# Nest Library Reference Manual

Alexander Dupuy
Jed Schwartz

Computer Science Department
Columbia University
New York, NY 10027-6699

Wednesday June 15th, 1988

CUCS-374-88

## Abstract

This manual describes the simulation library provided with Nest Version 2.5. Nest is available from Columbia University. For information, please contact the authors.

This research was supported in part by the Department of Defense Advanced Research Project Agency, under contract F29601-87-C-0074, and by the New York State Science and Technology Foundation, under contract NYSSTF CAT (87)-5.

# Table of Contents

1. Initialization and Simulation Functions 1
  1.1. Initialization Routines 1
    1.1.1. main 1
    1.1.2. nest_parse_args 2
    1.1.3. nest_read_graph 2
    1.1.4. simulate 3
  1.2. Monitor Routine 3
    1.2.1. nest_monitor 3
2. Communications Functions 5
  2.1. Sending and Receiving Messages 5
    2.1.1. sendm 5
    2.1.2. recvm 5
    2.1.3. recvmt 6
    2.1.4. any_messages 6
  2.2. Channel Functions and Function Stacks 6
    2.2.1. reliable 6
    2.2.2. channel_sendm 7
    2.2.3. safe_string 7
3. Node Execution Functions 9
  3.1. Node Timing Functions 9
    3.1.1. runtime 9
    3.1.2. cputime 9
    3.1.3. slumber 9
    3.1.4. stop_time 9
    3.1.5. start_time 10
    3.1.6. advance 10
  3.2. Mutual Exclusion Functions 10
    3.2.1. hold 10
    3.2.2. release 10
4. Utility Functions 11
  4.1. Network Configuration Functions 11
    4.1.1. get_node_id 11
    4.1.2. get_location 11
    4.1.3. get_neighbors 11
  4.2. Graphlanguage Utilities 12
    4.2.1. writegraph 12
    4.2.2. readgraph 12
    4.2.3. freegraph 12
    4.2.4. nest_save_graph 13
    4.2.5. nest_restore_graph 13
  4.3. Socket Communication Functions 13
    4.3.1. nbsocket 14
    4.3.2. nbconnect 14
    4.3.3. nbconn 14
    4.3.4. nbwrite 15
    4.3.5. nbsend 15
    4.3.6. nbrecv 15
    4.3.7. nbread 16
    4.3.8. discard 16
    4.3.9. firstfd 16
    4.3.10. Macros for fd_set structs 16
  4.4. Time Utility Functions 17
    4.4.1. atotv 17

# 1. Initialization and Simulation Functions

This section describes the functions used to start the simulation loop of the Nest library, and the monitor routine which can be used for communications management, logging, or network modification.

## 1.1. Initialization Routines

### 1.1.1. main

```
main (argc, argv)
int             argc;
char            *argv[];
```

`Usage: program [-n` *nodes*`] [-s` *stacksize*`] [-p` *port*`] [-f` *file*`]`

As in any Unix program, the user can write a main() routine which takes command line arguments and provides the overall program structure with calls to other routines to do initialization or processing. In most cases, the user of the Nest library will want to do this, but for convenience and fast prototyping, a generic main() routine is provided.

The purpose of the generic main() which we provide, or a user-written main() function, in Nest is to create an initial graph and then call the simulate() function, passing it this graph as an argument. Note that the Nest graph includes not only topology but also header information including some simulation parameters.

The generic main() routine lacks flexibility, but parses command line arguments specifying simulation parameters, and will prompt for any unspecified parameters when run. If the generic main() is used, the only function that will be run on the nodes will be the node_main() routine. All edges will have the function channel() as their channel function(). These two routines must be supplied by the programmer, and are the only functions which need be provided.

The main() routine determines the following parameters which must be passed to the simulation loop: maximum number of nodes allowed, total stack size, internet port number on which to listen for connections, and initial network graph. These can be specified by command line arguments, or supplied by the user in response to prompts from the program.

The -n option sets the maximum number of nodes, the -p option sets the port number on which the simulation should listen, and the -s option sets the initial size (in bytes) reserved for the stacks of the simulation nodes. The -f option specifies a file containing a graphlanguage description of the initial network configuration. This file can be created with the nest_save_graph() routine, or with the Nest user interface client.

If -n option is unspecified, the generic main() will prompt the user for a value. If the -s or -p options are unspecified, the main routine will default them to 1024 times the number of nodes, and the port returned by getservbyname("nest","tcp"), respectively. If the -f option is unspecified the user is prompted for the specifications of nodes and edges.

The main() function is in main.c.

