June 1, 1992

CSIM† Reference Manual (Revision 16)

Herb Schwetman

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

How to Read This Document

This Reference Manual documents all aspects of the user interface to CSIM. It is fairly terse and contains only one example and no information on computer simulation in general. You are urged to refer to any of a number of relevant textbooks and articles, if you need more information about simulation. Installation and maintenance procedures for CSIM are covered in other documents.

If this is your first exposure to CSIM, skip the Preamble and start with the Introduction. The Preambles are for people who are using earlier revisions and need to find out what is different in this revision. The preambles for Revisions 11, 12, 13, 14 and 15 are included at the end of this document, as an Appendix. They should be read only by users of earlier revisions. Note that Revision 11 was a major rewrite of the CSIM library. In particular, CSIM programs written for Revisions 10 and before will NOT compile with Revisions 11 and after. A new section, near the end of this document, gives some hints on how to convert old programs to operate with Revisions 11 and after.

Preamble to Revision 16

The following additions and corrections are found in Revision 16:

1. The dump_status() procedure was added. Dump_status() makes calls to status_events(), status_facilities(), status_mailboxes(), status_next_event_list(), status_processes(), and status_storages();
2. Routines to fetch the name, number of preempted requests, number of servers and service discipline for a specified facility were added (facility_name(), preempts(), num_servers(), and service_disp()).
3. Routines to fetch the number of completions, service time, throughput and utilization for a specified server at a facility were added (server_completions(), server_serv(), server_tput(), server_util()).
4. Some changes were required to support AT&T C++ 2.1.
5. Make a csim model callable as a function (i.e., the model has its own main() routine). This requires two new routines: proc_csim_args() and conclude_csim() and changes to all of the constructor routines (event(), facility(), etc.).
6. Change the error routine to put the error message on stderr (instead of stdout).
7. Change the report header, to print the version number and a system name (instead of the encoded number as in earlier versions). In addition, if the C++ version of the CSIM library, this is also indicated.
8. Allow sim() procedure to handle CSIM errors; new procedures are set_error_handler() and print_csim_error();
9. Allow program to direct different kinds of output to different output files; this necessitated the addtion of four new procedures: set_output_file(), set_error_file(), set_trace_file() and set_log_file_name();

† Copyright Microelectronics and Computer Technology Corporation, 1987, 1988, 1989, 1990, 1991, 1992

10. A function to return the number of simulation events processed so far has been added: events_processed();

11. Fix an error in delete_mailbox; a deleted global mailbox was not being deleted from the list of global mailboxes.

12. Fix an error in process classes; start_hold was not being initialized.

13. Add functions qtable_name() and table_name();

14. Add functions to access contents of a histogram: histogram_bucket(), histogram_high(), histogram_low(), histogram_num() and histogram_width().

15. In the C++ version, the methods in the class table all had table removed from the method; so table_cnt() is now cnt(). The same was done for the class qtable.

16. Add functions to access contents of a qhistogram: qhistogram_bucket_cnt(), qhistogram_bucket_time(), qhistogram_time(), and qhistogram_num().

17. Add functions to give facility statistics for process classes: class_completions(), class_qlen(), class_resp(), class_serv(), class_tput() and class_util().

18. Add functions to retrieve statistics and information for process classes: class_id(), class_name(), class_cnt(), class_lifetime(), class_holdcnt() and class_holdtime().

19. Add functions to retrieve statistics and information for storages: storage_name(), storage_capacity(), storage_request_amt(), storage_busy_amt(), storage_number_amt(), storage_waiting_amt(), storage_request_cnt(), storage_release_cnt(), storage_queue_cnt() and storage_time().

20. In two places, fix bug: calls to xerr with no error number (John Greene, of NCR, reported both of these).

21. In facilities report, change server numbering to 0 through N-1; this is to make this consistent with other numbering schemes in CSIM.

