Thomas Jefferson National Accelerator Facility
Data Acquisition Group
12000 Jefferson Ave., MS 12H
Newport News, VA 23606

Phone: (757)269-7030
FAX: (757)269-5800

TJNAF
Jefferson Laboratory

EVENT TRANSFER or "ET" SYSTEM

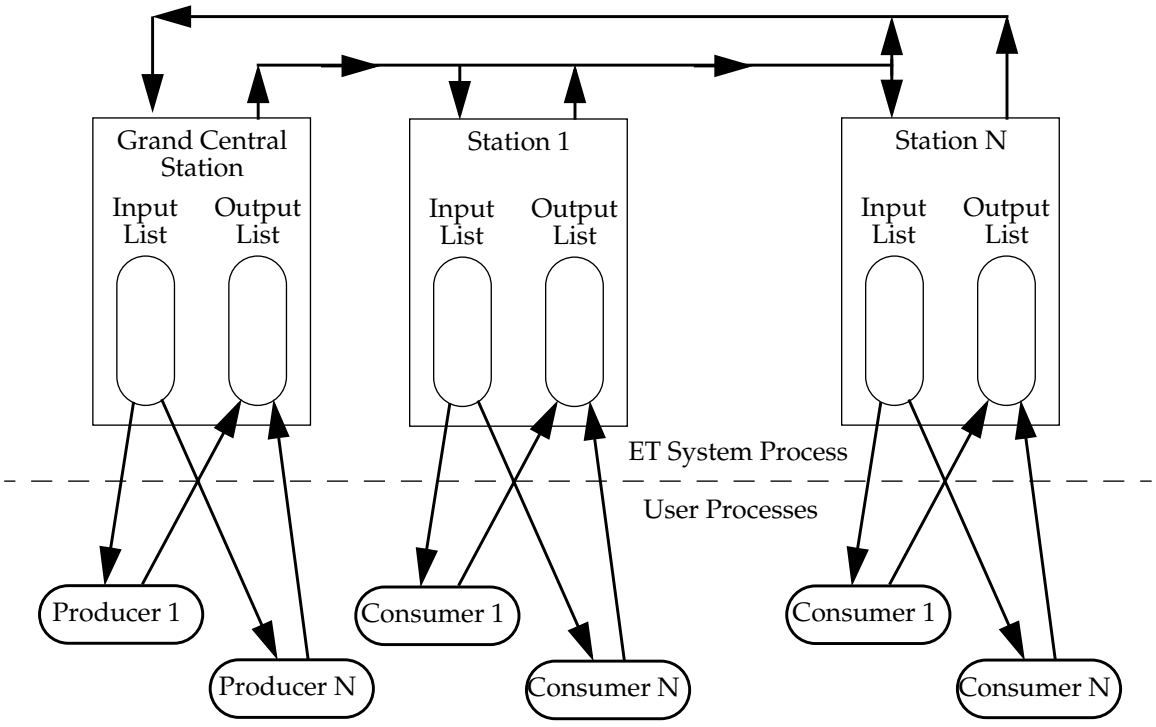
Carl Timmer April 17, 2000
TJNAF/Physics Division/Data Acquisition Group System Routines

Table Of Contents

Chapter 1 Introduction

The following is a diagram of the basic architecture of the ET system.

Figure 1. ET System Architecture



1.1 General Description of the ET System

The main idea behind this “Event Transfer System” software is to create an extremely fast method of transferring “events” from process to process. An event is simply an empty buffer or memory that can be filled with whatever data users wants to share with each other.

In a nutshell, the ET "system" consists of a single process which memory maps a file into its memory space. This file can be used by any "user" process to map that same memory into its own space. Although done transparently to the user, it allows for quick communication between processes and forms the foundation of the event transfer system. The system is responsible for the transfer of all events from user to user, or more accurately, from station to station.

The system consists of " stations" each of which are essentially two lists: 1) an input list of events to be read, and 2) an output list of events that have been read and are ready to be sent to the next station. These stations, in turn, are themselves formed into a linked list. Events pass from statin to station until they reach the last station in the list and are then returned to the first station.

The first station is special and for lack of a better name is called, grandcentral station. It is a repository of unused events which it gives to event " producers" who ask it for one. It is created automatically when starting up an ET system. All other stations are created by users. They are linked together in the order they are created on a first-come-first-serve basis.

User processes can use functions from an ET system library to connect to the mapped memory - also called opening the ET system. Once open, the user can proceed to create stations and then make attachments to those or other stations. Once attached to a particular station, one can read and write events from it. The above steps can also be reversed by detaching from stations, removing stations, and closing the ET system in that order.

In the process of reading or "getting" an event, the user grabs one from a station's input list and similarly, in the process of writing or " putting" it, the user places it into the station's output list. All output lists have enough space to contain all events in the system. Thus a user can put events with speed and impunity since there will always be room.

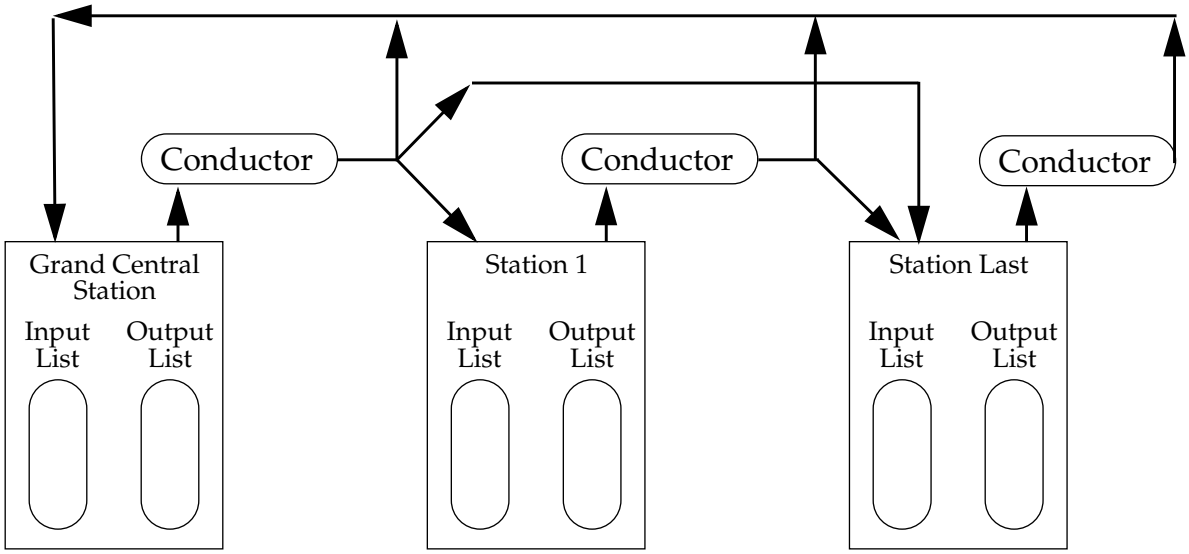
In the following document, processes which write data into event buffers thereby creating data are called producers, while processes which are interested in reading, analyzing, and even modifying data produced by others are called consumers.

1.2 Some Details of the ET System

Take a closer look by examining figure 2. It shows the flow of events with everything occurring completely in the ET system process. One advantage of doing things this way is that crashed user processes

will not affect the flow of events, avoiding bringing the whole system to a grinding halt.

Figure 2. Event Transfer by Conductor Threads.



The way that this is accomplished is that ET is multithreaded. Each station has its own event transfer thread - or conductor - which is waiting for output events. When an event is written, it wakes up the conductor which reads all events in the list, determines which events go where, and writes them in blocks to each station. The conductor also releases the specially allocated memory associated with temporary events (more on temp events later).