### 1.1.2. nest_parse_args

```
int nest_parse_args (argc, argv)
int             argc;
char           *argv[];

graph          *Graph;
unsigned        Nodes;
unsigned        StackSize;
int             PortNumber;
```

The nest_parse_args() function parses command line arguments in the same way as main(), initializing the global variables Nodes, StackSize, PortNumber and Graph as specified by the given command line arguments so that these can subsequently passed as arguments to simulate(). The function allows a user provided main() to parse the standard command line arguments in the same way the generic main() function does.

The global variable Graph should be initialized with an empty graph, containing only a graph header and function lists, before nest_parse_args() is called. This allows the nest_restore_graph() routine which is invoked by the -f option to properly set up the function values for nodes, edges, and the monitor function with addresses which are valid for the current simulation. Failure to do this may result in coredumps if a graph file from another version of the simulation is loaded.

If unrecognized command line options are given, a message is printed on stderr, and the return value status is -1. Otherwise the return value is 0. The two arguments argc and argv correspond to the standard arguments to main(), which main should pass to nest_parse_args().

The nest_parse_args() function is in nestargs.c.

### 1.1.3. nest_read_graph

```
graph *nest_read_graph (nodes, nodefn, chanfn, monfn)
unsigned        nodes;
int             (*nodefn) ();
int             (*chanfn) ();
graph          *(*monfn) ();
```

The nest_read_graph() function interactively prompts the user for an initial graph, including both graph topology (i.e. nodes and edges) and header flags. It takes the following arguments:

- nodes - the maximum number of nodes which the interactive user can specify. This value should be the same (or less than) the nodes argument which will subsequently passed to simulate().
- nodefn - the function which will be assigned as the node function for all of the nodes in the graph created by the user.
- chanfn - the function will be assigned as the channel functions for all the edges in the graph created by the user.
- monfn - the automatic monitor function which will be used in the simulation.

The nest_read_graph() function prompts the user on the standard error output, and reads from standard input, returning a complete Nest graphlanguage graph initialized in accordance with the arguments passed to it and the values supplied by the user.

The nest_read_graph() function is in gutils.c.

### 1.1.4. simulate

```
int simulate (nodes, size, initgraph, portnumber)
unsigned        nodes;
unsigned        size;
graph          *initgraph;
int             portnumber;
```

The simulate() function invokes the Nest simulator and returns upon completion of the simulation. If the simulation finishes normally, i.e. if all node functions return or if they deadlock waiting for messages, the function will return 0. If the simulation aborts due to a fatal error such as running out of stack space for the nodes, it will print an error message and return -1.

It takes four arguments: nodes, size, initgraph and portnumber.

The nodes argument represents an absolute limit on the number of nodes; dynamic reconfiguration of the network will never allow more than this number of nodes.

The size argument tells the simulation how much stack space to allocate initially. This number represents not the maximum size for any one node, but the maximum total size for all the stacks at any one time. If during the simulation, more space is needed, the simulation will try to get more space via realloc(). Therefore, there is no reason to overestimate this figure, since if it is too small, the simulation will not fail, unless there is no more memory available.

The initgraph argument specifies the initial network configuration as a graphlanguage structure. The nest graphlanguage structure is discussed in more detail in the appendix. While this argument can be left nil, the default initial graph will probably not be what you expected. A better idea is to use the nest_read_graph() routine to generate a graph at runtime.

The portnumber argument specifies the internet port number which the server routines will listen to for connections. It should be in network byte order (nest_parse_args() and getservbyname() return ports in network byte order). A value of -1 will disable the port. A good way to get a port number (rather than hard coding in a number) is to use the getservbyname() routine to look up an entry in the /etc/services database. You can then ensure that you are not using the same port number as anyone else, and can change it without recompiling or editing.

The simulate() function is in simulate.c.

## 1.2. Monitor Routine

The automatic monitor routine selected in the Nest graphlanguage header is called at the beginning of every pass of the simulation. You can supply any number of monitor functions, include them on the list of available monitor functions and switch between them while the simulation is running using the client program or the monitor function itself. A generic monitor function nest_monitor() is provided.

### 1.2.1. nest_monitor

```
graph *nest_monitor (oldgraph)
graph             *oldgraph;
```

The function accepts a single argument, oldgraph, which is the current graphlanguage model of the simulation state. The function then communicates with client monitors as described below in order to report to them the current simulation state. The function then returns oldgraph, unmodified.

The nest_monitor() function will send graphlanguage messages to clients (if any) when they initially connect to the simulation, and when the status of any nodes change from running (or blocked or paused) to dead or done. If it detects that all the nodes in a simulation are blocked indefinitely or dead or done, and clients are connected, it pauses the simulation so that the user can make modifications to the network and continue.

