June 1, 1992

CSIM† Users' Guide
(for use with CSIM Revision 16)

Herb Schwetman

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, TX 78759
(512) 338-3428

INTRODUCTION

CSIM is a process-oriented discrete-event simulation package for use with C or C++ programs. It is implemented as a library of routines which implement all of the necessary operations. The end result is a convenient tool which programmers can use to create simulation programs. The "CSIM Reference Manual" [Schw90] has a description of all of the CSIM procedures and functions, along with a sample CSIM program.

This Users' Guide is a revision of the Reference Manual. The primary difference is that this guide is organized around the structures which are used to implement CSIM models. Each section describes a simulation structure and then gives the routines which manipulate that structure. The Reference Manual is organized by statement types. The CSIM programmer is welcome to choose the organization which suits his/her needs.

A CSIM program models a system as a collection of CSIM structures and CSIM processes which interact with each other by visiting the structures. The model maintains a notion of simulated time, so that the model can yield insight into the dynamic behavior of the modeled system. Simulated time passes, in such a model, only when processes execute *hold* statements.

This document first gives a consistent description of the CSIM structures which are provided. Then it gives the statements required to initiate and control CSIM processes. A section on reports is followed by final section which gives information on miscellaneous features of the CSIM library and how to compile, execute and debug CSIM programs.

CSIM STRUCTURES

Every CSIM structure is implemented in the same manner. For each structure, the program must have a *declaration*, which is a pointer to an instance of a structure. Before a structure can be used, it must be initialized, by the *initialize* function for that kind of structure. Finally, some kinds of structures can be deleted as the model executes; for these, there is a *delete* function. These serve the same functions as object declarations, constructors and destructors in an object-oriented language such as C++.

The structures provided in CSIM are as follows:

- Facilities - consisting of servers reserved or used by processes,
- Storages - resources which can be partially allocated to processes,
- Events - used to synchronize process activities,
- Mailboxes - used for interprocess communications,
- Tables, etc. - used to collect data during the execution of a model, and
- Process classes - used to segregate statistics for reporting purposes.

It is important to remember that all of these structures are used in some manner by CSIM processes. It is the processes which mimic the behavior of active entities in the simulated system.

FACILITIES

A facility is normally used to model a resource 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 can at a time can hold a server. A multi-server facility contains a single queue and multiple servers. Here, all of the waiting processes wait 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 servers, 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-servered. Other service disciplines (other than priority order) can be established for a server. These are described below and include round-robin with timeslice and preempt-resume.

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* or a *report_facilities* is executed. In addition, a set of functions can be used to extract individual statistics for each facility.

Declarations and Initializers

FACILITY f;
f = facility("name");
f = facility_ms("name", ns);
declare and initialize f (a variable of type FACILITY) to be a facility of name “name”; in the first form, a facility with one server is created; with the second form, a multi-server with *ns* servers is created; facilities should be declared with global variables and initialized in the sim (main) process, prior to the beginning of the simulation part of the model; a variety of scheduling policies can be invoked at a facility by using the *set_servicefunc* and the *use* statements described below; the default policy is first-come, first-served.

FACILITY arr[n];
facility_set(arr, "name", n);
declare and initialize a set of n single server facilities; the i-th facility is named “name[i]”; the pointer to the i-th facility is stored at arr[i], where arr is an array of variables of type FACILITY and is of length n; the facilities in the set are indexed 0 through n-1.

set_name_facility(f, name);
change the name of the facility specified by f to the string "name".

collect_class_facility(f);
cause process class statistics to be collected for the facility specified by f; usage of the facility by processes in all of the process classes which visit the facility is reported in the facilities report.

collect_class_facility_all();
causes process class statistics to be collected for all of the facilities in existence when this statement is executed; usage of each facility by processes in all of the process classes is reported in the facilities report.

delete_facility(f);
delete_facility_set(arr);
delete the specified facility or facility_set; this is an extreme action and should be used only when really needed.

Operations

i = reserve(f);
if a server at the facility f is available, then it is reserved for the process which continues execution; if all servers at the facility are already reserved by other processes, then the reserving process is suspended until a server at the facility is released; suspended processes are granted access to f one-at-a-time in the order dictated by the priorities; in case of a tie in priorities, the most recent process goes behind processes which arrived earlier, making first-come, first-served the "default" scheduling rule; if the facility has more than one server, then the first available (free) server is assigned to the process; f must be of type FACILITY and initialized with a *facility*, *facility_ms* or *facility_set* statement; the *use* statement (see below) provides another way of "using" a facility; with the *use* statement, the order of process selection at the facility can be governed by any one of several scheduling disciplines; the value of the reserve() function is the index of the server assigned to the process.

release(f);
the facility f is released; if there are processes waiting for access to f, the process of highest priority is given access to the facility and then restarted; f must be of type FACILITY and initialized with a *facility*, *facility_ms* or *facility_set* statement; the process which reserved this facility must also be the process doing the release.

release_server(f, i);
releases the server whose index is i at the facility specified by f; i normally has the value returned by the reserve function (see above); in contrast to the release procedure, the ownership of the server is not checked; a process which did not reserve the facility may release it using the release_server statement with a server index.

use(f, t);
uses facility f for time interval t (expressed as a number of type float); the service discipline function currently established for this facility (see *set_servicefunc* below) is invoked when the *use* statement is executed; f must be of type FACILITY and initialized with a *facility*, *facility_ms* or *facility_set* statement; a *use* is similar to a *reserve*, followed by a *hold* and a *release* , in that the *use* also acquires a server, allows simulated time to pass and then releases the server; the differences are that with a *use*, these all become one atomic operation and that one of a variety of service disciplines can be selected.

States and Status

st = status(f);
st is set to BUSY if all servers at facility f are reserved; st is set to FREE if at least one server at facility f is available; BUSY (= 1) and FREE (= 0) are constants defined in the header file "csim.h".

i = qlength(f);
the value of this function is of type int; it is set to the number of processes currently waiting for access to the facility f.

i = num_busy(f);
is set to the number of busy servers at the facility f, where f is a facility; notice that num_busy(f)+qlength(f) is the number of processes at the facility f.

Service Disciplines

set_servicefunc(f, func);
this statement allows the programmer to change the default service discipline at facility f; fcfs (first-come, first-served) is the default service discipline; set_servicefunc change this default to some other service discipline, which is specified by func; func is the function which is invoked when the *use* statement (described above) references this facility; the purpose of this feature is to allow the modeling of different service disciplines which govern use of the facility; any of the predefined service discipline functions described below can be used as func; in addition, the programmer can supply a new service function (the interested

programmer should look at the implementations of the service discipline functions mentioned below to see how this is done).