The use of threads have made complete error recovery possible 99.9% of the time. The system and user processes each have a thread which sends out a heartbeat (increments integer in shared memory). The system monitors each process and each process monitors the system in yet another thread. If the system dies, user processes automatically return from any function calls that are currently pending and can make a function call to find out if the system is still alive or can wait until it resurrects. Likewise, if a user's heartbeat stops, the system kills the user and erases any trace of it from the system. All events tied up by the dead user process are returned to the system. Users can tell a station to take those events and send them to either: 1) the station's input list, 2) the station's output list, or 3) grandcentral station (essentially dumping them).

Safety features include tracking an event's owner - the process that currently has control over it. Keeping tabs on who has an event prevents the user from writing the same event twice or writing events into the system which it doesn't own and thereby creating serious problems.

Temporary events are called such because occasionally, a user will need an event to hold a large amount of data - larger than the space that was allocated for an event when the ET system was started and the event size was determined. In such cases, a request for a large event will cause a file to be memory mapped with all the requested space. When all users are done with it, this temporary event will be disposed of - freeing up its memory. Incidentally, this is all transparent to the user.

Events can be either high or low priority. High priority events that are placed into the system are always placed at the head of stations' input and output lists. That is, they are placed below other high priority, but above all the low priority items. (If there is a demand for it, the capability to generate events which could be immediately broadcast to all stations could be implemented. If the high/low priority business is useful/useless to the reader, the author would appreciate the feedback.)

The ET system consists of one process and allows no environmental variables to affect its behavior. In addition, there are no global or static variables in the code, making it reentrant. This allows one to use more than one ET system at the same time. Multiple systems peacefully coexist.

Currently ET systems will run on the Solaris and Linux operating systems.

1.3 Event Flow

From start to finish, events flow something like this. An ET system is started up with a unique filename to identify it. Once a system exists, a process can open the system which maps ET memory into its own space. At this point the user can begin to use it.

To do anything interesting, a user must attach to a station that it created or that already exists and receive a unique identifier which it can then use to read and write events. They can be read or written either singly or in blocks (i.e. arrays).

A process can attach to many different stations, and it will receive a unique identifier for each station that it attaches to. (There are some " gotchas'in this area to be discussed later). Processes which wish to be producers can do so by attaching to grandcentral and requesting new events. Alternatively, any attached processes can request new events and write them into their own stations.

After attaching to a station, one can also detach from that station. This is a necessary prerequisite - all attachments must be detached - should one want to remove a station. Grandcentral station (the first and automatically created station) can never be removed.

Should the ET system ever die, this can be detected. It is also possible to wait until the system restarts by calling a single routine.

Chapter 2

Creating an ET system

2.1 System Creation

To create an new ET system one only needs to make a single call to the ET library function, “et_system_start”. All ET systems are completely independent of each other, allowing the creation of as many as are necessary. However, bear in mind that the process creating an ET system must remain in existence. In other words, it does NOT spawn or fork off an independent ET process. (Although it is possible for any user to implement such behavior).

The arguments to “**et_system_start(et_sys_id* id, et_sysconfig sconfig)**” are a pointer to an ET system identification and an ET system configuration.

The behavior of this routine is as follows. If the name of the ET system file (see below) given in the configuration parameter does not exist, an ET system is created with no objections. If, however, such a file does exist, it is first mapped into the process’ memory. The memory is monitored to see if there is an living system heartbeat. If there is, then an ET system is already attached to it and an error is returned. If there isn’t, a check is made to see if the size and shape of the existing ET system is the same as the one in the configuration parameter. If it isn’t an error is returned. If it is, then the memory is carefully initialized taking care not to overwrite important data about the current state of the system and all necessary threads are started.

To close the newly created ET system, use the function **et_system_close(et_sys_id id)**. This may only be called by the process which called “ et_system_start” or an error will be returned.

2.1.1 ET system identification

An ET system id is created by declaring a variable of the type “et_sys_id”. A pointer to this variable is then passed to “ et_system_start”When the user has no more use for the ET system, a call to “et_system_close” will stop all ET-related threads and

unmap the ET memory. The process that called “et_system_start” can continue on its merry way, but all ET clients are left to fend for themselves.

Alternatively, if a user is attaching to an existing ET system instead of trying to create one, a pointer to a system id is one parameter passed to the “et_open” function (more on this later). A call to ” et_close’is then required to cleanly remove all connection to the ET system.

2.1.2 ET system configuration

An ET system configuration is created by declaring a variable of the type “ et_sysconfig”Once this variable is declared, it must be initialized before further use. Thus users must call the function “et_system_config_init”. After initialization, calls can be made to functions which set various properties of the specific configuration. Calls to these setting functions will fail unless the configuration is first initialized.

When the user is finished using a configuration variable, the user must call “et_system_config_destroy” with the configuration as an argument in order to properly release all memory used.

The configuration parameters that the user can set are the total number of events, the maximum size of each event, the maximum number of temporary events, the maximum number of stations, the maximum number of connecting processes, the maximum number of attachments to stations, and the name of the system. For remote users, one can also set how to find the remote ET systems (either by broadcasting or multicasting), what IP address to use, and what port number to use.

The functions used to SET these parameters are listed below along with a short explanation for each:

1. **et_system_config_setevents(et_sysconfig sconfig, int val)** : sets the total number of events.
2. **et_system_config_setsize(et_sysconfig sconfig, int val)** : sets the maximum size in bytes for each events’ data.
3. **et_system_config_settemps(et_sysconfig sconfig, int val)** : sets the maximum number of temporary events. These events are used when an event is required whose data size exceeds the limit set by the previous function. To accommodate large events, memory is specially allocated as needed. This cannot exceed the total number of events in the system.
4. **et_system_config_setstations(et_sysconfig sconfig, int val)** : sets the maximum number of stations.
5. **et_system_config_setprocs(et_sysconfig sconfig, int val)** : sets the maximum number of user processes which may open an ET system.

6. **et_system_config_setattachments(et_sysconfig sconfig, int val)** : sets the maximum number of attachments to stations.
7. **et_system_config_setfile(et_sysconfig sconfig, char *val)** : defines the name of an ET system. Each ET system is defined by a unique file name which is used to implement the memory mapped file basis of the ET system.
8. **et_system_config_setcast(et_sysconfig sconfig, int val)** : for remote users, the mechanism for finding ET systems is set to broadcasting by setting val=ET_BROADCAST, or to multicasting by setting it to ET_MULTICAST.
9. **et_system_config_setaddress(et_sysconfig sconfig, char *val)** : defines the IP address of the broadcast or multicast used for finding ET systems by remote users. The address must be in dotted-decimal form.
10. **et_system_config_setport(et_sysconfig sconfig, int val)** : for remote users, set the broad/multicast port #.

Similarly, functions used to GET these parameters are available and listed in the chapter describing all the ET library routines.