Note that, while nest_monitor does not make any changes to oldgraph, a user-implemented monitor function may do so in order to modify the graph topology or simulation state under program control. If such modifications are made to the graph which is returned by the user monitor function, then these changes will be made to the simulation.

The nest_monitor() function is in nestmon.c.

# 2. Communications Functions

## 2.1. Sending and Receiving Messages

Messages are implemented in Nest as two 32-bit quantities, by convention an integer key and a pointer to data, although both can hold either pointers or integers. Each message has a header associated with it, consisting of the node ids of the (initial) sender and destination.

### 2.1.1. sendm

```
sendm (dest, key, dataptr)
ident           dest;
long            key;
pointer         dataptr;
```

The function takes as arguments a destination or receiver node, a key which represents the message type as interpreted by the receiver, and a pointer to the message data in memory.

The destination may be any number between 1 and the maximum node number for the simulation (even if this number does not correspond to a currently running node). Also, if the point-to-point flag is not set to true, 0 may be chosen as a destination, which will result in the message being sent to all neighbors. Also, if the broadcast flag is set to true, the message will be sent to all neighbors regardless of the specified destination.

Note that one or neither, but not both, of the broadcast and point-to-point flags should be set to true at a given time. If both are set, all calls to sendm() will fail.

If the destination does not satisfy the above constraints, the function exits and a -1 is returned. Otherwise, the message is delivered to its destination node(s). This is accomplished by invoking the first function on the stack of channel functions for the edge connecting to each destination, and then placing the possibly transformed message on the receive queue of that destination node. The function then exits and returns the return value of the channel function stack. In the case of broadcast messages, the result returned is the additive sum of the return values of the channel function stack for each edge.

The default channel function, reliable(), returns 1 if the message is successfully transmitted, or 0 if it could not be transmitted, so that the net result of a broadcast message is the number of messages sucessfully sent. User written channel functions are of course free to return other values.

Note that if a message is delivered to a node that is currently stopped, that node will be able to receive that message if and when it begins running again.

The sendm() function is in sendm.c.

### 2.1.2. recvm

```
ident recvm (destp, keyp, dataptrp)
ident           *destp;
long            *keyp;
char           **dataptrp;
```

In order for a node function to receive a message, it must call the recvm() function. This function will return the first message to arrive at the node. Messages from different nodes are not segregated, but the sender of the message is returned by the recvm() call.

The recvm() function takes the arguments destp, keyp, and dataptrp, which are the addresses of variables which the function will set to the destination, key, and data pointer of the message. These arguments can be 0 (cast to the appropriate type) if the values are of no interest. The recvm() function returns the id of the sender of the message.

If no messages are available yet (determined by the implicit arrival time field in messages), a node which calls recvm() will block until one becomes available (that is, it will stop being scheduled, but will appear to be using simulated run time).

The recvm() function is in recvm.c.

### 2.1.3. recvmt

```
ident recvmt (destp, keyp, dataptrp, sendt, arrvt)
ident           *destp;
long            *keyp;
char           **dataptrp;
timev           *sendt;
timev           *arrvt;
```

The recvmt() function is functionally equivalent to the recvm() function, but takes two additional arguments, which can be used to get the time at which a message was sent and the time at which it arrived. These arguments can be 0 (cast to the appropriate type) if the values are of no interest. In all other respects, recvmt() is identical to recvm().

The recvmt() function is in recvmt.c.

### 2.1.4. any_messages

```
int any_messages()
```

The any_messages() function takes no arguments. It returns 1 if any messages are available for receipt by the node calling the function, and returns 0 otherwise.

The any_messages() function is in anymess.c.

## 2.2. Channel Functions and Function Stacks

When a message is sent from one node to another, the actual delivery is done by channel functions. Each edge has a list of channel functions called the channel function stack, or channel stack. A call to sendm() causes the first function on the channel stack to be called with the message, sender, and destination arguments to sendm(), the weight of the edge, and the remainder of the channel function stack. This channel function typically passes these same arguments, modified as desired, to the next function on the stack, although this behavior is not enforced. Thus, in addition to modifying the message data pointed to by dataptr, a channel function can change any of the parameters before passing them to the next function on the stack, thus, for example, disguising the apparent sender of the message by changing the value of sender.