22. Permanent tables (and qtables) are now numbered separately from normal tables (and qtables). In addition, the table and qtable reports indicate if a table or qtable is permanent.

23. Modify rerun(), so that static objects in C++ are not deleted on rerun; instead they are reinitialized on rerun.

24. Modify reset(), so that process classes are also reset.

25. Add functions to delete facilities, facility_sets, process_classes, qtables, storages and tables (delete_facility(), delete_facility_set(), delete_process_class(), delete_qtable(), delete_storage() and delete_table()).

26. Remove Cray support.

27. Add support for Macintosh with Think C compiler.

28. Move definition of TIME_BASE from csimdef.h to cprim.c; change definition of TIME_BASE for SYSTEM V configurations to use the CLK_TCK constant in <limits.h>.

29. In the PC486 version, add support for Metaware C compiler.

30. Add support for the HP Prism Architecture Workstations (HP-700).

31. Add the current_class() function, to allow a process to find out which class it is in.

32. Document the current_state function; this function returns the current state of a qtable.

33. Fix bugs in queue_any() and wait_any().

34. Add support for the IBM RS/6000 series workstations.

35. Changed facility and class reports so that statistics are printed if the release count is greater than zero; before, the statistics were printed if the usage time was greater than zero. The effect is that statistics will now be printed for servers and classes which had hold times of 0.0.

    All of the additions are highlighted in this document.

Introduction

CSIM is a process-oriented simulation language based on the C programming language [KeRi78], running under the UNIX†† or the VMS ††† operating systems. CSIM is patterned after the simulation language ASPOL, by MacDougall [MaMc73, MacD74, MacD75].

The motivation for CSIM arises from several observations:

1. Process-oriented simulation is a convenient tool for developing simulation models of computer and communication systems;
2. It is possible to use a process-oriented simulator, such as CSIM, as an execution environment for parallel programs; this is will be discussed in a subsequent document; and
3. Process-oriented simulation is be a useful technique for estimating program behavior on novel architectures.

This document is the reference manual for CSIM. In addition to a summary of the features and a brief description of CSIM, the document describes all of the statements in CSIM. It concludes with a sample program.

Features

CSIM includes the following features:

1. Programming is done in C; the simulation features are implemented primarily as calls to procedures in a runtime library.
2. The simulation programs (models) are compiled, not interpreted; execution times should be acceptable.
3. The environment simulated is a (quasi-)parallel execution environment; the basic unit of execution is a process; a program can initiate sub-processes; sub-processes can initiate other processes, etc.; processes can wait for events to happen and cause simulated time to pass.
4. Each process has both a private data store (memory) as well as access to global data.
5. Statistics gathering is partially automatic and easy to extend.
6. There are provisions for declaring and using facilities and storages; use of facilities can be governed by several different service disciplines.
7. There are provisions for tracing execution of the simulation model and for logging simulated events; this file of logged events can be analyzed by a separate program, such as the MONIT analysis program which executes on SUN workstations [Kero86].

Currently, CSIM is implemented as a collection of procedures and functions which are callable from C programs. CSIM is available for VAX computer systems, SUN workstations including the SUN4 (with the Sparc microprocessor) and the SUN 386i (with the Intel 80386 microprocessor), the Hewlett-Packard Series 300 workstation, the NCR Tower 650, and the Sequent Balance and Symmetry systems using the Berkeley 4.2 or 4.3 BSD versions of the UNIX operating system and the VMS operating system for VAXes. There are also versions of CSIM for the 386 PC with Xenix and the 486 PC with Unix, as well as DECStations with the MIPS processor.

Description

The primary unit of execution is a process. Execution of a process is initiated by another process; the main program (it must be named “sim”) is really process #1. All concurrently active processes execute in a quasi-parallel fashion; that is, with certain restrictions, each process appears to be executing in parallel with other active processes when in fact they are really executing one-at-a-time on a single processor.