2.2 Example

An example of a program to start an ET system is listed below.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <et.h>

main(int argc, char **argv)
{
 int             status, sig_num;
 sigset_t    sigwaitset;
 et_sysconfig  config;
 et_sys_id     id;

 if (argc != 2) {
  printf("Usage: et_start <name>\n");
  exit(1);
 }

 /*****************************/
 /* set configuration parameters */
 /*****************************/

 if (et_system_config_init(&config) == ET_ERROR) {
  printf("et_start: no more memory\n");
  exit(1);
```

```
}
 /* total number of events */
et_system_config_setevents(config, 1000);
 /* size of event in bytes */
et_system_config_setsize(config, 2000);
 /* max number of temporary (specially allocated mem) events */
 /* This cannot exceed total # of events                    */
et_system_config_settemps(config, 500);
 /* max number of stations */
et_system_config_setstations(config, 10);
  /* max number of attachments */
 /* max allowable = ET_ATTACHMENTS_MAX */
et_system_config_setattachments(config, 10);
/* max number of processes */
 /* max allowable = ET_PROCESSES_MAX */
et_system_config_setprocs(config, 10);
 /* remote users' use broadcast instead of multicast to find me */
et_system_config_setcast(config, ET_BROADCAST);
 /* remote users broadcast to this port */
et_system_config_setport(config, ET_BROADCAST_PORT);
 /* remote users broadcast to this IP address */
et_system_config_setaddress(config, "129.57.35.255");

 /* set ET system filename */
if (et_system_config_setfile(config,argv[1]) == ET_ERROR) {
  printf("et_start: bad filename argument\n");
  exit(1);
};

 /************************/
 /*    start ET system    */
 /************************/
printf("et_start: starting ET system %s\n", argv[1]);
if (et_system_start(&id, config) != ET_OK) {
  printf("et_start: error in starting ET system\n");
  exit(1);
}

 /* set level of debug output */
et_system_setdebug(id, ET_DEBUG_INFO);

 /* turn this thread into a signal handler */
sigemptyset(&sigwaitset);
sigaddset(&sigwaitset, SIGINT);
sigwait(&sigwaitset, &sig_num);
printf("I got CONTROL-C\n");

exit(0);
}
```

Chapter 3

Using An ET System

In the previous chapter, we learned how to create an ET system, and in this chapter we'll learn to use an existing system. This chapter shows how users can attach to ET systems, define, create and remove stations, attach to and detach from stations, handle events, and handle signals.

3.1 Opening an ET System

Opening a system is done by calling "**et_open (et_sys_id* id, char *filename,et_openconfig config)**". The user defines a variable of type "et_sys_id" and passes its pointer - a value-result argument - which then gives back an "ID" to the open ET system. In addition, the filename of an existing ET system and a parameter describing how the user would like to open the system are passed as parameters to " et_open" .

There are a number of functions used to create and define the " config"argument. It is initialized by a call to "**et_open_config_init (et_openconfig *config)**". When the user is finished using the configuration, "**et_station_config_destroy (et_statconfig config)**" must be called in order to properly release all memory used.

After initialization, calls can be made to functions which set various properties of the specific configuration. Calls to these setting functions will fail unless the configuration is first initialized. The functions used to SET these properties are listed below along with an explanation for each:

1. **et_open_config_setwait(et_openconfig config, int val)** : setting " val"to ET_OPEN_WAIT makes " et_open"block by waiting until the given ET system is fully functioning or a set time period has passed before returning. Setting val to ET_OPEN_NOWAIT makes " et_open"return immediately after determining whether the ET system is alive or not. If the system is remote, then broadcasting to find its location may take up to several seconds. The default is ET_OPEN_NOWAIT.

2. **et_open_config_settimeout(et_openconfig config, struct timespec val)** : in ET_OPEN_WAIT mode, this function sets the maximum amount of time to wait for an alive ET system to appear. If the time is set to zero (the default), an infinite time is indicated. If broad/multicasting to find a remote ET system, it is possible to take up to several seconds to determine whether the system is alive or not -- possibly exceeding the time limit.

3. **et_open_config_sethost(et_openconfig config, char *val)** : this sets the name of the host (or computer) on which the ET system resides. For opening a local system only, set val to ET_HOST_LOCAL (the default) or "localhost" (including quotes). For opening a system on an unknown remote computer only, set it to ET_HOST_REMOTE. For an unknown host which may be local or remote, set it to ET_HOST_ANYWHERE. Otherwise set val to the name or dotted-decimal IP address of the desired host. (See next routine also).

4. **et_open_config_setcast(et_openconfig config, int val)** : setting val to ET_BROADCAST (default) means using UDP broadcast IP packets to determine the location of remote ET systems so they can be opened. Setting val to ET_MULTICAST uses the newer UDP multicast IP packets to do the same. However, setting val to ET_DIRECT makes a direct connection to the ET system and requires that "et_open_config_sethost" use the actual host's name or "localhost" and not ET_HOST_LOCAL, ET_HOST_REMOTE, or ET_HOST_ANYWHERE. The tcp port number used in the direct connection is set by "et_open_config_serserverport" and defaults to ET_SERVER_PORT, defined in " et.h" as 11111.

5. **et_open_config_setTTL(et_openconfig sconfig, int val)** : when using multicasting, set the TTL value. This sets the number of routers to hop. The default is one which keeps things on the subnet.

6. **et_open_config_setport(et_openconfig config, unsigned short val)** : this sets the port number of the broadcast or multicast communications. The default is ET_BROADCAST_PORT, defined in " et.h" as 11111.

7. **et_open_config_setserverport(et_openconfig config, unsigned short val)** : this sets the port number of the tcp server thread of an ET system. The default is ET_SERVER_PORT, defined in " et.h" as 11111.

8. **et_open_config_setaddress(et_openconfig config, char *val)** : this sets the IP address of the broadcast or multicast communications in dotted-decimal form. It is automatically set to the local subnet broadcast address and if that fails it defaults to ET_BROADCAST_ADDR (which is defined to be the author's local subnet in "et.h"). If multicasting is used, the address must be explicitly set by this routine.