### 2.2.1. reliable

```
int reliable (a_node, sender, dest, key, dataptr, delay, chans)
ident           a_node;
ident           sender;
ident           dest;
long            key;
pointer         dataptr;
long            delay;
int             (*chans[]) ();       /* array of function pointers */
```

This function is the prototypical channel function. It is the ultimate delivery agent for messages, and is implicitly at the end of all channel stacks (see channel_sendm(), below). The function delivers a message to the receive queue of the destination node after a simulated time delay as specified by the delay argument, and without performing any transformations on the message data. If the delay argument passed to the function is non-zero, then this value is used as the delay in delivering the message. However, if the delay argument is 0, then the delivery delay used is the delay in the graph header.

The a_node argument is the actual recipient node for the message. This must be a valid node id, i.e. greater than 0, and less than or equal to the simulation's maximum node number. It is not required to be connected to the sending node. If this argument is invalid, reliable() will return 0.

The sender argument is the sending node id; the dest, key and dataptr arguments are from the initial sendm() arguments; and the delay argument is the weight specified in the graph's edge information; but any of these may be changed by channel functions earlier in the stack. No checking is done on these arguments.

The chans argument is the remainder of the channel stack. The reliable() channel function ignores this argument (it can safely be left out) since it delivers the message itself, but the argument is present in all calls to channel functions. It is an array of function pointers, terminated by a nil pointer.

The reliable() function is in messes.c.

### 2.2.2. channel_sendm

```
int channel_sendm(to, sndr, dest, key, dptr, weight, channels)
ident           to;
ident           sndr;
ident           dest;
long            key;
pointer         dptr;
long            weight;
int             (*channels[]) ();     /* array of function pointers */
```

The channel_sendm() macro is provided as a standard method for channel functions to pass the message on to the remainder of the channel stack. It has the same calling sequence as other channel functions, but its behavior is rather different. It pops the first function pointer from the channels argument, and if the pointer isn't nil, it invokes that function with the other arguments, including the modified chans argument. If the pointer is nil, the reliable() function is invoked instead.

The channel_sendm() macro is in nest.h.

### 2.2.3. safe_string

```
int safe_string(a_node, sender, dest, key, dataptr, delay, chans)
ident           a_node;
ident           sender;
ident           dest;
long            key;
pointer         dataptr;
long            delay;
int             (*chans[]) ();        /* array of function pointers */
```

The safe_string() channel function changes only its dataptr argument. It assumes that dataptr is a pointer to a null-terminated string, and replaces it with a pointer to a copy of the string. The copy is created with malloc(), and copied with strcpy(). The altered message is then passed to channel_sendm().

The safe_string() function is in safestr.c.

# 3. Node Execution Functions

## 3.1. Node Timing Functions

Since functions running under the Nest simulation library are timesharing a Unix process, the standard Unix functions involving process times will not work properly. Therefore, Nest has several functions to provided simulated timer interfaces using the timev type, defined in dcctime.h to be a Berkeley timeval struct (sys/time.h).

Do *not* use these functions in monitor or channel functions.

### 3.1.1. runtime

```
timev runtime ()
```

The function runtime() takes no arguments, and returns the total simulation time of the node which called the function since the beginning of the simulation. This time includes time spent waiting to receive messages, sleeping, and not running, in addition to time spent running. This time is not reset when a node starts running again, nor is it affected by switching the node function running at the node.

The runtime() function is in the file runtime.c.

### 3.1.2. cputime

```
timev cputime ()
```

The function cputime() takes no arguments, and returns the current cpu time used for the local node since the beginning of the simulation. This time does *not* include time spent waiting to receive messages, sleeping, or not running. This time is not reset when a node starts running again, nor is it affected by switching the node function running at the node. The function is in the file cputime.c.

### 3.1.3. slumber

```
int slumber (naptime, wakeup)
timev          *naptime;
flag            wakeup;
```

The slumber() function is a replacement for the Unix sleep and select functions (which should *not* be called in the simulation). The function takes two arguments, naptime and wakeup. The function will block the node for an amount of time pointed to by naptime. If naptime is nil, the node will block indefinitely. Note that this is different from a naptime argument of 0 seconds, which does not block the node.

If wakeup is true, then if any messages arrive while the node is blocked, it will be awakened, and the slumber function will return 1. Note that pending messages, as indicated by any_messages() returning 1, will not prevent a node from slumbering. If no messages arrive while the node is slumbering, it returns 0.

The time spent slumbering is charged as simulation runtime, but not cputime, for the node.

The slumber() function is in slumber.c.

### 3.1.4. stop_time

```
stop_time ()
```

This function is provided to turn off time accounting for simulation processing time used by a node. It takes no arguments. After this has been called, any simulation cpu time used by the node will not count against its cputime or runtime until start_time() is called (however, time spent in blocked for recvm or slumbering will count against runtime). No value is returned.