Note: the *use* statement (as opposed to the *reserve*) statment must be used for the selected service discipline to be effective. The service discipline functions are as follows:

fcfs	first-come-first-served
inf_srv	infinite servers (no queueing delay)
lcfs_pr	last come, first served preempt
pre_res	preempt resume (based on process priority)
prc_shr	processor sharing (load dependent service function - see set_loaddep below)
rnd_rob	round robin with time slice (see set_timeslice below)

set_timeslice(f, t);
set the time slice value for facility f to t (expressed as a number of type float); this time slice is used only with the round robin service discipline function.

t = timeslice(f);
the value of this function (of type float) is set to the current value of the timeslice for facility f.

set_loaddep(f, rate, n);
set the load dependent service rate function for facility f to the n values found in the array named rate (with elements of type float); this function is used by the *prc_shr* service discipline to control the time each job uses the facility; the default load dependent rate function has rate[i] equal to i; the entries in the rate array will be used to alter the service times of jobs using the facility; when i tasks are in service, then the service time of each job will be multiplied by rate[i]; the altered service times are recomputed as tasks arrive at and leave the facility; if n is less than the constant NTASKS (currently defined as 20 in the header file "csimdef.h"), then the last value of rate is replicated until NTASKS values are available; if n is greater than NTASKS-1, then only NTASKS values are used.

Retrieving Informations and Data from Facilities and Servers

These functions each return a statistic which describes some aspect of the usage of the specified facility. The type of the returned value for each of these functions is as indicated, where an *x* is type float, *n* is type int, and is type char*.

a = facility_name(f);	returns pointer to name of facility f;
n = num_servers(f);	returns number of servers at facility f;
a = service_disp(f);	returns pointer to name of service discipline at f;

n = completions(f);	returns number of completions at facility f;
n = preempts(f);	returns number of preempted requests at f;
x = qlen(f);	returns mean queue length at facility f;
x = resp(f);	returns mean response time at facility f;
x = serv(f);	returns mean service time at facility f;
x = tput(f);	returns mean throughput rate at facility f;
x = util(f);	returns utilization at facility f.

n = server_completions(f, i);	returns number of completions for server i at f;
x = server_serv(f, i);	returns service time for server i at facility f;
x = server_tput(f, i);	returns throughput rate for server i at facility f;
x = server_util(f, i);	returns utilization for server i at facility f;

n = class_completions(f, c);	returns number of completions for class c at f;
x = class_qlen(f, c);	returns queue length for class c at f;
x = class_resp(f, c);	returns response time for class c at f;
x = class_serv(f, c);	returns service time for class c at f;
x = class_tput(f, c);	returns throughput rate for class c at f;
x = class_util(f, c);	returns utilization for class c at f;

status_facilities()
prints a report on the status of all of the facilities established by the model. The report gives 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.

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 storages.

A set of usage and queueing statistics is automatically maintained for each storage. These are "printed" whenever a *report* or a *report_storages* statement is executed.

Declarations and Initializers

STORE s;
s = storage("name", sz);
declare and initialize s to be a storage of name "name", with sz units of storage; s is a variable of type STORE and sz is a variable of type int; storages should be declared with global variables in the sim (main) process, prior to the beginning of the simulation part of the model.

STORE arr[n];
storage_set(arr, "name", sz, n);
declare a set of n storages, each with sz units of storage; the i-th storage is named "store[i]"; the pointer to the i-th storage is stored at arr[i], where arr is an array of variables of type STORE and is of length n; the storages are indexed 0 through n-1.

set_name_storage(st, name);
change the name of the storage specified by st to the string "name".

delete_storage(s);
delete_storage_set(s);
deletes the specified storage or storage set from the model; this is an extreme action and should be used only when needed.

Operations

allocate(m, s);
the amount of storage requested (m) is compared with the amount of storage available at s; if this is sufficient, the amount available is decreased by m and the requesting process continues; if the amount available is not sufficient, then the requesting process is suspended; when storage is deallocated, any waiting processes which will fit are automatically allocated the requested storage and 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; m is a variable of type int and s must be of type STORE and initialized by a *storage* or a *storage_set* statement.

deallocate(m, s);
return m units of storage to s; if there are processes waiting, then these processes are examined in priority order and those which will now fit have their requests satisfied and then are allowed to continue; m is a variable of type int and s be of type STORE and initialized using a *storage* or *storage_set* statement; an error is detected and execution stops if a deallocate() operation causes the count of the number of using processes to become negative; there is no check to insure that a process returns only the amount of storage it has been allocated.

add_store(sz, s);
the amount of storage at storage s is changed by adding the quantity sz (an integer); s must be of type STORE and initialized using either the *storage()* or *storage_set()* statement.

States and Status

m = avail(s);
m is set to the value of the amount of storage available at storage s.

Retrieving Informations and Data from Storages

These functions each return a statistic which describes some aspect of the usage of the specified storage. The type of the returned value for each of these functions is as indicated, where an *x* is type float, *n* is type int, and is type char*.

_	
name = storage_name(s)	returns pointer to name of storage s;
n = storage_capacity(s)	returns capacity of storage s;
n = storage_request_amt(s)	returns sum of requested amounts
n = storage_number_amt(s)	returns time-weighted sum of requesters
n = storage_busy_amt(s) returns	the busy time-weighted sum of amounts
n = storage_waiting_amt(s)	returns waiting time_weighted sum of amounts
n = storage_request_cnt(s)	returns total number of requests
n = storage_release_cnt(s)	returns total number of completed requests
n = storage_queue_cnt(s)	returns number of queued requests
n = storage_time(s)	returns time spanned by report

status_storages();
causes a report on the status of all storages established by the model to be printed. This report includes information on the amount of storage which is available and the name and id of processes waiting in the queue.

EVENTS

Events are used to synchronize the operations of CSIM processes. An event consists of a state variable and two queues for processes waiting for the event to “happen”. One of these queues is for processes which have executed a *wait* statement and one is for processes which have executed a *queue* statement. When the event “happens”, all of the “waiting” processes and ONE of the “queued” processes are allowed to proceed. An event happens when a process (obviously one which is not waiting somewhere) sets the event to the *occurred* state. An event is automatically reset to the *non-occurred* statement when all of the processes at the event which can proceed have done so.

An event_set is an array of events. A process can wait (or queue) for any event in an event_set to “happen”. In addition, a process can specify a time-out interval, so that it will not wait more than a specified amount of time to pass before it proceeds (a returned status lets the process know whether the event occurred or the time-out expired).