9. **et_open_config_setmode(et_openconfig config, int val)** : setting val to ET_HOST_AS_LOCAL (default) means users which are on the same machine as the ET system (local) will realize this and take advantage of it. However, setting val to ET_HOST_AS_REMOTE means users will be treated as if they were remote even if they are local. All transactions will be through the ET system's server and not through shared memory.
10. **et_open_config_setdebugdefault(et_openconfig config, int *val)** : this sets default level of debug output. Set val to: ET_DEBUG_NONE for no output, ET_DEBUG_SEVERE for output describing severe errors, ET_DEBUG_ERROR for output describing all errors, ET_DEBUG_WARN for output describing warnings and errors, and ET_DEBUG_INFO for output describing all information, warnings, and errors.

More on remote ET systems can be found in the chapter entitled Remote ET on page 41. All of the above "set" functions have their counterpart " get" functions as well.

Once an ET system has been opened, users can use the id as a handle for that particular system. Users can open more than one system at a time, referring to each by their respective handles.

3.2 Definition of Stations

3.2.1 Definition

Analogous to the opening or creation of ET systems, users begin by declaring a variable of type "et_statconfig". Once this variable is declared, it must be initialized before further use. Thus users must also call the function "**et_station_config_init(et_statconfig* sconfig)**" After initialization, calls can be made to functions which set various properties of the specific configuration. Calls to these setting functions will fail unless the configuration is first initialized.

When the user is finished using a configuration variable, the user must call "**et_station_config_destroy(et_statconfig sconfig)**" with the configuration as an argument in order to properly release all memory used.

The functions used to SET station parameters are listed below along with an explanation for each:

1. **et_station_config_setblock(et_statconfig sconfig, int val)** : setting "val" to ET_STATION_BLOCKING makes the station block by looking at all events in the system, while setting it to ET_STATION_NONBLOCKING allows the station to fill up a cue of events and when that is full, events flow to the next station downstream. The default is blocking.

2. **et_station_config_setcue(et_statconfig sconfig, int val)** : when in nonblocking mode, this sets the maximum number of events that are to be in the station's input list ready for reading (in so far as it is possible). The default is 10.
3. **et_station_config_setprescale(et_statconfig sconfig, int val)** : when in blocking mode, every Nth event of interest is sent to the user by setting the " val" to N. The default is 1.
4. **et_station_config_setuser(et_statconfig sconfig, int val)** : setting "val" to ET_STATION_USER_SINGLE allows only one user process to attach to this station, while setting it to ET_STATION_USER_MULTI allows multiple users to attach. The default is multiuser.
5. **et_station_config_setrestore(et_statconfig sconfig, int val)** : when a process dies or detaches from a station, the events it read but did not write are recovered and sent to a station's output list if " val"is set to ET_STATION_RESTORE_OUT. Similarly, it can be sent to the input list with ET_STATION_RESTORE_IN or back to grandcentral station with ET_STATION_RESTORE_GC. The default is restoration to the output list.
6. **et_station_config_setselect(et_statconfig sconfig, int val)** : for selection of all events and no filtering set "val" to ET_STATION_SELECT_ALL. For selection using a user-defined routine loaded through a shared library set it to ET_STATION_SELECT_USER. For mimicking the DD system, set it to ET_STATION_SELECT_MATCH. The last option takes an event's array of control integers and does a comparison with the station's selection integers or words. The results of all logical comparisons are ORed together. An event is selected if result = 1. See below for more details. The default mode is ET_STATION_SELECT_ALL.
7. **et_station_config_setselectwords(et_statconfig sconfig, int *val)** : the argument is an array of integers used when the station select mode is set to ET_STATION_SELECT_MATCH or possibly ET_STATION_SELECT_USER (depending on what algorithm a user-defined, event selection routine uses). For DD users, it is identical to the old " p2ctl"element of the old " fmode"structure. The default is to set all integers to a value of " -1" .
8. **et_station_config_setlib(et_statconfig sconfig, char *val)** : for a select mode of ET_STATION_SELECT_USER, " val"is the name of the shared library containing the function to be used for selecting events.
9. **et_station_config_setfunction(et_statconfig sconfig, char *val)** : for a select mode of ET_STATION_SELECT_USER, " val"is the name of the function to be used for selecting events.

Just a few notes on some of the details. When selecting the "ET_STATION_RESTORE_IN" mode for event restoration, be aware of a few things. If there is only one process attached to such a

station and it dies, the events go to the output list in order to prevent them from being lost to a station with no event readers. If there is more than one process attached and one dies, its events will be put into the input list with the assumption that the recovered events are higher in priority to those already in the station's input list. To be exact, the recovered high priority events are placed " above" (sooner to be read) all other events, and the recovered low priority events are placed below high priority but above all other low priority events.

The mode denoted by ET_STATION_SELECT_MATCH has the following behavior. A check is made to see if the first element of the station's selection array is equal to -1. If it is, then the first element of the event's control array is ignored and the event is not marked for selection. Similar comparisons continue for each element of the arrays. Thus, if all elements of a station's selection array are set to -1, the event will NOT be selected. If the first element of the station's selection array is not -1 but is equal to the first element of the event's control array, then the event is selected. If the bitwise AND (&) of the station's and event's second elements is true, then the event is selected. This pattern is repeated with the even elements 0,2,4, 6, ... compared for equality and the odd elements 1, 3, 5, ... compared for bitwise AND. If any of the comparisons are true, then the event is selected. This is the logic employed by the old DD system in its " conditional" mode.

Similar function to those mentioned above are available to GET the values associated with a station configuration.

3.2.2 Examples

Since one of the more difficult tasks facing the first time user is how to properly configure a station, let's look at two examples first:

```
/* declarations */
et_stat_config   sconfig;
/* set values */
et_station_config_init(&sconfig);
et_station_config_setselect(sconfig, ET_STATION_SELECT_ALL);
et_station_config_setblock(sconfig, ET_STATION_NONBLOCKING);
et_station_config_setuser(sconfig, ET_STATION_USER_SINGLE);
et_station_config_setrestore(sconfig, ET_STATION_RESTORE_GC);
et_station_config_setcue(sconfig, 20);
```

Here is a station to which only 1 user may attach. It accepts all events no matter what values the selection integers have. It is nonblocking, meaning that once the system fills up its input list with a maximum of 20 events, all other events will bypass the station and be placed somewhere downstream. If the user process should die, the events that it owns will be placed back in grandcentral station, and no one else will get them.

A more complicated example can be seen below:

```
/* declarations */
int selections[] = {17,22,-1,-1};
```

```
et_stat_config  sconfig;
/* set values */
et_station_config_init(&sconfig);
et_station_config_setselect(sconfig, ET_STATION_SELECT_ALL);
et_station_config_setblock(sconfig, ET_STATION_BLOCKING);
et_station_config_setuser(sconfig, ET_STATION_USER_MULTI);
et_station_config_setrestore(sconfig, ET_STATION_RESTORE_IN);
et_station_config_setprescale(sconfig, 5);
et_station_config_setselect(sconfig, ET_STATION_SELECT_USER);
et_station_config_setselectwords(sconfig, selections);
if (et_station_config_setlib(sconfig, "/stuff/libet_user.so") == ET_ERROR) {
   printf(" cannot set library\n");
}
if (et_station_config_setfunction(sconfig, "et_my_function") == ET_ERROR) {
   printf("cannot set function\n");
}
```

In the above example, there is a station to which multiple users can attach. Its select mode (ET_STATION_SELECT_USER) says that the user will be supplying a function in a shared library to determine which events are to be selected. Since this station is set to block events, all events which meet its selection criteria are placed in its input list, even if it means slowing the whole ET system down to a crawl. Actually, the prescale factor imposes an additional selection criterion since it is in blocking mode. Thus, only every 5^{th} event which passes through the user's filter gets placed in the station's input list. Its restore mode says that if this user process should ever die, the events that it currently owns will be placed in the station's input list.

3.3 Creation & Removal of Stations

Once a configuration is defined, it is passed to the function "**et_station_create(et_sys_id id, et_stat_id *stat_id, char *stat_name, et_statconfig sconfig)**" In addition to the arguments, id and sconfig, which have already been covered, the user must supply a unique name and is returned a station identification number " stat_id" . This station id is used in other station-related routines.

Possible errors returned by the function "et_station_create" are ET_ERROR_EXISTS if a station by that name exists already, ET_ERROR_TOOMANY if the user is the second user to try to attach to a station designated for one user only, or ET_ERROR for other unrecoverable errors. If the user is a remote client, the error ET_ERROR_REMOTE indicates a bad arg or not being able to allocate memory, and ET_ERROR_READ & ET_ERROR_WRITE indicate problems with the network communication.

Removing stations can be accomplished by calling "**et_station_remove(et_sys_id id, et_stat_id stat_id)**" .

3.4 Attaching to and Detaching from Stations

Until a user's process attaches to a station, the station is placed in an idle mode, meaning, that it is not participating in the flow of events - it is getting by-passed. Once a process attaches to a station, it becomes active and begins to receive events. This logic ensures that events do not get stuck in stations with no one to process them or that the entire flow of events does not come to a grinding halt.

Attach to a station by calling "**et_station_attach(et_sys_id id, et_stat_id stat_id, et_att_id *att)**". This routine returns a unique attachment number, "att", by which a process identifies itself in certain function calls. For example, when reading and writing events, this parameter is required. In this manner, a single process can attach to different stations and yet be differentiated by the ET system. With this type of interface, for example, a user could conceivably have multiple threads with each attached to the same station on a different attachment. The idea is that this id represents a single attachment to a single station.

To detach from a station call "**et_station_detach(et_sys_id id, et_att_id att)**" If a user is the last one to detach from a station, all of the events left in the station's input list are passed to the output list. In addition, after a user detaches, a search is made for any events that were read but not written back into the ET system by " att"They are recovered and placed according to the station's property set by the function " et_station_config_setrestore" .

3.5 Handling Events

After opening an ET system, creating a station, and attaching to it, users are ready to start creating, reading and writing events.

3.5.1 Creating

When creating an new event, users are called producers. There are two routines that can be used for doing this. The first is for getting a single, blank event by calling "**et_event_new(et_sys_id id, et_att_id att, et_event **pe, int wait, struct timespec *time, int size)**" At this point users are familiar with the first two arguments " id"and " att" . The third is a pointer to a pointer to an event. In the code, declare a pointer to an event (i.e. et_event *pe) and pass its address. Upon a successful return, " pe"points to a new event. The fourth arg, " wait" , is a flag that can be set by using some predefined macros. By setting this " wait"to ET_SLEEP, the call will block until the next free event is available. By setting it to ET_ASYNC, the call returns immediately with a status. And by setting it to ET_TIMED, the call waits for the amount of time given by the " time"arg if no events are immediately available. Finally, the last arg is the requested size in bytes. If the size is larger than those the system was created with, the newly created event will be declared a special " temporary"event and will allocate the necessary memory. (This, of course, slows things down).

Similarly the user can call "**et_events_new(et_sys_id id, et_att_id att, et_event *pe[], int wait, struct timespec *time, int size, int num, int *nread)**" for obtaining an array of new events. In this case, " pe" is an array of pointers to events, "num" in the number of events desired, and "nread" is the number of events actually read and placed into the array (which may be less than what was asked for).

3.5.2 Reading

When reading events, users are called consumers. There are two routines that can be used for reading. The first is for reading single events and has the form "**et_event_get(et_sys_id id, et_att_id att, et_event **pe, int wait, struct timespec *time)**" The arguments are the same as those for creating a new event but without the size.

The second type of routine is for reading an array of events by using the call, "**et_events_get(et_sys_id id, et_att_id att, et_event **pe, int wait, struct timespec *time, int num, int *nread)**". The arguments are almost the same as for reading single events with the exception that the user passes an array of pointers to events. There are also additional arguments specifying the number of events the user wants to read and the number actually read. Although less events may be returned, the user will never get more than the amount asked for.

3.5.3 Writing

After reading an event, the user has access to a number of its properties for manipulation. Routines to accomplish that are given in the following list:

1. **et_event_setpriority(et_event *pe, int pri)** : this routine sets the priority of an event, " pri"to be ET_HIGH or ET_LOW (default). A high priority means that such an event gets placed below other high priority but above low priority events when placed in a station's input or output list. Thus, high priority events are always the first to be read. No other guarantees are made.
2. **et_event_getpriority(et_event *pe, int *pri)** : this routine returns the priority of an event.
3. **et_event_setlength(et_event *pe, int len)** : sets the length or size of the event's data in bytes.
4. **et_event_getlength(et_event *pe, int *len)** : returns the length of the event's data in bytes.
5. **et_event_setcontrol(et_event *pe, int con[], int num)** : sets the control information of an event. The " con"argument is an array of integers which control the flow of the event through the ET system, and the " num"argument gives the size of the array. The DD system had ctlw1, ctlb1, ctlw2, and ctlb2 as four integers used to carry this information. These

integers are now replaced respectively by an array of integers. The size of this array is determined at compile time by " ET_STATION_SELECT_INTS" which defaults to four.

6. **et_event_getcontrol(et_event *pe, int con[])** : gets the event's array of control information.
7. **et_event_getdata(et_event *pe, void **data)** : this routine returns a void pointer to the start of an event's data location.
8. **et_event_getdatastatus(et_event *pe, int *status)** : this routine gets the status of an event's data. It can be either ET_DATA_OK, ET_DATA_POSSIBLY_CORRUPT, or ET_DATA_CORRUPT (not currently used). Data is ET_DATA_OK unless a previous user got the event from the system and then exited or crashed without putting it back. If the ET system recovers that event and puts in back into the system, its status becomes ET_DATA_POSSIBLY_CORRUPT as a warning to others.
9. **et_event_setendian(et_event *pe, int endian)** : though normally the ET system automatically keeps track of the endianness of an event's data, this routine can override and directly set the endian value of the data. It may be ET_ENDIAN_BIG, ET_ENDIAN_LITTLE, ET_ENDIAN_LOCAL (same endian as local host), ET_ENDIAN_NOTLOCAL (opposite endian as local host), or ET_ENDIAN_SWITCH. See the chapter Remote ET.
10. **et_event_getendian(et_event *pe, int *endian)** : this routine returns the endian of an event's data - either ET_ENDIAN_BIG or ET_ENDIAN_LITTLE. See the chapter Remote ET.
11. **et_event_needtoswap(et_event *pe, int *swap)** : this routine tells the caller if an event's data needs to be swapped or not by returning either ET_SWAP or ET_NOSWAP. See the chapter Remote ET.
12. **et_event_CODAswap(et_event *pe)** : this routine swaps the data of an event in CODA format.

After setting an event's priority, data length, control array and perhaps its endian value, and writing data, the user is finished with the event and wishes to place it into the ET system. Or perhaps the user has only read the data and is done with the event. In any case, the event must be written back into the system by two possible means. Either write a single event with "**et_event_put(et_sys_id id, et_att_id att, et_event *pe)**" or write multiple events with "**et_events_put(et_sys_id id, et_att_id att, et_event *pe[], int num)**" In the latter case, the user gives the number " num"of events to put back in the array " pe"All events will always be successfully written and will never block as a station's output list has enough room for all events in the whole ET system.

The ET system checks to see if the "att" that read the event is the same one that is writing it. If it isn't, the call returns an error and nothing is written.

3.5.4 Dumping

After reading existing events or creating new ones, it's possible that these events may no longer be of interest to the user of any other user on the system. In that case, one may dump or recycle these events by calls to two routines. They are identical to the routines *et_event(s)_put* in their arguments. The first is "**et_event_dump(et_sys_id id, et_att_id att, et_event *pe)**" and dumps a single evernt. Similarly, "**et_events_dump(et_sys_id id, et_att_id att, et_event *pe[], int num)**" dumps multiple events.

3.6 Closing an ET System

When finished using an ET system, it can be removed from a process' memory by using the "**et_close(et_sys_id id)**" routine. This unmaps the ET system memory from the process and makes it inaccessible. It also stops the heartbeat and system-heartbeat-monitor threads. In order to close, all attachments must be detached first. However, there is another function "**et_forcedclose(et_sys_id id)**" which will automatically do all the detaching first. Of course, the ET system continues to function for other processes as before.

Chapter 4

ET Programming Details

This chapter gives some details on programming with an ET system. It answers questions about program flow, handling signals, useful ET library functions, how to define user functions for selecting events, and various odds & ends.

4.1 Program Flow

Being such a complicated, multithreaded, multiprocess system, it is probably not at all obvious how a user would put all the calls to the ET library together in a coherent manner. Given below is a bare bones outline of how a user's process should look.

