	storage	5.12
storage::class_release_cnt()	storage	5.12
storage::class_request_cnt()	storage	5.12
storage::class_request_total()	storage	5.12
storage::class_waiting_amt()	storage	5.12
storage::collect_class_storage()	storage	5.12
storage::dealloc(long);	storage	5.3
storage::deallocate(long);	storage	5.3
storage::first_process();	storage	9.3
storage::insert_process(process_t p);	storage	9.3
storage::last_process();	storage	9.3
storage::name();	storage	5.12
storage::number_amt();	storage	5.12
storage::qlength();	storage	5.12
storage::queue_cnt();	storage	5.12

storage::release_cnt();	storage	5.12
storage::release_total();	storage	5.12
storage::remove_process(process_t p);	storage	9.3
storage::remove_store(long);	storage	5.9
storage::request_amt();	storage	5.12
storage::request_cnt();	storage	5.12
storage::request_total();	storage	5.12
storage::reset();	storage	5.5
storage::set_name(const char* name);	storage	5.1
storage::synchronous(double, double);	storage	5.8
storage::time();	storage	5.12
storage::timed_alloc(long, double);	storage	5.7
storage::timed_allocate(long, double);	storage	5.7
storage::waiting_amt();	storage	5.12
storage_set(char * name, long size, long n)	storage_set	5.6
storeage::add_store(long amount);	storage	5.9
storeage::alloc(long amount);	storage	5.2
storeage::allocate(long amount);	storage	5.2
storeage::avail();	storage	5.12
stream()	random	18.3
stream::bernoulli(double);	random	18.5
stream::beta(double, double, double, double);	random	18.5
stream::binomial(double, long);	random	18.5
stream::cauchy(double, double);	random	18.5
stream::empirical(const long, double*, long*, const double*);	random	18.5
stream::erlang(double, double);	random	18.5
stream::erlang2(double, double);	random	18.5

stream::erlangn(double, long);	random	18.5
stream::expntl(double mn)	random	18.5
stream::exponential(double);	random	18.5
stream::gamma(double, double);	random	18.5
stream::geometric(double);	random	18.5
stream::hyperx(double, double);	random	18.5
stream::hypoexponential(double, double);	random	18.5
stream::lognormal(double, double);	random	18.5
stream::normal2(double, double);	random	18.5
stream::pareto(double);	random	18.5
stream::poisson(double);	random	18.5
stream::prob();	random	18.5
stream::random_int(double mn, double mx)	random	18.5
stream::reseed(long);	random	18.2
stream::reseed2(long, long)	random	18.2
stream::set_stream_prob((void*)func())	random	18.6
stream::state()	random	18.2
stream::state2(long *, long*)	random	18.2
stream::triangular(double, double, double);	random	18.5
stream::uniform(double, double);	random	18.5
stream::uniform_int(long, long);	random	18.5
stream::uniform01();	random	18.5
stream::weibull(double, double);	random	18.5
stream::zipf(long);	random	18.5
stream::zipf_sum(long, double *);	random	18.5
stream_empirical(NIL,n,cut,als,val)	random	18.5
stream_negative_binomial(NIL,sn,ps)	random	18.5
stream_normal(NIL,mean,stddev)	random	18.5

stream_prob(void*);	random	18.5
table(name)	table	11.1
table::add_histogram(long, double, double);	table	11.7
table::batch_count();	table	11.7
table::batch_size();	table	11.7
table::cnt();	table	11.7
table::compute_confidence_statistics(double confLev, double *mn,double *hw, double *rel)	table	11.7
table::conf_accuracy(double confLev);	table	11.7
table::conf_halfwidth(double confLev);	table	11.7
table::conf_lower(double confLev);	table	11.7
table::conf_mean();	table	11.7
table::conf_upper(double);	table	11.7
table::confidence();	table	11.7
table::converged();	table	11.7
table::cv();	table	11.7
table::hist();	table	11.7
table::histogram_bucket(long);	table	11.7
table::histogram_high();	table	11.7
table::histogram_low();	table	11.7
table::histogram_num();	table	11.7
table::histogram_total();	table	11.7
table::histogram_width();	table	11.7
table::max();	table	11.7
table::mean();	table	11.7
table::min();	table	11.7
table::moving_window(long);	table	11.7
table::name();	table	11.7

table::range();	table	11.7
table::record(double val);	table	11.2
table::record_value(double val);	qtable	12.2
table::reset();	table	11.9
table::run_length(double, double, double);	table	11.5
table::set_moving_table(long numEntries);	table	11.6
table::set_name(const char* name);	table	11.8
table::stddev();	table	11.7
table::sum();	table	11.7
table::sum_square();	table	11.7
table::summary();	table	11.7
table::tabulate(double);	table	11.2
table::var();	table	11.7
table::window_size();	table	11.6
terminate()	process	3.4
this_process();	process	9.1
this_struct();	process	9.1
time_of_day();	utility	21.1
trace_msg(char *);	utility	20.5
trace_object(char *);	utility	20.3
trace_off();	utility	20.1
trace_on();	utility	20.1
trace_process(const char *);	utility	20.2
triangular(double mn, double mx, double mode)	random	18.1
uniform(double mn, double mx)	random	18.1
uniform_int(double mn, double mx)	random	18.1
uniform01()	random	18.1

weibull(double shape, double scale)	random	18.1
zipf(long n)	random	18.1
zipf_sum(long n, long sum)	random	18.1

26.2 Facility Service Disciplines

fcfs	first-come, first-served
fcfs_sy	first-come, first-served synchronous
inf_srv	infinite server
lcfs_pr	last-come, first-served priority
prc_shr	processor shared
pre_res	preempt resume
pre_rst	preempt restart
rnd_pri	round robin timeslice priority
rnd_rob	round robin timeslice

26.3 Data Structures (not used in C++ Version)

BUFFER	used to define a buffer
CLASS	used to define a process class
EVENT	used to define an event or event_set
FACILITY	used to define a facility or facility_set
HIST	used to define a histogram
MBOX	used to define a mailbox
QHIST	used to define a qhistogram
QTABLE	used to define a qtable
STORE	used to define a storage
STREAM	used to define a stream of random numbers
TABLE	used to define a table
TIME	used to define time variables (double precision)

26.4 Constant Values

BUSY FREE	status of facility
NIL	0
OCC NOT_OCC	status of event (occurred)
EVENT_OCCURRED TIMED_OUT	value of timed_operation
MAXCLASSES	default maximum number of process classes
MAXEVNTS	default maximum number of events
MAXFACS	default maximum number of facilities
MAXHISTS	default maximum number of historgrams
MAXMBOXS	default maximum number of mailboxes
MAXMSGS	default maximum number of messages
MAXPROCS	default maximum number of processes
MAXQTBLS	default maximum number of queue histograms
MAXSTORS	default maximum number of storage units
MAXSERVS	default maximum number of server/facility
MAXSIZEH	default maximum size of a histogram
MAXTBLS	default maximum number of tables

26.5 Special Structures

default_class	class that all process belong to initially

26.6 Compatibility with CSIM 19 Programs

CSIM 20 supports all CSIM 19 functionality and will run CSIM 19 programs without modification.

26. Reserved Words