Once a process is initiated, it can:

†† UNIX is a Trademark of Bell Laboratories.

††† VMS is a Trademark of Digital Equipment Corporation.

- wait for simulated time to pass (by executing a “hold” statement),
- wait for an event to occur (by executing a “wait” or “queue” statement),
- cause an event to occur (by executing a “set” statement),
- request exclusive or shared use of a facility (by executing a *reserve* or a *use* statement) or of a storage (by executing an *allocate* statement),
- and eventually terminate.

A process is a member of a process class, where a process class is a collection of processes grouped together for the purpose of segregating statistical outputs. The syntax of the language is normal C (or C++), with the addition of what appear to be extra simulation verbs.

Programs are able to initiate instances of processes (possibly several instances of the same process at the same time), and to declare events, facilities and storages. In addition, processes are able to set their own priorities and to await signals from other processes. A basic level of performance data is automatically collected and reported at the end of the run; additional data can be collected at the discretion of the program. A trace facility for debugging is available as is an event logging feature. Several functions for generating (pseudo-) random numbers drawn from different probability distributions are also supplied.

A complete listing of the CSIM statements and procedures appears in the following sections.

Implementation

CSIM is a set of procedures and functions which are callable by C programs. While most of these procedures are written in C, a few had to be written in assembly language (primarily the routines which manipulate the runtime stack). Thus, moving CSIM to other systems involves not only recompiling the library of C programs, but also rewriting some machine-dependent C procedures and these assembly language routines.

CSIM Statements

CSIM is embedded in the C programming language. In addition to C, the following statements are part of CSIM:

Declarations and Initializers -

Note: Initializers are really executable CSIM functions which return pointers to CSIM data structures. The initializer values and the receiving variables need to be of the same type. Initializer statements must be executed AFTER the *create* statement in their respective processes.

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”.

EVENT ev;
ev = event("name");
ev = global_event("name");
declare ev (a variable of type EVENT) to be a pointer to an event of name "name"; events declared in the sim (first) process are globally accessible (assuming ev is a globally accessible variable of type EVENT); events declared 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 statement 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 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 by using a wait (or queue) statement (wait(arr[i]);), or it can wait for any event in the set by using a wait_any() (or queue_any()) function (j = wait_any(arr);).

FACILITY f;
f = facility("name");
f = facility_ms("name", ns);
declare 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 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 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.

HIST h;
h = histogram("name", n, lo, hi);
h = permanent_histogram("name", n, lo, hi);
declare 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 (see below) 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.

MBOX mb;
mb = mailbox("name");
declare 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.

QHIST qh;
qh = qhistogram("name", n);
qh = permanent_qhistogram("name", n);
declare qh (a variable of type QHIST) to be a qhistogram with n entries; a qhistogram is a table of values which are the relative frequencies of the amount 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 (see below) 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.

QTABLE qt;
qt = qtable("name");
qt = permanent_qtable("name");
declare 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 be a 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.

STORE s;
s = storage("name", sz);
declare 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.

TABLE t;
t = table("name");
t = permanent_table("name");
declare 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.

Limits

There is a limiting (maximum) number for 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.

Process Initiation -

proc(arg1, ..., argn);
i = proc(arg1,...,argn);
a user-written 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.

Execution Statements -

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

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 have been declared as type STORE and initialized by a *storage* or a *storage_set* statement.

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

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

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

collect_class_facility(f);
causes 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.

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

create("name");
when executed by 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 id'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; when a process is created, it is automatically a member of the process class named “default_class”; the *set_process_class()* is used to change the class attribute of a process.

****
cl = current_class();
returns the process class of the the process; cl is of type CLASS.

i = current_state(qt);
returns and integer which is the current state of a qtable; qt is of type QTABLE, and i is of type integer.
****

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 must have been declared as 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.

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 have been declared to be of type EVENT and initialized using either an *event* or *event_set* statement.

delete_event(ev);
delete the event designated by ev; the event must have been declared to 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 have been declared using the EVENT statement and initialized using the *event_set* statement.