The stop_time() function is in stoptime.c.

### 3.1.5. start_time

```
start_time ()
```

This function is provided to turn on time accounting for simulation processing time used by a node. It takes no arguments. After this has been called, the node will once again be charged for any cpu time it uses. No value is returned.

The start_time() function is in stoptime.c.

### 3.1.6. advance

```
advance (addtime)
timev           addtime;
```

This function is used to charge time to a node. It works much in the way that slumber does, except that it can't be interrupted, and the time is charged as cpu time as well as runtime. It takes one argument, a time struct (*not* a time pointer) which is to be charged to the node. The node will be charged for this time whether stop_time() has been called or not.

This function can be used with stop_time() in effect to provide an artificial measure of cpu time usage which is independent of machine speed.

The advance() function is in advance.c.

## 3.2. Mutual Exclusion Functions

### 3.2.1. hold

```
hold ()
```

This function can be used by a node function to enter a critical section. The hold() function prevents the node from being interrupted. The function takes no arguments and returns no values. Multiple calls to hold() can be made, and calls after the first only increment a hold count.

Critical sections which should not be interrupted include accesses of any global data which may be modified by any other node (this sort of data sharing is best avoided altogether), and calls to library functions which are non-reentrant, (this includes any function which calls malloc()).

The hold() function is in handlers.c.

### 3.2.2. release

```
release(key)
int             key;
```

This function is used to exit a critical section entered with a call to hold(). The function takes a single argument, key. If key is 0, it will release all hold() calls for the node. If it is 1, it will only release one hold() call.

The release() function is in handlers.c.

# 4. Utility Functions

## 4.1. Network Configuration Functions

This section documents a number of utility functions which are designed to be used by node functions running under the Nest simulation library. Many of these functions take a node id as an (only) argument which specifies the node in question. Except where noted otherwise, when these functions are called by routines running on a node, 0 can be used as a node id for the common case of the local node.

When the function for a node is called, it is passed only one argument; its node id. In order for a node to find out more about the network configuration several functions are provided to simulate the local information which a node might have.

### 4.1.1. get_node_id

```
ident get_node_id ()
```

Whenever a node function is called, it is passed a single argument which is the node it is running on. But since it would be inconvenient and inefficient to pass the node id to every subfunction on a node which might need to know it, the get_node_id() function is provided.

This function takes no arguments. If called by a node function, it returns the local node id. If called from a channel function, it returns the sending node. If called from a monitor function, it returns 0.

The get_node_id() macro is in nest.h.

### 4.1.2. get_location

```
position get_location (nodeid)
ident           nodeid;
```

The location of a node and its neighbors, as found in the graphlanguage description of the network, can be determined with the get_location() function. This function takes one argument, a node id, and returns a position struct with integer x, y, and z coordinates as defined in graphlang.h. The argument can be any valid node id, not just the local one. Note that this function returns a position struct, *not* a pointer to a position struct.

The get_location() function is in getloc.c.

### 4.1.3. get_neighbors

```
node_ptr get_neighbors (nodeid)
ident           nodeid;
```

A list of the node ids of the connected neighbors of a node can be obtained with the get_neighbors() function. This function takes one argument, a node id, and returns a pointer to a node_item struct as defined in graphlang.h. The argument can be any valid node id, not just the local one. The list returned is created with malloc() and sorted by node id. It should be freed when no longer needed.

This function should *not* be used by a monitor or channel function.

The get_neighbors() function is in getnbrs.c.

## 4.2. Graphlanguage Utilities

### 4.2.1. writegraph

```
byte *writegraph (oldgraph, header_fnx, node_fnx, edge_fnx)
graph            *oldgraph;
word              (*header_fnx) (),
                  (*node_fnx) (),
                  (*edge_fnx) ();
```

The writegraph() function takes as arguments a pointer to a graph, oldgraph, and the addresses of three functions. Header_fnx is a function which will write out a graphlanguage message bytestream corresponding to a graph's header. Node_fnx is a function which will write out a graphlanguage message bytestream corresponding to a node in a graph. Edge_fnx is a function which will write out a graphlanguage message bytestream corresponding to an edge in a graph.

The writegraph() function allocates a buffer of appropriate size using malloc and then uses these three functions to fill the buffer with a bytestream containing a graphlanguage message which corresponds to oldgraph. The function returns a pointer to this buffer. Note that the function does not free, or any way modify, oldgraph.

The writegraph() function is in graphs.c, as are the three functions (writeheader, writenode and writeedge) which usually passed to writegraph() when called by the Nest library.