```
/* declare variables */
int             status;
et_statconfig   sconfig;
et_openconfig   openconfig;
et_event        *pe;
et_sys_id       id;
et_stat_id      my_stat
et_att_id       attach;

/* define station */
et_station_config_init(&sconfig);
et_station_config_setblock(sconfig, ET_STATION_BLOCKING);
et_station_config_setselect(sconfig, ET_STATION_SELECT_ALL);
et_station_config_setuser(sconfig, ET_STATION_USER_SINGLE);
et_station_config_setrestore(sconfig, ET_STATION_RESTORE_OUT);

/* open ET system */
et_open_config_init(&openconfig);
et_open(&id,"  /tmp/my_et_system_file" , openconfig);
et_open_config_destroy(openconfig);

/* create and attach to station */
et_station_create(id, &my_stat, "  my_station" , sconfig);
et_station_attach(id, my_stat, &attach);

while (1) {
```

```
    while(et_alive(id)) {
      status = et_event_get(id, attach, &pe, ET_SLEEP, NULL);
      status = et_event_put(id, attach, pe);
    }
    et_wait_for_alive(id);
}
```

Besides defining a station, the first thing to do is to initialize with "et_open". This maps the given file into the user's memory giving access to the ET system. It also starts a heartbeat and begins to listen for the ET system's heartbeat. Even if the ET system should die and resurrect, this need not be repeated. However, after an"et_close" it will have to be repeated to regain access to the ET system.

Create any desired stations, then attach to one of them. By attaching, one receives a unique identifier ("attach" in this case). This will be used in the rest of the transactions.

Once finished attaching, one can read and write events, checking every now and then to see if the ET system is alive. If the ET system dies while the user is in a read waiting for events, the read call will return with the error ET_ERROR_DEAD. Although not shown in this code, be sure to carefully check the status of each read and write statement.

Popping out of the read/write while loop, one reaches the call to " et_wait_for_alive"which simply waits for a living ET system (one with a heartbeat). If a dead system resurrects, the flow should go back to reading and writing events. Remember that when an ET system is restarted, it looks at the existing shared memory and is able to pick up where it left off (at least that is how it is designed). All events at the time of the crash will be lost but stations and attachments should remain intact.

4.2 Handling Signals

Because the ET software uses multiple POSIX threads, signal handling must be done carefully. Be sure to use POSIX routines and only those that are thread safe. Refer to the book Programming with POSIX Threads by David Butenhof for a good reference on this subject.

Functions that meet this standard are "pthread_sigmask", " pthread_kill"", sigwait"", sigwaitinfo"and" sigtimedwait"When masking signals use the function " pthread_sigmask"NOT " sigprocmask" since its behavior in a threaded process is undefined.

The best way to handle things is to initially block or mask **all** signals. Once the user has called "et_open", the new threads that were started as a result of calling it will also have all signals blocked because the new threads inherit the signal mask of its parent thread. Once open, handle the signal catching in the main thread or some additional thread spawned from the main thread (i.e. the user's code). See the examples in the example chapter.

4.3 Defining Functions for Event Selection

Should the user wish to provide an event selection capability for a station not already present in the ET system, this may be accommodated by defining a function especially for that purpose. The function must be part of a shared library and must have the arguments:

et_my_function (et_sys_id id, et_stat_id stat_id, et_event *pe) .

This function will be called whenever its associated station is collecting events to gather into its input list. The return value must be one for a selected event and zero otherwise.

The function-writer has access to the event's data through functions mentioned in the previous chapter, Similarly, there is access to information about the station's configuration through the following ET library functions:

1. **et_station_getattachments(et_sys_id id, et_stat_id stat_id, int *numatts)** : gets the number of attachments to a station.
2. **et_station_getstatus(et_sys_id id, et_stat_id stat_id, int *status)** : gets a station's status.
3. **et_station_getblock(et_sys_id id, et_stat_id stat_id, int *block)** : gets a station's blocking mode
4. **et_station_getrestore(et_sys_id id, et_stat_id stat_id, int *restore)** : gets a station's restore mode
5. **et_station_getuser(et_sys_id id, et_stat_id stat_id, int *user)** : gets a station's user mode
6. **et_station_getprescale(et_sys_id id, et_stat_id stat_id, int *prescale)** : gets a station's prescale value
7. **et_station_getcue(et_sys_id id, et_stat_id stat_id, int *cue)** : gets a station's cue value
8. **et_station_getselect(et_sys_id id, et_stat_id stat_id, int *select)** : gets a station's select mode
9. **et_station_getselectwords(et_sys_id id, et_stat_id stat_id, int *select)** : gets a station's selection integer array
10. **et_station_getlib(et_sys_id id, et_stat_id stat_id, char *lib)** : gets a station's shared library name
11. **et_station_getfunction(et_sys_id id, et_stat_id stat_id, char *function)** : gets a station's function name
12. **et_station_getinputcount(et_sys_id id, et_stat_id stat_id, int *cnt)** : gets the number of events in a station's input list. This function may not be so useful in that this data can change so quickly.
13. **et_station_getoutputcount(et_sys_id id, et_stat_id stat_id, int *cnt)** : gets the number of events in a station's output list. This function may not be so useful in that this data can change so quickly.

Using these functions, all relevant information about the ET system necessary to select events for a particular station can be obtained.

4.4 Useful ET Library Functions

There are a number of other routines available to the ET system users. Use the following to get information about stations:

1. **et_station_name_to_id(et_sys_id id, et_stat_id *stat_id, char *name)** : returns a station id given a station's name.
2. **et_station_isattached(et_sys_id id, et_stat_id stat_id, et_att_id att)** : tells if " att" is attached to a station.
3. **et_station_exists(et_sys_id id, et_stat_id *stat_id, char *stat_name)** : tells if a station exists and returns its id.

There are routines available to get information about an ET system:

1. **et_system_getnumevents(et_sys_id id, int *numevents)** : tells how many events a system has.
2. **et_system_geteventsize(et_sys_id id, int *eventsize)** : tells the size in bytes of a system's events.
3. **et_system_getlocality(et_sys_id id, int *locality** : tells whether the ET system is on a remote node or is local.
4. **et_system_getpid(et_sys_id id, pid_t *pid)** : gives the unix process id or pid or the ET system process.
5. **et_system_getheartbeat(et_sys_id id, int *heartbeat)** : tells the heartbeat count.
6. **et_system_getattsmax(et_sys_id id, int *attsmax)** : tells the max number of attachments allowed.
7. **et_system_getstationsmax(et_sys_id id, int *stationsmax)** : tells the max number of stations allowed.
8. **et_system_gettempsmax(et_sys_id id, int *tempsmax)** : tells the max number of temporary events.
9. **et_system_getprocsmax(et_sys_id id, int *procsmax)** : tells the max number of processes allowed to open the ET system locally.
10. **et_system_getattachments(et_sys_id id, int *atts)** : tells the current number of attachments.
11. **et_system_getstations(et_sys_id id, int *stations)** : tells the current number of stations.
12. **et_system_gettemps(et_sys_id id, int *temps)** : tells the current number of temporary events.
13. **et_system_getprocs(et_sys_id id, int *procs)** : tells the current number of processes with the ET system open locally.
14. **et_system_gethost(et_sys_id id, char *host)** : tells which host computer the ET system is running on.

15. **et_system_getserverport(et_sys_id id, unsigned short *port)** : tells the port number of the ET system's TCP server thread.

Two routines affecting user processes are:

1. **et_wakeup_attachment(et_sys_id id, et_att_id att)** : this routine wakes up a particular attachment which is currently blocked on an event read call on a particular station.
2. **et_wakeup_all(et_sys_id id, et_stat_id stat_id)** : this routine wakes up all attachments which are currently blocked on an event read call on a particular station.

Then there are:

1. **et_alive(et_sys_id id)** : this return 1 if the ET system is alive and 0 if it is not.
2. **et_wait_for_alive(et_sys_id id)** : this waits indefinitely until the ET system is alive and then it returns.

4.5 How to Avoid Blocking Forever

Be careful when attaching to more than more station at a time. Multiple attachments and blocking stations are a bad combination. If one is reading and writing from a blocking station, there is the potential to lock up the whole ET system.

The problem arises when the read and write statements of a program are done serially in a single logical loop. Without going into the details, in some circumstances, events all pile up in the input list of one station while the user is waiting to read events from another station. Check your logic carefully.

Similar problems can arise when producing events at an attachment that is also being used for reading or consuming events. The difficulty is that if the user blocks when calling et_event_new, all the events may have previously piled up in the user's station's input list. In this situation the call to et_event_new will never return.

4.6 Includes , Flags, and Libraries

Using the ET system library functions requires users to include the file " et.h" in any programs, as in the following:

#include <et.h>

The name of the ET shared library is **libet.so**, and the name of the static library is **libet.a** .

On Solaris, users will also need to link against three additional libraries, **-lpthread -lposix4 -lthread**, as well as the math and socket libraries and use the multithreading compiling flag " **-mt**" .