****

delete_facility(f);
deletes a facility; no check is made for processes using the facility or waiting to use the facility; this is intended for use at the end of a model's execution; its use is normally not required.

delete_facility_set(arr);
deletes a facility_set; all of the individual facilities are deleted; this is intended for use at the end of a model's execution; its use is normally not required.
****

delete_mailbox(mb);
deletes 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.

****
delete_process_class(c);
deletes a process_class. this is intended for use at the end of a model's execution; its use is normally not required.

delete_qtable(qt);
deletes a qtable. this is intended for use at the end of a model's execution; its use is normally not required.

delete_storage(s);
deletes a storage; this is intended for use at the end of a model's execution; its use is normally not required.

delete_storage_set(arr);
deletes a storage_set; all of the individual storages which are in the set are deleted; this is intended for use at the end of a model's execution; its use is normally not required.

delete_table(t);
deletes a table. this is intended for use at the end of a model's execution; its use is normally not required.
****

hold(t);
the process is suspended for a 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.

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

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 have been declared to be of type MBOX and initialized with a *mailbox* statement; the value of the function is of type int.

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

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

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.

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 have been declared to be of type EVENT and initialized with an *event* or *event_set* statement.

j = queue_any(arr);
lets a process queue for any event in the event set arr; the integer function value (j) is the index of the event which triggered the queued process; arr must have been 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.

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

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 queue until a message arrives at the mailbox; the process will restart when a message is received; mb must have been declared to be of type MBOX and initialized with an *mailbox* statement.

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 have been declared to 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 below); in contrast to the release procedure (see above), 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.

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 have been declared to be of type FACILITY and initialized with a *facility* 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 integer function is the index of the server assigned to the process.

send(mb, msg);
send message msg to mailbox mb; currently, a message is a single integer; unreceived messages will be queued in order of arrival; mb must have been declared to be of type MBOX and initialized with a *mailbox* 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 have been declared to be of type EVENT and initialized with an *event* or *event_set* statement.

set_process_class(c);
change the process class attribute of a process to the process class specified by c.

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

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

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

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

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.

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

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".

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".

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

n = timed_queue(ev, tm);

allows the process executing this statement to wait either for the event ev to occur or for the 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*.

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*.

n = timed_wait(ev, tm);
allows the process executing this statement to wait either for the event ev to occur or for the 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*.

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 have been declared to be of type EVENT and initialized with an *event* or *event_set* statement.

j = wait_any(arr);
lets a process wait for any event in the event set arr; the integer function value (j) is the index of the event which triggered the waiting process; arr must have been 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 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 = wait_cnt(ev);
returns the number of processes waiting at the event ev.

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.

Facility Usage -

set_servicefunc(f, func);

this statement allows the programmer to change the default service discipline at facility f, which is fcfs (first-come, first-served), to some other service discipline, which is specified by func; func is the function which is invoked when the *use* statement (described below) 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 (instead the *reserve*) statement must be used for the selected service discipline to be effective.

service discipline functions -

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)

use(f, t);

uses facility f for time interval t (expressed as a number of type float); the service discipline function supplied in the facility statement is invoked when the *use* statement is executed; f must have been declared to be of type FACILITY and initialized with a *facility*, *facility_ms* or *facility_set* statement; the *use* is similar to the *hold* statement in that it also allows simulated time to pass.

t = timeslice(f);

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

set_timeslice(f, t);

set the timeslice 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.

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.

Data Collection -

note_entry(qt);
note_exit(qt);

these two statements are used to collect data which look like queue length data; qt must be of type QTABLE (or QHIST) and initialized by the *qtable* (or *qhistogram*) 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.