Declarations, Initializers and Deleters

EVENT ev;
ev = event("name");
ev = global_event("name");
declare and initialize ev (a variable of type EVENT) to be a pointer to an event of name “name”; events declared and initialized in the sim (first) process are globally accessible (assuming ev is a globally accessible variable of type EVENT); events declared and initialized in other processes are local to the declaring process; these local events can be passed as parameters to other processes; local events are *deleted* when the declaring process terminates; a process can use the global_event function, to initialize an event which is globally accessible; notice that the event statment in the first process (sim) defaults to the global form and the event statement in a process other than sim defaults to the local form; events are initialized to the *not occurred* state.

EVENT arr[n];
event_set(arr, "name", n);
declare and initialize a set of n events; the i-th event is named “name[i]”; the pointer to this event is stored in arr[i], where arr is an array variables of type EVENT and is of length n; the events are indexed 0 through n-1; a process can either wait for a specific event in the set using a wait (or queue) statement (wait(arr[i]);), or it can wait for any event in the set using a wait_any() (or queue_any()) function (j = wait_any(arr);).

delete_event(ev);
delete the event designated by ev; the event be of type EVENT and initialized using either an *event* or *event_set* statement; furthermore, if the event is local, the process deleting the event must be the process which created the event.

delete_event_set(arr);
deletes an event_set; all of the individual events plus the event_set structure are deleted; the event set must be of type EVENT and initialized using the *event_set* statement.

set_name_event(ev, name);
change the name of the event specified by ev to the string "name".

Operations

wait(ev);
wait for event ev to occur; if ev has already occurred, ev is set to the *not occurred* state and execution continues; if ev has not occurred, execution of the process is suspended until ev is placed in the *occurred* state by the execution of a set statement in another active process; at that time, execution of the suspended process is resumed; it is possible for several processes to be waiting for the occurrence of the same event; all waiting processes will resume when the specified event occurs; ev is placed in the *not occurred* state when all waiting processes have been reactivated; ev must be of type EVENT and initialized with an *event* or *event_set* statement.

queue(ev);
this is similar to *wait(ev)* except only one queued process is allowed to resume; the queued process of highest priority (first to arrive in case of a tie) is restarted when event ev occurs; all other queued processes remain queued until subsequent occurrences of ev; when there are both queued and waiting processes at an event, all of the waiting processes plus one queued process (if there is one) are reactivated; ev must be of type EVENT and initialized with an *event* or *event_set* statement.

set(ev);
set event ev to the *occurred* state; when ev is set, all waiting processes and one queued process will be returned to the active state; if there are no waiting or queued processes, the event will be in the *occurred* state after the set statement executes; if there are waiting or queued processes, the event will be in the *not occurred* state after the set statement executes; ev must be of type EVENT and initialized with an *event* or

event_set statement.

clear(ev);
reset event ev to the *not occurred* state; the event must be of type EVENT and initialized using either an *event* or *event_set* statement.

j = wait_any(arr);
lets a process wait for any event in the event set arr; the function value (j) is the index of the event which triggered the waiting process; arr must of type EVENT and initialized using the *event_set* statement; when multiple events in the set are in the *occurred* state, the lowest numbered event is the one recognized by the next waiting process; all processes which are waiting at the set are triggered by the next event which occurs, and these processes all receive the same index value (function value); when an event in the set is recognized, it is automatically returned to the *not occurred* state.

j = queue_any(arr);
lets a process queue for any event in the event set arr; the function value (j) is the index of the event which triggered the queued process; arr must be declared with an EVENT statement and initialized using the *event_set* statement; when multiple events in the set are in the *occurred* state, the lowest numbered event is the one recognized by the next queued process; only one process which is queued at the set is triggered by the next event which occurs; when an event in the set is recognized, it is automatically returned to the *not occurred* state.

n = timed_wait(ev, tm);
allows the process executing this statement to wait either for the event ev to occur or for the interval of time specified by tm to elapse; ev is of type EVENT and tm is of type float; if the event occurs, the processing is identical to the processing of a *wait* statement (see below); if the time interval expires, the process is automatically removed from the event queue and continues; the value of this function indicates which of the two possible actions occurred; this integer value is either EVENT_OCCURRED or TIMED_OUT, depending on what happened; these values are defined in the header file *csim.h*.

n = timed_queue(ev, tm);
allows the process executing this statement to wait either for the event ev to occur or for the interval of time specified by tm to elapse; ev is of type EVENT and tm is of type float; if the event occurs, the processing is identical to the processing of a *queue* statement; if the time interval expires, the process is automatically removed from the event queue and continues; the value of this function indicates which of the two possible actions occurred; this integer value is either EVENT_OCCURRED or TIMED_OUT, depending on what happened; these values are defined in the header file *csim.h*.

wait(event_list_empty);
can be used to allow a process to wait until there are no active processes; this will occur either when all other processes have left the system (terminated) or, if there are other processes, they are all waiting at some point (e.g. for a facility) within the model.

States and Status

j = wait_cnt(ev);
returns the number of processes waiting at the event ev.

j = queue_cnt(ev);
returns the number of processes queued at the event ev.

st = state(ev);
st is set to OCC if the event ev is in the *occurred* state; st is set to NOT_OCC if event ev is in the *not occurred* state; OCC (= 1) and NOT_OCC (= 2) are constants defined in the the header file “csim.h”.

n = event_qlen(ev);
returns the number of processes at the event specified by ev; this is the sum of the number of processes which have done a wait and which have done a queue at ev; ev must be of type EVENT and initialized using either an *event* or *event_set* statement.

status_events();
causes a report on the status of all events established by the model to be printed. This report includes information on the state of the event and the name and id of each process which is waiting on the event to occur.

MAILBOXES

A mailbox is a container for holding CSIM messages. A process can send a message (an integer or a pointer) to a mailbox and a process can receive a message from a mailbox. If a process does a *receive* operation on an empty mailbox, is automatically waits until the next message is *sent* to that mailbox. As with events, a process can specify a time-out interval when it does a receive.

Declarations, Initializers and Deleters

MBOX mb;
mb = mailbox("name");
declare and initialize mb (a variable of type MBOX) to be a mailbox; the mailbox can be either global (declared in the sim process) or local (declared in a process other than sim); local mailboxes disappear when the declaring process terminates.

delete_mailbox(mb);
delete the local mailbox designated by mb, a variable of type MBOX; the mailbox must have been initialized using the *mailbox* statement; furthermore, the process deleting the mailbox must not be process #1 (sim) and must be the process which created the mailbox.

set_name_mailbox(mb, name);
change the name of the mailbox specified by mb to the string "name".