### 4.2.2. readgraph

```
graph *readgraph (bytes, header_fnx, node_fnx, edge_fnx)
byte             *bytes;
grhead           *(*header_fnx) ();
grnodedat        *(*node_fnx) ();
gredgedat        *(*edge_fnx) ();
```

The function takes as its first argument a pointer to a buffer containing a bytestream which is a graphlanguage message corresponding to a graph. Its additional arguments are the addresses of three functions. Header_fnx is a function which will read a bytestream and create a corresponding header data structure. Node_fnx is a function which will read a bytestream and create a corresponding node data structure. Edge_fnx is a function which will read a bytestream and create a corresponding edge data structure.

The readgraph function allocates a graph structure using malloc and then uses these three functions to assign the appropriate values to the fields within the graph. The function returns a pointer to the created graph. Note that the function does not free, or any way modify, the graphlanguage message buffer.

The readgraph() function is in graphs.c, as are the three functions (readheader, readnode and readedge) which usually passed to readgraph() when called by the Nest library.

### 4.2.3. freegraph

```
freegraph (deadgraph, header_fnx, node_fnx, edge_fnx)
graph            *deadgraph;
funcptr           header_fnx,
                  node_fnx,
                  edge_fnx;
```

The function takes as arguments a graph, deadgraph, and the addresses of three functions. Header_fnx is a function which will free any additional storage associated with a graph header. Node_fnx is a function which will free any additional storage associated with a graph node data structure. Edge_fnx is a function which will free any additional storage associated with a graph edge data structure. Freegraph frees deadgraph, and uses these functions to free any additional storage associated with the graph.

The freegraph() function is in graphs.c, as are the two functions (freeheader and freeedge) which usually passed to freegraph() when called by the Nest library. (Since there is no additional storage associated with a node, there is no freenode() function).

### 4.2.4. nest_save_graph

```
nest_save_graph (filename, oldgraph, header_fnx, node_fnx, edge_fnx)
char            *filename;
graph           *oldgraph;
word             (*header_fnx) (),
                 (*node_fnx) (),
                 (*edge_fnx) ();
```

The nest_save_graph() function takes as arguments a filename, a pointer to a graph, oldgraph, and the addresses of three functions. Except for the filename argument, these are exactly the same as the arguments to writegraph ().

The nest_save_graph() function opens the file specified by filename, and writes a graphlanguage message which corresponds to oldgraph. If any errors occur, a message is printed on stderr, and -1 is returned. Otherwise, 0 is returned. Note that the function does not free, or any way modify, oldgraph.

The nest_save_graph() function is in gload.c.

### 4.2.5. nest_restore_graph

```
graph *nest_restore_graph (filename, oldgraph, header_f, node_f, edge_f)
char            *filename;
graph           *oldgraph;
grhead          *(*header_f) ();
grnodedat       *(*node_f) ();
gredgedat       *(*edge_f) ();
```

The nest_restore_graph() function takes as arguments a filename, a pointer to a graph, oldgraph, and the addresses of three functions. These last three arguments are exactly the same as the last three arguments to readgraph ().

The nest_restore_graph() function opens the file specified by filename, and reads a saved graph in graphlanguage message form. The second argument, oldgraph, should be have at least a valid header and function lists to allow the nest_restore_graph() function to properly set up the function values for nodes, edges, and the monitor function with addresses which are valid for the current simulation. If the oldgraph argument is nil, nest_restore_graph() will still work, but coredumps may occur if a graph file from another version of the simulation is loaded.

If any errors occur, a message is printed on stderr, and -1 is returned. Otherwise, 0 is returned. Note that the function does not free, or any way modify, oldgraph.

The nest_restore_graph() function is in gload.c.

## 4.3. Socket Communication Functions

These functions provide support for asynchronous i/o on non-blocking file descriptors. These can be sockets, pipes, fifos, character special devices, or regular files, although these routines are most useful for ''slow'' devices, since writes to files never block.

They are *not* reentrant for each file descriptor; you must do some sort of descriptor locking or interrupt masking if you call these from a SIGIO/SIGURG (or other signal) handler.

These functions deal with a number of incompatibilities between various versions of the Berkeley networking software, from 4.1c to 4.2 to 4.3. However, there are incompatibilities which aren't hidden; to see how one can deal with them, look at the _server_init() code in server.c.

### 4.3.1. nbsocket

```
int nbsocket (options, address)
int              options;
struct sockaddr *address;
```

The nbsocket() function can be used to create a non-blocking stream socket in the ARPA Internet domain. The nbsocket() function takes two arguments, options, which is a bitmask of desired socket options, and address, which is an address to which the socket should be bound. If the address argument is nil, the system will assign one (4.1c bsd) or none will be assigned (4.2/4.3 bsd).