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

****
facility information -

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

****

facility usage functions (all of type float except completions()) -

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;

****
server statistics -

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

facility class statistics -

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;

storage statistics and information

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_busy_amt(s);	returns the busy time-weighted sum of amounts
n = storage_number_amt(s);	returns time-weighted sum of requests
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

class statistics and information

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

****

table and histogram statistics functions (all but table_cnt and table_name are of type float; table_cnt is of type int and table_name is of type char*) -

s = table_name(t);	pointer to name of table t;
n = table_cnt(t);	number of values collected in table t;
x = table_mean(t);	mean of values collected in table t;
x = table_min(t);	minimum of values collected in table t;
x = table_max(t);	maximum of values collected in table t;
x = table_sum(t);	sum of values collected in table t;
x = table_sum_square(t);	sum of squares of values collected in table t;
x = table_var(t);	variance of values collected in table t;

****

histogram functions (in addition to table functions):

n = histogram_bucket(t, i);	contents of bucket i of table t;
x = histogram_high(t);	value of highest bucket of table t;
x = histogram_low(t);	value of lowest bucket of table t;
n = histogram_num(t);	number of buckets of table t;
x = histogram_width(t);	width of each bucket of table t;

Note: if a histogram has n buckets, then bucket[0] is the number of entries less than the lowest value, bucket[n+1] is the number of entries greater than or equal to the highest value, and bucket[i] is the number of entries in the range (low + width*(i-1)) and (low + width*i)).

****

qtable and qhistogram statistics functions:

s = qtable_name(qt);	pointer to name of qtable qt;
n = qtable_cnt(qt);	count of 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 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).

****
qhistogram functions (in addition to qtable functions):

n = qhistogram_bucket_cnt(qt, i);	contents of i-th state count;
x = qhistogram_bucket_time(qt, i);	contents of i-th state time;
x = qhistogram_time(qt);	elapsed time
n = qhistogram_num(qt);	number of states

Note: if a qhistogram has n states, then state 0 is the time spent in state 0 and state n is time spent in state n or greater; the percent of time in state i is qhistogram_bucket_time(qt, i) divided by qhistogram_time(qt).
****

Reports -

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

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.

report();
causes a complete report to be printed; this report includes facility usage data, storage data, and 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.

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

rerun();

allows the model to be rerun;
****
all processes are eliminated; all facilities, storages, events, mailboxes, process_classes, tables and qtables established before the first create statement (the create for the first (“sim”) process) are reinitialized; the remaining facilities, storages, events, etc., are eliminated; the clock is set to zero; only the random number generator (rand()) is not reset; the intent is to allow a new model (or the same model with modifications) to be run as part of a program; however, special provisions have to be made for C++ programs with static objects which are automatically constructed before the program begins execution.
****

reset();
causes all of the statistics gathering fields to be cleared; in addition to statistics at facilities, storages and process_classes, 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.

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

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.

Random Numbers -

In the following, the variables y, u, u1, u2, v and s are of type float; i, i1 and i2 are of type int.

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.

Runtime Options and Debugging -

****

There are two distinct ways of writing CSIM programs; the standard way is for the 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”.

****

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.

****

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 also 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.

****
dump_status();
prints a report on the status of the model; this report is intended to be self explanatory; dump_status() makes calls to the following routines:

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

There is some flexibility in how CSIM errors are handled. The default is for a CSIM routine to print and error message, print a usage report and then dump the status of the model. An alternative way is for the program is install an *error_handler*, to respond to errors. The routine to do this is

set_error_handler(proc);
where proc should be declared as
void (*proc)();
proc is called when a CSIM error is detected; the argument to proc is an integer which is the number of the CSIM error encountered. The routine
print_csim_error(n);
can be used to print the error message on stderr.