Operations

send(mb, msg);
send message msg to mailbox mb; currently, a message is a single integer (or a single word pointer); unreceived messages will be queued in order of arrival; mb must be of type MBOX and initialized with a *mailbox* statement.

receive(mb, &msg);
receive the next message from mailbox mb; the message will be placed in the integer variable msg; if no message is ready, the process will wait until a message arrives at the mailbox; the process will restart when a message is received; mb must be of type MBOX and initialized with an *mailbox* statement.

n = timed_receive(mb, &msg, tm);
allows the process executing this statement to wait either for the arrival of a message at the mailbox mb or for the time specified by tm to elapse; mb is of type MBOX and tm is of type float; if a message arrives (or has already arrived), the processing is identical to the processing of a *receive* statement; if the time interval expires, the process is automatically removed from the mailbox queue and continues; the value of this function indicates which of the two possible actions occurred; this integer value is either EVENT_OCCURRED or TIMED_OUT, depending on what happened; these values are defined in the header file *csim.h*.

States and Status

i = msg_cnt(mb);
i is set either to the count of the number of unreceived messages at mailbox mb or the negative of the

number of processes waiting to receive messages at mb; mb must be of type MBOX and initialized with a *mailbox* statement; the value of the function is of type int.

status_mailboxes();
causes a report on the status of all mailboxes established by the model to be printed. This report include information on the number of unreceived messages and the number of waiting processes at each mailbox; in addition, the name and id of each waiting process is listed.

TABLES, QTABLES, HISTOGRAMS and QHISTOGRAMS

As has been noted in the descriptions of facilities and storages, CSIM automatically collects and produces some statistics on the usage of simulated resources. In addition to these, CSIM has four different kinds of structures which can be used to gather more specialized data as a model executes. A *table* contains a statistical summary of the values which have been recorded (in the table). A *histogram* is a table which has been augmented by a frequency histogram. When a table or histogram is printed (using the *report_table* statement), the summary (and histogram) are printed in a standard format. In addition, all of the tables in a model are printed when a *report* statement or a *report_tables* statement is executed.

A *qtable* is normally used to maintain statistics on the behavior of a queue of processes. These statistics summarize the number of processes in this queue and the time processes wait in this queue. The term queue in this context really means some collection of identifiable states. One common use for a qtable is to collect information on the number of processes at a facility. A *note_entry* statement is executed by a process as it enters a facility, and a *note_exit* statement is executed when a process leaves a facility. A *qhistogram* gives the distribution of the number of processes “in the queue”. The contents of qtables and qhistograms are printed with the *report_qtable*, *report_qtables* and *report* statements. A set of data retrieval functions can be used to extract individual statistics from a table or qtable.

The contents of tables and qtables, etc., can be cleared by executing *reset_table* (or *reset_qtable*) statements and are cleared whenever a model is *reset*. These statements can be used to discard data gathered during the warmup of a model. *Permanent_tables* and *permanent_qtables*, etc. are not cleared when a model is *reset* or *rerun*; these can be used to hold data which summarizes multiple runs of a model.

Declarations and Initializers

TABLE t;
t = table("name");
t = permanent_table("name");
declare and initialize t (a variable of type TABLE) to be a table, with name “name”, to be used for recording statistics; tables should be declared using global variables in the sim (main) process before the *simulation* part of the program begins; the contents of a permanent table are not cleared out when a model executes a *reset*, *rerun* or *clear_tables* statement, while a normal table is cleared by one of these.

HIST h;
h = histogram("name", n, lo, hi);
h = permanent_histogram("name", n, lo, hi);
declare and initialize h (a variable of type HIST) to be a histogram, with name “name”; the histogram will contain n+2 buckets: one bucket counts the recorded values less than lo; one counts values greater than or equal to hi; the remaining n equally distributed buckets count the values corresponding to each bucket; a histogram has a table automatically associated with it; thus, statements which clear and print tables can also be used with histograms; permanent histograms are not cleared by executing a *reset*, *rerun*, or *clear_tables* statement; normal histograms are cleared by executing one of these statements.

QTABLE qt;
qt = qtable("name");

qt = permanent_qtable("name");
declare and initialize qt (a variable of type QTABLE) to be a qtable; a qtable is a data collection structure, used to collect information which looks like queue length data; the commands *note_entry(qt)* and *note_exit(qt)* are used to mark the entry and/or exit of a process at a queue; this can also be used to collect data on any quantity dealing with questions about how many things are doing what; when a report is generated, the qtables are automatically printed as part of the report; a permanent qtable is not affected by a *reset*, *rerun*, or *clear_tables* statement, while a normal qtable is.

QHIST qh;
qh = qhistogram("name", n);
qh = permanent_qhistogram("name", n);
declare and initialize qh (a variable of type QHIST) to be a qhistogram with n entries; a qhistogram is a table of values which are the fraction of time the *thing being observed* is in a specified state; for a queue, the state could be the number of tasks in the queue; a qhistogram has a qtable automatically associated with it; when a report is generated, the qhistograms are printed as part of the report; a permanent qhistogram is not affected by a *reset*, *rerun*, or *clear_tables* statement; it is like a normal qhistogram in all other aspects, while a normal qhistogram is cleared by one of these statements.

delete_table(t);
delete_qtable(qt);
deletes the specified table or qtable; this also deletes a histogram or qhistogram;;

Operations

record(x, t);
enter the value x into table t; the table will maintain the sum of the x values, the sum of x squared values, the number of entries, and the minimum and maximum of the entered values; x is of type float; t must be of type TABLE and initialized using the *table* statement; if t was declared as a histogram, then the value x is also recorded as an entry in the associated histogram.

note_entry(qt);
note_exit(qt);
these two statements are used to collect data which looks like queue length data; qt must be of type QTABLE and initialized by the qtable statement above; when *note_entry* is called, the queue state data is updated and the current queue length (queue state) is increased by one; when *note_exit* is called, the queue state data is updated and the current queue length is decreased by one.

report_table(t);
report_qtable(qt);
these statements cause the specified table (or histogram) or qtable (or qhistogram) to be formatted and printed.

reset_qtable(qt);
reset_table(t);
causes the contents of the specified table or qtable to be cleared to zeros.