On Linux, users must link against **-lpthread** and the math and socket libraries. Since pthread mutexes cannot be shared by multiple processes, the compile flag " -DMUTEX_NOSHARE"must be used. The effect of this flag is to treat local clients on Linux as if they were remote - meaning they access the shared memory through the ET server and sockets. However, because they are local, instead of sending all event data through sockets, only pointers to the shared memory are sent. The clients can map the shared memory and so obtain access to the data directly. Of course, all of this is transparent to the user.

At this time the ET system has been compiled on Solaris 2.5/2.6 with no problems. It has also been compiled on Redhat Linux 6.0 and earlier versions. However, be aware that there are bugs in the pthread library in Redhat 5.1 and earlier that prevent an ET system from functioning properly (Redhat 5.2 was never used so its library may or may not be buggy).

On both Solaris and Linux, pthread mutexes have the default behavior such that if a mutex is locked by some thread, any other thread may unlock it. This is non-portable behavior and must not be relied on according to the man pages. However, it's use is very convenient when recovering from a crashed process which has locked one or more mutexes. The alternative method to recover from such situations is to re-initialize the locked mutexes. Such behavior can be implemented at compile time by specifying the flag "-DMUTEX_INIT" .

4.7 Debug Output

To help in finding problems and finding out information about an active ET system, users can adjust the debug output printed by the system. The two routines used for this purpose are:

1. **et_system_setdebug(et_sys_id id, int debug)** : sets the level of debug output desired.
2. **et_system_getdebug(et_sys_id id, int *debug)** : gets a system's current debug level.

The possible values of the argument " debug" are:

1. **ET_DEBUG_NONE** - this value results in no output
2. **ET_DEBUG_SEVERE** - this value outputs only the most severe errors
3. **ET_DEBUG_ERROR** - this value outputs all errors
4. **ET_DEBUG_WARN** - this value outputs all errors and all warnings
5. **ET_DEBUG_INFO** - this value outputs everything including informational output

The debug level of an ET system or client defaults to ET_DEBUG_ERROR. Notice that the debug level of a system can only be set after the call to “et_open” or “et_system_start.” This means that in order to get output other than errors from these two routines, the source code must be changed and recompiled.

Normally, by default, debug output is simply printed by means of “printf” statements. If the user wishes to use the coda routine “ daLogMsg”to output debug messages, simply recompile ET with the flag “-DWITH_DALOGMSG”. Be sure to link with the library libcmlog.so when doing so.

4.8 TCL/TK Interface

A Tcl/Tk interface is provided in the file et_wish.c . It can be compiled with the command “make et_wish”. Access to an ET system is provided with the “et_connect <et filename>” command which returns a handle used to get information about the system. Similarly, the connection to an ET system can be closed with the “ et_disconnect <et_id>” command.

There are five commands used to get information about a system:

1. **et_sys_config_info <et_id>** which gets static system configuration information which should not change
2. **et_sys_dynamic_info <et_id>** which gets dynamic system information which will be constantly changing
3. **et_stat_config_info <et_id>** which get static station information that should not change
4. **et_stat_dynamic_info <et_id>** which gets dynamic station information which will be constantly changing
5. **et_proc_info <et_id>** which gets all information about processes
6. **et_att_info <et_id>** which gets all information about attachments

The format of data returned from each of these tcl/tk commands is listed respectively below:

1. {system_pid, max#_events, event_size, max#_tempevents, max#_stations, max#_processes, max#_attachments, et_filename}
2. {heartbeat, #_tempevents, #_stations, #_processes, #_attachments}
3. {{station_name, user_mode, restore_mode, block_mode, prescale, cue, select_mode, {list of selectwords}, function_name, sharedlib_name}, {next station}, ... }
4. {{station_name, status, #_attachments, {list of attachment ids}, {input_list_count, events_try, events_in, events_out}, {output_list_count, events_try, events_in, events_out}}, {next station}, ... }

5. {{process_id, unix_pid, heartbeat, #_attachments, {list of attachment ids }}, {next process}, ... }
6. {{attachment_id, process_id, station_id, blocked, events_put, events_get, events_make}, {next attachment}, ... }

Just a couple of notes. The return for item 4 include "events_try" which - for blocking stations - is simply an a count of all the events which match the station's criteria. Not all of these are accepted due to the prescale factor. Also, the " events_out"of the input list and the "events_in" of the output list are not monitored and will return 0. Finally, the "blocked" entry of item 6 is one if the attachment is blocked in a read statement and zero otherwise.

4.9 Monitoring an ET System

There is a program provided to monitor an ET system. It simply maps the ET system into its memory if it's local or gets data over the network if remote and prints out the values that it reads there. If the reader does run into trouble, this program can help isolate any problems. The usage is:

et_monitor [-f <et name>] [-p <update period (sec)>]

[-h <host>] [-r]

If an ET system name is not given it defaults to /tmp/et_sys_<session> where session is the value of the environmental variable SESSION. It defaults to the local host with a period of 5 seconds between updates. If the user wants the monitor to communicate with the ET system as if remote even if it's local, use the -r option. The value of <host> can be provided in various formats. It can be an IP address in dotted-decimal form, the name of the host with or without the domain, ".local" or "localhost" which means look locally only, ".remote" which means look remotely only, or " .anywhere"which means any local or remote node which responds.

Chapter 5

Examples

5.1 Event Producer

An example of a program written to produce events for an ET system is adapted from the file *et_producer1.c*. It follows below:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
#include <limits.h>
#include <et.h>

#define NUMLOOPS 20000
#define CHUNK 10

main(int argc,char **argv)
{
  int           i, j, size, status;
  int           freq, freq_tot=0, freq_avg, iterations=1, count;
  et_att_id     attach1;
  et_sys_id     id;
  et_openconfig openconfig;
  et_event      *pe[CHUNK];
  struct timespec  t1, t2;
  double        time;

  /* handy data for testing */
int     numbers[] = {0,1,2,3,4,5,6,7,8,9};
char *stuff[] = {" One" ," Two" ," Three" ," Four" ," Five" ," Six" ," Seven" ," Eight" ," Nine" ," Ten
int     control[] = {17,8,-1,-1};  /* 17 & 8 are arbitrary */

  /* pass the ET filename and event size on command line */
  if ((argc != 2) && (argc != 3)) {
    printf("  Usage: et_producer <et_filename> [<eventsize>]\n"  );
    exit(1);
  }
  size = 10;
```

```
if (argc == 3) {
  size = atoi(argv[2]);
}

/* open local ET system and don't wait for it */
 et_open_config_init(&openconfig);
 if (et_open(&id, argv[1], openconfig) != ET_OK) {
  printf("  et_producer: et_open problems\n"  );
  exit(1);
}
et_open_config_destroy(openconfig);

/* set level of debug output (everything) */
et_system_setdebug(id, ET_DEBUG_INFO);

/* attach to grandcentral station */
if (et_station_attach(id, ET_GRANDCENTRAL, &attach1) < 0) {
  printf("  et_producer: error in station attach\n"  );
  exit(1);
}

/* while the ET system is alive, do the following loop */
while (et_alive(id)) {
   /* read time for future statistics calculations */
   clock_gettime(CLOCK_REALTIME, &t1);
   /* loop NUMLOOPS times before printing out statistics */
   for (j=0; j < NUMLOOPS ; j++) {
      /* get CHUNK new events at a time */
      status = et_events_new(id, attach1, pe, ET_SLEEP, NULL, size, CHUNK, &count);
      if (status == 0) {
        /* everything is OK */;
      }
      else if (status == ET_ERROR_DEAD) {
        printf("  et_producer: request detach\n"  );
        break;
      }
      else if (status == ET_ERROR_TIMEOUT) {
        printf("  et_producer: got timeout\n"  );
        break;
      }
      else if (status == ET_ERROR_EMPTY) {
        printf("  et_producer: no events\n"  );
        break;
      }
      else if (status == ET_ERROR_BUSY) {
        printf("  et_producer: grandcentral is busy\n"  );
        break;
      }
      else if (status == ET_ERROR_WAKEUP) {
        printf("  et_producer: someone told me to wake up\n"  );
        break;
      }
      else if (status != ET_OK) {
        printf("  et_producer: request error\n"  );
        goto error;
      }

       /* write data, set priority, set control values here */
      if (1) {
```

```
        void *pdata;
        for (i=0; i < count; i++) {
          /*allow et_client modes 3 & 4 to work (see et_client.c below */
          et_event_setcontrol(pe[i], control, 4);
         et_event_getdata(pe[i], &pdata);
         memcpy(pdata, (const void *) &numbers[i], sizeof(int));
          et_event_setlength(pe[i], sizeof(int));
        }
       }

      /* put events back into the ET system */
      status = et_events_put(id, attach1, pe, count);
      if (status == ET_OK) {
        ;
      }
      else if (status == ET_ERROR_DEAD) {
        printf("  et_producer: put detach\n"  );
        break;
      }
      else if (status != ET_OK) {
        printf("  et_producer: put error\n"  );
        goto error;
      }
    } /* for NUMLOOPS */

    /* statistics */
    clock_gettime(CLOCK_REALTIME, &t2);
    time = (double)(t2.tv_sec - t1.tv_sec) + 1.e-9*(t2.tv_nsec - t1.tv_nsec);
    freq = (count*NUMLOOPS)/time;
    /* if numbers get too big, start over */
    if ((INT_MAX - freq_tot) < freq) {
      freq_tot   = 0;
      iterations = 1;
    }
    freq_tot += freq;
    freq_avg = freq_tot/iterations;
    iterations++;
    printf("  et_producer: %d Hz,  %d Hz Avg.\n"  , freq, freq_avg);

    /* if ET system is dead, wait here until it comes back */
    if (!et_alive(id)) {
      et_wait_for_alive(id);
    }
  } /* while(alive) */


  error:
    printf("  et_producer: ERROR\n"  );
    exit(0);
}
```