A CSIM program can cause different kinds of output to be directed to different files. The standard output (produced by report procedures) is directed to a file specified by the FILE pointer named output_file. Similarly, errors are directed to error_file, and trace output to trace_file. These three file pointers are set at initialization time to stdout. The log file (for the MONIT option) is named by default "csim_log"; this name can be changed by the set_log_file_name() procedure. After initialization time, they can each be set by using the appropriate procedure, as follows:

    set_output_file(fp);
    set_error_file(fp);
    set_trace_file(fp);

where, in each case, fp is a pointer to an "opened" output file. (I.e., fp = fopen("file_name", "w")).

    set_log_file_name(string);

sets the name of the log file.
****

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. MAXIMUM SIZE OF HISTOGRAM EXCEEDED -
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 was process #1 (sim), 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 (such as a "runaway" program loop).
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 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 QUEUES 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 -
42 STACK UNWIND FAILURE - HPPA (INTERNAL ERROR)
43. ODD OR SMALL STACK LENGTH - HPPA (INTERNAL ERROR)
44. SET_STACK ROUTINES MAY NOT BE INVOKED AFTER CALLING CREATE - HPPA
45. UNRECOVERABLE STACK OVERFLOW - HPPA

46. INITIAL STACK SIZE TOO SMALL - HPPA

Using CSIM

At MCC, CSIM is located on on some of the SUN and DEC 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*. *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.

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 storage (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. 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.

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 declared (if more than one process class (the default class) has been declared).
5. A summary for each table (and histogram) declared, and
6. 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 CSIM 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 system 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 (requests 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 a 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	of values recorded
variance	of values recorded
min	minimum of values recorded
max	maximum of values recorded
number	of entries
histograms	
bucket	boundaries (low value - high value)
number	of values in each bucket
percentage	of values in each bucket
cumulative	percentage
total	number of entries

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

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.

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..

Converting Old Programs (Revision 10 and before)

The following steps have been shown to be useful in converting old CSIM programs (Revision 10 and earlier) to the new form, for use with new revisions:

- The first thing is to make certain that the correct include file is being referenced.
- The biggest change is to convert all of the *int* variables which are pointers to CSIM data structures to the appropriate new types. The new types are EVENT, FACILITY, HIST, MBOX, QHIST, QTABLE, STORE and TABLE.
- All calls to the *mailbox* function need to be modified, to supply a name for the mailbox, as in “mb = mailbox("mb");”.
- You need to think about the limits (in the new CSIM) and try to determine if you need to raise (or lower) the default limit values.
- If you used inputf, cmdinput, checkcmd, or testoption, you must write your own routine to handle command line parameters.
- If you reference the variable *clock* or the function *time*, you might need to make a few more changes. Just remember that *clock* is of type TIME (double) and that *time()* has been changed to *simtime()*.
- You should try to run some test cases, possibly with the old and the new revisions of CSIM. If you do this, be certain to read the pertinent section in the preamble.

Preambles to Earlier Revisions

The major changes found in Revision 11 of CSIM are as follows:

1. The different objects (data structures) used in a CSIM program are now declared as C *typedefs*. In particular, events, facilities, mboxes, histograms, qhistograms, qtables, storages, and tables must be declared with the correct types. Failure to do this will cause the C compiler to issue warnings.
2. Almost all of the internal data structures of a CSIM program are now dynamically allocated during the execution of the program. This means that one version of the CSIM library routines can accommodate any size CSIM model. There are limits on the numbers of individual kinds of CSIM objects, but these can be changed by executing runtime statements (using the “max_” functions). Of course, there are limits on the size of model which are imposed by the amount of memory and swap space available on a system. The calls to the system routine *malloc* are now checked for valid returns, and an error is detected if a malloc failure occurs.
3. The MONIT event logging facility is now integrated into the CSIM library. The interested user can refer to the appropriate MONIT documentation (see List of References) to make use of this facility for monitoring the performance of the system being simulated.

4. The *mailbox* function has been modified to require a "name" argument. This was done to maintain consistency with all of the other data structures and because the MONIT feature needed this name.