Retrieving Data from Tables and Qtables

These table and histogram statistics functions all return values of type float except for table_cnt which returns a value of type int and table_name which returns a pointer to the table name (type char*).

s = table_name(t);	pointer to name of table t;
n = table_cnt(t);	number of values recorded in table t;
x = table_mean(t);	mean of values recorded in table t;
x = table_min(t);	minimum of values recorded in table t;
x = table_max(t);	maximum of values recorded in table t;
x = table_sum(t);	sum of values recorded in table t;
x = table_sum_square(t);	sum of squares of values recorded in table t;
x = table_var(t);	variance of values recorded in table t;
n = histogram_bucket(t, i);	contents of bucket i in histogram t;
x = histogram_high(t);	value of highest bucket in histogram t;
x = histogram_low(t);	value of lowest bucket in histogram t;
n = histogram_num(t);	number of buckets in histogram t;
x = histogram_width(t);	width of the buckets in histogram t;

These qtable and qhistogram statistics functions return values of type float, int or char* as indicated by an “x”, “n”, or “s” respectively.

s = qtable_name(qt);	pointer to name of qtable qt;
n = qtable_cnt(qt);	count of state exits from qtable qt;
n = qtable_cur(qt);	current number of entries, but not exits, from qt;
n = qtable_max(qt);	maximum number of entries, but not exits, in qt;
x = qtable_qlen(qt);	mean number of state entries in qt (queue length);
x = qtable_qtime(qt);	average time between entry and exit in qt (time in queue);
x = qtable_qtsum(qt);	sum of time*number factors (space-time integral).
n = qhistogram_bucket_cnt(qt, i);	contents of state i count;
x = qhistogram_bucket_time(qt, i);	contents of state i time;
n = qhistogram_num(qt);	number of states;
x = qhistogram_time(qt);	elapsed time;

PROCESSES

As stated earlier, 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 (or 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 “fork” statement, for those readers who know what a fork statement is.

A process can be invoked with input arguments, but it cannot return values to the invoking process. There can be several simultaneously “active” instances of the same process, and 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) and waiting for an event to happen (or a facility to become available, etc.). When an instance of a process terminates (equivalently does a procedure exit), it is deleted from the CSIM system. Processes have a unique process id’s and inherit their priority from the invoking process; this priority can by changed by the process.

Process Initiation and Termination

proc(arg1, ..., argn);
i = proc(arg1,...,argn);
a process named “proc” is initiated by giving its name (as with a procedure call or function invocation in C) and any arguments to be passed; if desired, the process id of the initiated process (really the new instance of the initiated process) is returned as the function value to the initiating process;
Note: A process cannot return a function value; also, care must be exercised to avoid passing parameters which are addresses of variables local to the initiating process (e.g., local arrays) because when the initiated process is executing, the initiating process will be suspended and the addresses could be invalid.
Note: a *create()* statement (see below) must appear in the initiated process.

create("name");
when executed within a procedure, *create* establishes that procedure as a CSIM process with the given “name”; normally the *create()* statement will be the first executable statement in the body of each process description; each process is given a (unique) process id (process is’s are not reused); the process terminates when it either executes a terminate statement or when it does a normal procedure exit; processes can invoke procedures and functions in any manner; processes can initiate other processes as indicated above; the priority of the initiated process is inherited from the initiator; for the sim (main) process, the default priority is 1; to change the priority of process, the *set_priority()* statement is used.

terminate();
the process executing this statement is ended; a *terminate* statement is not required if the process (procedure) exits normally.

set_priority(p);
the priority of the process is set to p (of type int), this statement should appear after the *create* for its process.

p = priority();
p is set to the priority (type int) of the process.

id = identity();
id is set to the identity (process number of type int) of the process executing the statement.

Process status

The status of each process in the model is summarized in a process_status report.

status_processes();
causes a status report on all of the active processes in a model to be printed. This report lists each active process, along with its status, priority and process_class membership.

status_next_event_list();
causes a status report about the “next event lsit” to be printed. This report gives the name and id of each process which is currently in the next event list.

PROCESS CLASSES

Process classes are used to segregate data for reporting purposes. A class has a name and is referenced by a pointer (just like other csim objects). A "process class report" gives for each process class the number of processes which were in the class, the average lifetime *in the class*, the number of *hold* statements executed, the average amount of hold time per process, and the average wait time (lifetime minus hold time) per process.

Declarations and Initializers

CLASS c;
c = process_class("name");
declare and initialize c (a variable of type CLASS) to be a pointer to a class of name “name”.

delete_process_class(c);
deletes the specified process class from the system; note, if a facility is collecting statistics for the deleted class, this will continue.

Operations

set_process_class(c);
cause the process executing this statement to become a member of the class pointed to by c; all processes are automatically members of the “default_class”; a process changes class membership by executing this *set_process_class* statement.

c = current_class();
returns the current class of the process executing this statement; c is a variable of type CLASS.

Retrieving Information and Statistics for Process Classes

n = class_id(c)	return id of process class c
name = class_name(c)	return pointer to name of c
n = class_cnt(c)	return number of instances of c
x = class_lifetime(c)	return total lifetime for all instances of c
n = class_holdcnt(c)	return number of holds for all instances of c
x = class_holdtime(c)	return total holdtime for all instances of c

RANDOM NUMBERS

The CSIM library has several functions which can be used to generate random derivates from specified distributions. In the following section, the variables y, u, u1, u2, v, and s are of type float; i, i1 and i2 are of type integer. Most of these functions use *prob()* to generate a [0,1] uniformly distributed random sample as the basis for a random sample from some other distribution.

y = erlang(u, v);
y is set to a random derivate drawn from an Erlang-k distribution, where u is the mean, v is the variance, and k is (u*u/v + 0.5).

y = expntl(u);
y is set to a random derivate drawn from a negative exponential distribution with mean u.

y = hyperx(u, v)
y is set to a random derivate drawn from a hyperexponential distribution with mean u and variance v.

y = normal(u, s)
y is set to a random derivate drawn from a normal distribution with mean u and standard deviation s.

y = prob();
y is set to a random derivate drawn from a uniform [0,1) distribution; note: prob() calls the C function rand().

i = random(i1, i2);
i is set to a random derivate drawn from a uniform [i1,i2] distribution.

reset_prob(1);
reset_prob(i);
resets the random number generator; the first form (i = 1) will cause the sequence of random numbers to begin again; reset_prob(i) causes the random number sequence to be initialized to a function of i.

y = uniform(u1, u2);
y is set to a random derivate drawn from a uniform (u1,u2) distribution.

REPORTS

As stated above, usage and queueing statistics for facilities and storages are gathered automatically and printed when requested (by a *report* statement or a *report_facilities* or a *report_storages* statement).