Multiple socket level options can be set true, by simply or'ing them together. On 4.1c bsd, if the socket will be used to accept connections, the SO_ACCEPTCONN option must be set. On 4.2/4.3 bsd, listen() should be used instead.

4.3 socket options which take values other than the integer 1 can't be set, nor can 4.2 SO_DONTLINGER be set.

The nbsocket() function is in the file nbsock.c.

### 4.3.2. nbconnect

```
int nbconnect (fd, mask, address)
int              fd;
fd_set          *mask;
struct sockaddr *host;
```

Connecting a socket to a port is done with the nbconnect() function. It takes the file descriptor of the socket, a mask pointer, and a socket address. If a connection cannot be made immediately, it returns 0 and sets the descriptor bit in the mask so that later calls to select() will indicate when the connection has been made or has failed. At this point a second call to nbconnect() will return the correct status (either 1 or -1), and the descriptor bit will be cleared. In a second call to nbconnect() the address argument is ignored and can be 0.

The nbconnect() function is in nbsock.c.

### 4.3.3. nbconn

```
int nbconn (fd, mask, host, port)
int              fd;
fd_set          *mask;
struct hostent  *host;
int              port

struct in_addr  *nbconnaddr;
int             nbtryagain;
```

A more convenient connection interface for 4.2/4.3 systems is provided by the nbconn() function. It takes the file descriptor of the socket, a mask pointer, a host entry (such as returned by gethostbyname()), and a port number. The host and port arguments are used to create a struct sockaddr argument for nbconnect().

If a connection cannot be made immediately, it returns 0 and sets the descriptor bit in the mask so that later calls to select() will indicate when the connection has been made or has failed. At this point a second call to nbconn() will return the correct status (either 1 or -1), and the descriptor bit will be cleared. In a second call to nbconn() the host and port arguments are ignored and can be 0.

An additional feature of nbconn(), not present in nbconnect(), is that a pointer to the address (a struct in_addr) is placed in the global variable nbconnaddr. On 4.3 bsd, or other systems where multiple host addresses can be returned in a struct hostent, if a connection attempt fails, and there are more addresses which can be tried, -1 will be returned, but the global variable nbtryagain will be set true.

In this case, if nbconn() is called with a host argument of 0 (cast to the appropriate type), it will try connecting to the next address. If a non-zero host argument is given, it will use that instead.

The nbconn() function is in nbsock.c.

### 4.3.4. nbwrite

```
nbwrite (fd, bytes, length)
int            fd;
char          *bytes;
int            length;
```

The nbwrite() function is a non-blocking version of the write() system call. No data is actually sent until nbsend() is called, but it returns the length, or -1 if the arguments are invalid. Each write is separated, so that nbread() will never return more than one nbwrite of data at a time. It is possible to write more data before and during sending with nbsend().

Once nbwrite() has been used to write a message, nbsend() should be called to actually start sending it. If the global fdset variable Writes has the bit set corresponding to the descriptor which is written, Nest will eventually write this data, and clear the bit in Writes.

The nbwrite() function is in the file nbsock.c.

### 4.3.5. nbsend

```
int nbsend (fd, mask)
int            fd;
fd_set        *mask;
```

The nbsend() function is the function actually used by Nest to write the data queued by nbwrite(). Data is written from the queue to the fd file descriptor until it blocks or all of it is written. If nbsend() cannot send all the data written, it sets the descriptor bit in mask so that later select() calls will note when the descriptor is ready for additional output and returns 0. If all the data in the queue is written, the bit for the file descriptor is cleared from the fdset pointed to by mask and 1 is returned. If an error occurs, -1 is returned.

The nbsend() function is in nbsock.c.

### 4.3.6. nbrecv

```
int nbrecv (fd, mask)
int            fd;
fd_set        *mask;
```

Receiving data is done by calling nbrecv(). If a complete message is available, nbrecv() returns the length of the message. The message itself can be obtained by calling nbread(), which returns a pointer to the buffer holding the message (which can be disposed after use). Otherwise it will return 0 and set the the descriptor bit in mask so that later select() calls will indicate when the descriptor has more data to read. If end of file is detected, errno is set to EPIPE, any data received is thrown out, and the descriptor bit in mask is cleared. On this or any other error, nbrecv() returns -1.

The nbrecv() function is in nbsock.c.

### 4.3.7. nbread

```
pointer nbread (fd)
int             fd;
```

The nbread() function returns a pointer to the last complete message received by nbrecv(). The contents of the message are in a buffer created with malloc() that should be freed when no longer needed. Note also that when nbrecv() indicates that a complete message is available, nbread() should be called before any further calls to nbrecv(), or the message buffer will be lost.