5. The command line options which are interpreted by the CSIM routines have been altered. The *inputf* feature has been eliminated, as have the *cmdinput* and *checkcmd* statements. The runtime command line options -T and -L can be used to invoke the debug trace and the MONIT logging features respectively. Other command line options are passed on to the user's program, using the conventions which are normal for C programs.

6. In Revision 11, each instance of a process has a unique process id. In previous revisions, these id's were recycled as processes terminated.

7. The function *time()* (which returned the current value of the simulated clock) has been renamed *simtime()*. This was done to avoid naming conflicts on some systems.

8. The *tabulate()* statement has been removed.

9. *Rerun()* and *reset()* have been modified, to work correctly with the new forms of CSIM data structures.

10. *Report* and *report_qtable* have been modified, to allow processes still holding a storage or still at a queue (in the sense of a qtable) to contribute to the statistics being tabulated. The statistics for facilities and for tables have not been modified.

11. The error messages have been revised. Most have new numbers; some have been deleted; and some have been added.

12. An error dealing with the ordering of events in simulated time has been corrected. This error caused events occurring at large simulated times (e.g., at times greater than about 120.0 time units) to be misplaced in the next-event list; the time of the event was being truncated to single precision as the event list was searched; the double precision time was stored correctly, but the position of the event on the list could be incorrect. This would cause some events to be slightly misordered in simulated time. The impact on a user is that models converted to run with Revision 11 could produce slightly different answers when executed with the same parameters (when compared to the equivalent Revision 10 run). In all testing so far, these differences, when they have been observed, have been very small.

Revision 12 provides some new statements. In addition, a number of steps have been taken to improve the runtime performance of CSIM programs. This preamble details only changes to the user interface.

1. Two new statements, *reset_table* and *reset_qtable* have been added.

2. A new statement, *set_model_name*, has been added.

3. Three new statements, *timed_queue*, *timed_receive*, and *timed_wait*, have been added. These allow a process to suspend until either the specified event has occurred or until the specified amount of time has elapsed.

4. The procedure *reset_prob* has been added; this resets the random number generator and must be used instead of *srand()*.

5. The *mdlstat()* procedure has been modified, to give a clearer description of the resources being used by a CSIM program.

6. The statement *global_event* has been added. This statement allows a process other than *sim* to initialize a global event.

7. The statements *report_facilities*, *report_storages*, and *report_tables* have been added. These statements allow the programmer to print a report of the specified class of simulation structures without getting an entire report.

8 The *testoption* statement has been removed.

9. A few errors have been fixed. One involved the “max_” functions; the limit checks were off by one. Another involved the rerun statement, which did not work correctly in Revision 11.

Revision 13 provides the following enhancements and modifications:

1. The report_hdr() statement has been added.
2. The delete_event_set() and event_qlen() statements has been added.
3. The completions(f) function has been added; this function returns the number of completions (releases) at the specified facility.
4. The reserve(f) function has been made into a function; the returned value is the index of the server assigned (see release_server(f) below).
5. A new procedure, release_server(f, i) has been added; this procedure lets any process release the specified server; the purpose of this addition is to let one process pass “ownership” of a facility (really a server at a facility) to another process.
6. Permanent tables, qtables, histograms and qhistograms are now part of CSIM. The new functions permanent_table(), permanent_qtable(), permanent_histogram() and permanent_qhistogram(), establish tables, etc. which will not be affected by reset, rerun and clear_tables statements. This feature allows data to be collected across multiple runs within a model.
7. Histograms and qhistograms are now dynamically allocated; thus, they can be of any size; the function max_sizehist() has been added; it can be used to change the maximum size allowed for an individual histogram.
8. Event sets have been modified, so that a process can wait on the occurrence of any event in a set of events. Three new statements dealing with event sets have been added: wait_any(), queue_any() and delete_event_set();
9. The value of the constant BUSY is now 1 (the documentation said this but the status function returned -1).
10. A new function named num_busy() has been added; this function returns the number of busy servers at a facility.
11. Two new functions, table_sum and table_sum_square, return the sum of the values and the sum of the squares of values, respectively, in a table.
13. Two functions have been added which return the number of processes at an event: wait_cnt() returns the number waiting, and queue_cnt() returns the number queued.
14. A set of functions for extracting statistics from qtables and qhistograms has been added.
15. A “special”, built-in event has been added; this event, named “event_list_empty”, is set if the internal next-event list becomes empty and one or more processes is *waiting* at this event. This event can be used to wait for all of the processes to leave a system (model).
16. CSIM can now be used with a SUN 386i workstation.
17. Management of the next-event-list in CSIM has been changed to utilize the "calendar queues" algorithm by Brown [Brow88]. The result is that model with many active processes should run much faster; however, models with only a few active processes might run slightly slower.
18. Several bugs have been detected and fixed. These include:
    a. Problems with rerun - storages, qhistograms, qtables.
    b. A problem with the set_loaddep(f, arr, n) procedure has been fixed (thanks to Mr. Khanna, Univ. of Texas).
    c. The prc_shr() procedure has had a delete_event() statement added (thanks to Teemu Kerola, Univ. of Helsinki).
    d. A problem with timed_receive(): there was an error in keeping track of the number of waiting processes (thanks to Conrad Kwok at UC Davis).

Revision 14 provides the following enhancements and modifications:

1. The Sun 4 and Sun 386i workstations, the HP Series 300 workstation, the DEC DecStation 3100 (MIPS chip) and the Sequent Symmetry system are now supported. Also, the NCR Tower 650 and a 386 PC (and also a 486) with the Xenix operating system are supported.

2. Drop support for the Celerity.
3. Correct the description of delete_event(ev) in this document.
4. Add a check in deallocate(), to check for a negative value for the current number of using processes.
5. Correct the description of timed_receive() in this document.
6. Add note to description of max_mailboxes() in this document.
7. Removed tabulate() as a synonym for record().
8. Fix a bug in log.c (xsuspend_log).
9. Fix a bug in cprim.c (dealing with timed_receive).
10. Fix a bug in machdep.c (truncations in stack address computations in x_create).
11. Add trace_msg() procedure.
12. Add provisions for use with C++.

Revision 15 offered the following:

1. Fix a bug in prc_shr (the old version gave incorrect holding times in some cases).
2. Fix a bug in the Sun4 version; if a program had too large a call stack and was suspended, then a random runtime error could occur when it was restored.
3. Modify the calendar queue routines (enq_evl, etc.) to use tail pointers; the net effect should be to speed up programs which schedule many events at the same point(s) in simulated time.
4. Provide support for C++ (AT&T Release 2.0) and Gnu C++. Also, modified library to support compilation with the Gnu C compiler (gcc).
5. Added new routines set_name_event(), set_name_facility(), set_name_mailbox(), and set_name_storage().
6. Fixed a bug in 386 PCUNIX version; register variables were not begin handled correctly.
7. Added support for 486 PCUNIX.
8. Added process classes to csim. The new routines added to support classes are process_class(), collect_class_facility(), collect_class_facility_all(), set_process_class(), max_classes() and report_classes();
9. Added check in facility_ms() for zero or negative number of servers.
10. Fix bug in processor sharing (prc_shr()) so that report correct utilization and average queue length.
11. Change the reset() procedure so that time intervals tabulated for facilities, storages and qtables are the full intervals, rather than intervals which are shortened by having their start-times set to the time of the reset (which has been the case).
12. Added a count of the number of events processes during the run to the mdlstat set of statistics.
13. Fix bugs in the timed_receive, timed_wait and timed_queue functions; these bugs caused some processes to get “lost” and simulated time to be incorrectly altered (infrequently).
14. Fix a bug in event_set’s and wait_any() and queue_any(). This bug was reported by Raj Yavatkar, at the University of Kentucky.

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.