Note: if no report statement is specified, no output from CSIM will be generated.

report();
causes a complete report to be printed; this report includes facility usage data, storage data, contents of tables, histograms, qtables and qhistograms; the section entitled “Output from CSIM” gives a complete description of a report.

report_hdr();
report_facilities();
report_storages();
report_classes();
report_tables();
each cause a report on the usage of the indicated class of resources or the contents of all of the tables and qtables to be printed; report_hdr prints the header section from the full report, including date and time of the report, the simulated interval of time and the cpu time required; the section entitled “Output from CSIM” gives a complete description of each kind of report.

set_model_name("name")
causes the model name string to be changed from “CSIM” to the string specified; this name appears near the top of the standard report.

mdlstat();
causes a report to be printed giving the usage of computing resources by the CSIM model as it executed; the report begins with the user mode CPU time consumed and a count of the number of events processed; it also includes a count of the number of processes created , the amount of main memory obtained via malloc calls, and statistics on the number of processes started and active and on the amount of storage devoted to runtime stacks for CSIM processes.

dump_status()
causes status reports on all of the major components of a model to be printed. The different CSIM components covered with following status reports:

```
status_events();
status_facilities();
status_mailboxes();
status_next_event_list();
status_processes();
status_storages();
```

Notices that each of the above status statements are callable; thus, a “customized” status report can be created by the programmer.

Output from CSIM

The output generated by the *report();* statement presents a report on the simulation run as it has progressed *so far*. The report has six parts (these are described in detail below):

1. A brief header, giving the date, simulated time, etc.,
2. A report on facility usage (if any facilities were declared),
3. A report on storage usage (if any storages were declared),
4. A report on the process classes (if more than one process class (the default process class) has been declared),
4. A summary for each table (and histogram) declared, and
5. A summary for each qtable (and qhistogram) declared.

The following tables give a complete description of each of these six segments of the report:

Header	
Date, time of report and version number	
Time: Value of clock (simulated time)	
Interval: Length of simulation interval (since last reset)	
CPU Time: Real CPU required (since last report())	
Revision (and version) Specifier	

Facility Usage	
facility	name (for a facility set, the index is appended)
srv	server number (for facilities with multiple servers)
disp	service discipline (when one was declared)
means	(see Note below)
serv_tm	service time per request
util	utilization (busy time divided by elapsed time)
tput	throughput rate (completions per unit time)
qlen	number of requests waiting or in service
resp	time at facility (both waiting and in service)
counts	
cmp	number of requests completed
pre	number of preemptions which occurred

Note: if collection of process class statistics is specified, then the above items are repeated on a separate line for each process class which visits the facility.

Storage Usage

storage	name
capacity	size of storage
means	(see Note below)
amt	amount of storage per request
util	fraction of storage in use during the simulation interval
srv_tm	time in storage per request
qlen	number of requests in storage or waiting
resp	time requests are in storage or waiting
counts	
cmp	number of requests completed
que	number of requests which had to wait

Process Classes

id	process class id
name	process class name
number	number of processes becoming members of this class
lifetime	mean time each process in class
hold ct	mean number of hold statements per process
hold time	mean hold time per process
wait time	mean wait time per process (lifetime - hold time)

Note: If no classes are specified, the report is for the “default” class (every process begins as a member of this class). If classes are specified, then the report does not include the default class.

Tables and Histograms

mean	average of values recorded
variance	variance of values recorded
min	minimum of values recorded
max	maximum of values recorded
number	number of entries in table
Histograms	
Low	low value for this bucket
High	high value for this bucket
Count	of entries in this bucket
Fraction	fraction of total number of entries
Cumulative	cumulative sum of fraction fields
Total	number of entries in histogram

Qtables and Qhistograms	
Mean queue length	tasks between note_entry and note_exit
Mean time in queue	time between note_entry and note_exit
Max queue length	max number of processes *in queue*
Number of entries	number of note_entry's and _exit's divided by 2

Qhistograms	
Length	Number of entities in queue (state)
% of Elapsed Time	Percentage of time at this length
Cumulative	Cumulative percentage
Count	Number of times at this queue length
Mean Time	Mean time interval at this length

Note: when computing averages based on the number of requests for facilities, the number of *completed* requests is used; thus, any requests still at devices when the report is printed do not contribute to these statistics; in addition, all quantities based on usage (such as utilization and service time) have busy time for only completed requests. For storages and qtables, the requests in service when the report is requested do contribute to the derived quantities.

MISCELLANEOUS

This is the section on miscellaneous features of CSIM.

Simulated Time

t = clock;
t is set to the current simulation time; clock is declared in the include file "csim.h"; it is of type TIME (normally double precision).

hold(t);
the process is suspended for simulated time interval of length t; notice that simulated time passes only through the use of *hold* or *use* statements; the input argument, t, must be of type float.

t = simtime();
t is set to the current simulated time; simtime() is a function of type TIME; this statement is equivalent to the *t = clock;* given above.

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 by using the "max_" functions. These maximums serve as limits on the number of structures of a particular type which exist simultaneously. These "max_" functions, along with the default values in CSIM, are as follows:

Function name	Default value
i = max_classes(n);	5
i = max_events(n);	100
i = max_facilities(n);	100
i = max_histograms(n);	10
i = max_mailboxes(n);	50
i = max_messages(n);	1000
i = max_processes(n);	1000

i = max_qtables(n);	10
i = max_sizehist(n);	1000
i = max_storages(n);	20
i = max_servers(n);	200
i = max_tables(n);	20

In each of these functions, if the argument, n, is zero, the current value of the maximum is returned; if n is greater than zero, n becomes the new maximum value (limit) and is returned as the value of the function. The default values are defined as constants in the header file "csimdef.h" and can be changed by editing this file and recompiling the CSIM library.

NOTE: Because each mailbox includes an event, the maximum number of events must include at least one event per mailbox; thus, if max_mailboxes() is increased, then it is likely that max_events() must also be increased.
NOTE: It is an error to change the maximum number of classes after a collect_class_... statement has been executed.

Program Structure and Execution

There are two distinct ways of writing CSIM programs; the standard way is programmer to write a routine named *sim()*. In this way, the usual *main()* is provided from the CSIM library, all initialization is invoked from main(), the command line is processed by main(), and *sim()* is called with argc and argv repositioned to point to the non-CSIM arguments. The second (new) way is for the programmer to provide the *main()* routine. In this way, the program can call *sim()* (or any routine) which becomes the first (base) CSIM process when it executes the *create()* statement. There is a CSIM routine, named *proc_csim_args(argc, argv)*, which can be called to process the CSIM command line arguments. The reason for providing this second method is so that a CSIM model can be embedded in a surrounding "tool".