The nbread() function is in nbsock.c.

### 4.3.8. discard

```
discard (fd, message)
int             fd;
char           *message;
```

The discard() function is used to get rid of a file descriptor on which an error has occurred. It takes two arguments, a file descriptor, fd, and a character string error message. The file descriptor fd is closed, and cleared from the fdsets used by Nest, and the error message is printed out on stderr using perror().

The discard() function is in server.c.

### 4.3.9. firstfd

```
int firstfd (mask)
fd_set         *mask;
```

The firstfd() function is useful in manipulating the fdsets used by Nest. It returns the corresponding descriptor number of the first bit set in the fdset pointed to by mask, and clears that bit from mask.

The firstfd() function is in server.c.

### 4.3.10. Macros for fd_set structs

The following macros for manipulating fd bit strings are in dcctypes.h:

```
zerofds (set)
fd_set         *set;
```

Clears to 0 all bits in the fd_set pointed to by set.

```
clrfd(set, fd)
fd_set         *set;
int             fd;
```

Clears to 0 the bit for fd in the fd_set pointed to by set (other bits are not modified).

```
setfd(set, fd)
fd_set         *set;
int             fd;
```

Sets to 1 the bit for fd in the fd_set pointed to by set (other bits are not modified).

```
testfd(set, fd)
fd_set         *set;
int             fd;
```

Returns the value of bit for fd in the fd_set pointed to by set: 1 or 0.

## 4.4. Time Utility Functions

### 4.4.1. atotv

```
timev atotv (timestring)
string           timestring;
```

The function accepts a character string representing a time interval and converts it into a timeval struct which it returns. The timestring is a positive or negative decimal number of seconds, with an optional decimal fraction of seconds, i.e. "[+-]xxx[.yyy]".

The atotv() function is in times.c.

### 4.4.2. Assorted timeval macros

A zero-valued timeval structure called time_zero is declared in times.c. This is a global which can be passed to functions, as can pointers to it. However, it must be treated as a read-only variable -- it must not be changed or many functions will break.

The following are macros which are in dcctime.h and perform some useful operations on timeval structures:

```
int time_nonzero(tv)
timev            tv;
```

Takes a timeval argument, and returns true if it is non-zero, otherwise returns false.

```
int time_iszero(tv)
timev            tv;
```

Takes a timeval argument, and returns true if it is zero, otherwise returns false.

```
int time_equal(tv1,tv2)
timev            tv1;
timev            tv2;
```

Takes two timeval arguments, and returns true if they are equal, otherwise returns false.

```
int time_after(tv1,tv2)
timev            tv1;
timev            tv2;
```

Takes two timeval arguments, and returns true if the first is greater than the second, otherwise returns false.

```
int time_before(tv1,tv2)
timev            tv1;
timev            tv2;
```

Takes two timeval arguments, and returns true if the first is less than the second, otherwise returns false.

```
int time_positive(tv)
timev            tv1;
timev            tv2;
```

Takes a timeval argument, and returns true if it is greater than zero, otherwise returns false.

```
int time_negative(tv)
timev            tv1;
timev            tv2;
```

Takes a timeval argument, and returns true if it is less than zero, otherwise returns false.

```
timeval time_clear(tv)
timev            tv;
```

Takes a timeval argument and clears it to zero.

```
timeval  time_normalize(tv)
timev            tv;
```

Takes a timeval argument and converts it to normal form with the microsecond form corresponding to a positive fraction of a second.

```
timeval time_plus(tv1,tv2)
timev           tv1;
timev           tv2;
```

Takes two timevals and returns their sum.

```
timeval time_minus(tv1,tv2)
timev           tv1;
timev           tv2;
```

Takes two timevals and returns the difference of the first minus the second.

```
timeval time_add(tv1,tv2)
timev           tv1;
timev           tv2;
```

Takes two timevals and adds the second one to the first.

```
timeval time_sub(tv1,tv2)
timev           tv1;
timev           tv2;
```

Takes two timevals and subtract the second one to the first.

```
timeval time_elapsed_add(tv1, tv2, tv3)
timev           tv1;
timev           tv2;
timev           tv3;
```

Takes three timevals and adds the difference of the second and the third to the first.

```
boolean timerisset(timevalp)
timev           *timevalp;
```

Takes a pointer to a timeval and returns true if the time is non-zero.

```
timerclear(timevalp)
timev           *timevalp;
```

Takes a pointer to a timeval and clears it to zero.