An executing CSIM program can use a number of routines to control execution and handle CSIM errors; these are described in this sections.

rerun();
allows the model to be rerun; all non-permanent tables, etc. are cleared; all processes are eliminated; all facilities, events, mailboxes, process_classes, storages, tables and qtables established before the first create statement (the create for the first ("sim") process) are reinitialized; all remaining facilities, storages, events, etc., are eliminated; the clock is set to zero; the random number generator (rand()) is not reset and the contents of permanent tables, etc. are kept. The intent is to allow a new model or the same model with modifications to be executed are part of a CSIM program; however, special provisions are required for C++ programs with static objects which are constructed before the programs begins execution.

reset();
causes all of the statistics gathering fields to be cleared; in addition to statistics at facilities, storages and process_classes, non-permanent tables and histograms are also cleared; this feature can be used to eliminate the effects of start-up transients; the global variable _start_tm is set to the current time and is used as the starting point for calculating statistics; the (variable) *clock* is not altered; time intervals for facilities, storages and qtables which begin before the reset are tabulated in their entirety if they end after the reset() call.

t = cputime();
t is set to the amount of user mode cpu time (floating point seconds) accumulated by the program so far.

Runtime Options and Debugging

There are two kinds of input arguments (command line arguments) for a CSIM program. These are arguments addressed to the CSIM routines and arguments addressed to the program. The CSIM arguments are processed before the first user-provided procedure is called. This first procedure, *sim*, is called with *argc* and *argv* as its two input parameters, just as the *main* program is normally invoked in the standard C

runtime environment. The CSIM arguments must appear before (in the command line) the program's arguments; these CSIM arguments are used as follows:

a.out -T
the -T option causes the switch *trace_sw* to be set as the soon as the program begins execution; the *trace_sw* switch controls tracing of CSIM events; this trace is a listing of some information about each event as it occurs; the listing is directed to the standard output file.

a.out -L
the -L option causes the event logging mechanism in CSIM to be activated; as the program executes, a file (named csim_log) of event records is created; each event record is time-stamped with the simulated time and contains an event id plus additional information particular to that event. The postrun analysis program, named MONIT, can be used to analyze this event file. MONIT is described in an MCC Technical Report [Kero86].

Note: The command line options -T and -L (if present) are "picked off" by the CSIM main procedure and then the remainder of the command line arguments are passed on to the sim procedure using argc and argv under the normal C conventions. If a *main()* procedure is used (instead of *sim()*), then the *proc_csim_args()* procedure can be used to process the input arguments:

proc_csim_args(&argc, &argv);
can be invoked by a *main(argc, argv)* program, to cause the CSIM arguments to be processed. On return from proc_csim_args(), the -T and -L arguments (if present) have been processed and argc and argv modified to point to any remaining arguments.

conclude_csim();
can be invoked by a *main* program, to cause the CSIM model to be correctly concluded; if a model is to be *rerun*, then the rerun() statement should be executed; to reset the random stream, use reset_prob(1);

The *trace_sw* can be controlled while the program is executing. The procedures for doing this are as follows:

trace_on();
trace_off();
turns on (or off) the *trace_sw*; as these are executable statements, the trace can be turned on and/or off during the execution of the the program; in particular, these procedures can be used to provide selected tracing of portions of the program.

trace_msg(str);
inserts a string (message) into the trace output; the string is printed only if the trace_sw is on.

Using CSIM

At MCC, CSIM is located on the PPVAX, PPSEQUENT and DBVAX systems and on some of the SUN workstations. You should copy the command file (or the proper equivalent for your system):

```
/users/pp/hds/csim/csim
```

to your local file area. Then the command:

```
csim file.c
```

will compile your CSIM program (named file.c, or anything else you choose) and the executable module placed in the file name a.out. All of the standard options to the C compiler can be used, including the -o option which specifies a name for the executable module.

To execute your program, just give the command:

```
a.out
```

assuming that the standard defaults were used.

The command

```
a.out -T
```

will cause a CSIM debugging event trace to be created on the standard output file. Similarly, the command

```
a.out -L
```

causes the event logging feature to be activated. As described earlier, any combination of these, as well as other command line arguments, can be used on the command line. The CSIM arguments, if present, are extracted from the argument string before *sim* is called. The argument count, argc, and the remainder of the argument string (pointed to by argv) are passed to *sim* as is normally done for C programs.

The first executable procedure must be named *sim* (there is an exception to this - see below). *Sim* must contain a *create* statement. The header file “csim.h” must be included in your program; to do this, have the statement (or its equivalent referencing the proper file on your system)

```
#include "/users/pp/hds/csim/lib/csim.h"
```

at the beginning of your program.

It is also possible for a CSIM model to have its own *main* routine.

Error handlers -

Reminders

When writing a CSIM program, the following things should be remembered:

1. In the current version, there is a limit of 1000 concurrently active processes; this can be changed by using the function *max_processes*.
2. When a process (a procedure containing a *create()* statement) is called with parameters, it is recommended that these be either parameters passed as values (the default in C) or addresses of variables in global (or static) storage. Process are managed in CSIM by copying the runtime stack out (to a save area) when the process suspends and then back to the stack when the process resumes. This means that if a process receives a parameter which is an address in the local storage of the initiating process (i.e. in that process’s stack frame), the address will not point to the value when the called process is executing. THIS IS VERY IMPORTANT! Beware of local arrays and strings which are parameters for processes!
3. The statements initializing globally accessible entities (facilities, storages, tables and global events) should be declared using global variables of the correct types; they must be initialized prior to being referenced.
4. For programmers not accustomed to C, an array of length n is indexed 0,1,...,n-1.
5. The *reset()* statement allows values accumulated in statistical counters (for facilities, storages, tables and qtables) to be discarded; this can be used to eliminate the effects of startup transients which may be present in the behavior of the modeled system; permanent tables, histograms, qtables and qhistograms are not affected by this statement.
6. The *rerun()* statement causes all of the CSIM structures of the program to be deleted from the model; the seed for the random number generator is not altered; this can be used to either produce replications of a modeled system or variations of a modeled system to be used in the same execution of the

CSIM program; the “clean up” is complete, to the point that the *create* statement in the procedure *sim* must be executed again; permanent tables, etc. and programmer defined data areas are not altered by this statement.

Error Messages

The following error messages can be printed by a CSIM program which *gets into trouble*; with each error message is a brief interpretation and a suggested course of action:

1. NEGATIVE EVENT TIME - tried to schedule an event to occur at a negative time; probably either a negative hold interval or a program which has truly run away.
2. EMPTY EVENT LIST - (a popular error in CSIM) - somehow, every active process is waiting for an event to occur, and there is no process which can cause an event to happen; sometimes, this error indicates that a create() statement was left out of a process; sometimes, this is a symptom of deadlock; sometimes, it is a subtle error in process synchronization; if all else fails, 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 NTASKS processes at a facility declared with the proc_share service function; NTASKS (currently 20) is a constant defined in the header file “csimdef.h”.
6. NOTE FOUND CURRENT STATE LESS THAN ZERO - the note_chg procedure was asked 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 for 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, the calling process had not created the event, or the argument did not point to an event created by the calling process.
8. ERROR IN DELETE MAILBOX - the delete_mailbox procedure was called and one of the following failures occurred: the argument was NIL, the calling process was process #1 (sim), or the argument did not point to a mailbox created by the calling process.
9. MALLOC FAILURE - The UNIX routine name *malloc* was unable to allocate more memory to the program. Malloc is used to alloc space for process control blocks, 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. ERROR IN CANCEL EVENT FOR PROCESS (INTERNAL ERROR) - Somehow the process_sharing or last-come, first-servered service disciplines have gotten confused and have tried to preempt a process which does not hold the facility; see your system maintainer.
11. ILLEGAL EVENT TYPE (INTERNAL ERROR) - Somehow, the procedure for creating events has been called with a mode (type) parameter which is not recognizable; see your system maintainer.
12. TOO MANY EVENTS - The limit on the number of events which can be simultaneously in existence is being exceeded; this indicates one of two conditions: the program needs more events (see the *max_events* function) or there is an error.
13. TOO MANY FACILITIES - The limit on the number of facilities which can be simultaneously in existence is being exceeded; this indicates one of two conditions: the program needs more facilities (see the *max_facilities* function) or there is an error.
14. TOO MANY HISTOGRAMS - The limit on the number of histograms which can be simultaneously in existence is being exceeded; this indicates one of two conditions: the program needs more histograms (see the *max_histograms* function) or there is an error.
15. TOO MANY MAILBOXES - The limit on the number of mailboxes which can be simultaneously in existence is being exceeded; this indicates one of two conditions: the program needs more mailboxes

(see the *max_mailboxes* function) or there is an error.

16. TOO MANY MESSAGES - The limit on the number of messages which can be simultaneously in existence is being exceeded; this indicates one of two conditions: the program needs more messages (see the *max_messages* function) or there is an error.
17. TOO MANY PROCESSES - The limit on the number of processes which can be simultaneously in existence is being exceeded; this indicates one of two conditions: the program needs more processes (see the *max_processes* function) or there is an error.
18. TOO MANY QTABLES - The limit on the number of qtables which can be simultaneously in existence is being exceeded; this indicates one of two conditions: the program needs more qtables (see the *max_qtables* function) or there is an error.
19. TOO MANY STORAGES - The limit on the number of storages which can be simultaneously in existence is being exceeded; this indicates one of two conditions: the program needs more storages (see the *max_storages* function) or there is an error.
20. TOO MANY SERVERS - The limit on the number of servers which can be simultaneously in existence is being exceeded; this indicates one of two conditions: the program needs more servers (see the *max_servers* function) or there is an error.
21. TOO MANY TABLES - The limit on the number of tables which can be simultaneously in existence is being exceeded; this indicates one of two conditions: the program needs more tables (see the *max_tables* function) or there is an error.
22. CANNOT OPEN LOG FILE - The event logging procedures are not able to open the file “csim_log”; there is probably some problem with privileges and protection in the current directory you are using.
23. DEQUEUE FROM QUEUE FAILED - Not currently valid.
24. TRIED TO RETURN AN UNALLOCATED PCB
25. TRIED TO CHANGE MAXIMUM CLASSES AFTER COLLECT
26. TOO TOO MANY CLASSES
27. IN RETURN EVENT, FOUND WAITING PROCESS - an attempt was made to delete a local event and a process was discovered to be waiting on 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; somehow, the function thinks that there is an event in the set which has *occurred*, but it did not find one; if this error occurs, contact your system maintainer.
31. STORAGE DEALLOCATE ERROR: CURRENT COUNT < 0 - the deallocate() procedure has detected a negative value for the current number of users at a storage; this is probably the result of having some processes doing a deallocate() without a prior allocate() operation.
32. TIMED_RECEIVE ERROR - MSG WAS LOST - things have gotten confused in timed_receive(); this should not happen.
33. MULTISERVER FACILITY - ZERO OR NEG. NUMBER OF SERVERS - obvious.
34. TRIED TO CHANGE MAX_CLASSES AFTER CREATING PROCESS CLASSES - after specify collecting process class statistics at a facility, max_classes cannot be changed.
35. ASKED FOR STATS ON NON-EXISTENT SERVER - make a call to some server statistics function and specified an out-of-range server number.
36. ERROR IN CALENDAR QUEUE INIT - an internal error.
37. ERROR IN DELETE FACILITY -

38. ERROR IN DELETE PROCESS CLASS -
39. ERROR IN DELETE QTABLE -
40. ERROR IN DELETE STORAGE -
41. ERROR IN DELETE TABLE -

Acknowledgements

Teemu Kerola assisted in the initial implementation of CSIM. He also designed and implemented the MONIT event logging feature and the postrun analysis program for the SUN. Bill Alexander has provided consultation on the wisdom of many proposed features. Leonard Cohn 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 have 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. Murthy Devarakonda, of IBM T.J. Watson Research Labs, did most of the work on the IBM RS/6000 support.

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.

[KeRi78] Kernighan, B.W. and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.

[Kero86] Kerola, T., *Monit: An Analyzer for PPL and CSIM Trace Files*, Microelectronics and Computer Technology Corporation, Technical Report PP-304-86.

[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.

[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.

[Schw85] Schwetman, H.D., *CSIM: A C-Based, Process-Oriented Simulation Language*, Microelectronics and Computer Technology Corporation, Technical Report PP-080-85.

[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.

[Schw90a] Schwetman, H.D., "CSIM Reference Manual (Revision 14)", Microelectronics an Computer Technology, Technical Report ACA-ST-257-87 Rev 14).

[Schw90b] Schwetman, H.D., "Introduction to Process-Oriented Simulation and CSIM", *Proceedings of the 1990 Winter Simulation Conference*, December, 1990, pp. 154- 157..

A 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 qhistogram feature to collect the frequency distribution of the queue length.