5.2 Event Consumer

An example of a program written to consume events produced by an ET system is adapted from the file *et_client.c*. It follows below:

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
#include <thread.h>
#include <et.h>

#define NUMEVENTS 200000
#define CHUNK 100

main(int argc,char **argv)
{
 int           i, j, status, swtch, numread, totalread=0;
 int           con[ET_STATION_SELECT_INTS];
 et_statconfig   sconfig;
 et_openconfig   openconfig;
 et_event      *pe[CHUNK];
 et_att_id      attach1;
 et_stat_id     my_stat;
 et_sys_id      id;
 int           selections[] = {17,15,-1,-1};  /* 17 & 5 are arbitrary */


 if (argc != 4) {
  printf("  Usage: et_client <et_filename> <station_name> <mode>\n"  );
  exit(1);
 }

 /* open local ET system and don't wait for it */
 et_open_config_init(&openconfig);
 if (et_open(&id, argv[1], openconfig) != ET_OK) {
  printf("  et_client: et_open problems\n"  );
  exit(1);
 }
 et_open_config_destroy(openconfig);

 /* User selects which type/mode of station to create - got 6 choices.
  * (Of course, many more combinations of settings are possible.)
  */
 swtch = atoi(argv[3]);

 /* set some common values */
 et_station_config_init(&sconfig);
 et_station_config_setuser(sconfig, ET_STATION_USER_MULTI);
 et_station_config_setrestore(sconfig, ET_STATION_RESTORE_OUT);
 et_station_config_setprescale(sconfig, 5);
 et_station_config_setcue(sconfig, 20);

 if (swtch==1) {
    /* DD system "  all"   mode */
    et_station_config_setselect(sconfig, ET_STATION_SELECT_ALL);
    et_station_config_setblock(sconfig, ET_STATION_BLOCKING);
 }
 else if (swtch==2) {
    /* DD system "  on req"   mode */
    et_station_config_setselect(sconfig, ET_STATION_SELECT_ALL);
    et_station_config_setblock(sconfig, ET_STATION_NONBLOCKING);
 }
```

```
else if (swtch==3) {
   /* DD system "  condition"   mode */
   et_station_config_setselect(sconfig, ET_STATION_SELECT_MATCH);
   et_station_config_setblock(sconfig, ET_STATION_BLOCKING);
   et_station_config_setselectwords(sconfig, selections);
}
else if (swtch==4) {
   /* new non-blocking "  condition"   mode */
   et_station_config_setselect(sconfig, ET_STATION_SELECT_MATCH);
   et_station_config_setblock(sconfig, ET_STATION_NONBLOCKING);
   et_station_config_setselectwords(sconfig, selections);
}
else if (swtch==5) {
   /* user's condition, blocking  mode */
   et_station_config_setselect(sconfig, ET_STATION_SELECT_USER);
   et_station_config_setblock(sconfig, ET_STATION_BLOCKING);
   et_station_config_setselectwords(sconfig, selections);
   if (et_station_config_setfunction(sconfig, "  et_my_function"  ) == ET_ERROR) {
         printf("  et_client: cannot set function\n"  );
         exit(1);
   }
   if (et_station_config_setlib(sconfig, "  /.../libet_user.so"  ) == ET_ERROR) {
    printf("  et_client: cannot set library\n"  );
         exit(1);
   }
}
else if (swtch==6) {
   /* user's condition, nonblocking mode */
   et_station_config_setselect(sconfig, ET_STATION_SELECT_USER);
   et_station_config_setblock(sconfig, ET_STATION_NONBLOCKING);
   et_station_config_setselectwords(sconfig, selections);
   if (et_station_config_setfunction(sconfig, "  et_my_function"  ) == ET_ERROR) {
         printf("  et_client: cannot set function\n"  );
         exit(1);
   }
   if (et_station_config_setlib(sconfig, "  /.../libet_user.so"  ) == ET_ERROR) {
    printf("  et_client: cannot set library\n"  );
         exit(1);
   }
 }

 /* set level of debug output */
 et_system_setdebug(id, ET_DEBUG_INFO);

 /* create the station */
 if ((status = et_station_create(id, &my_stat, argv[2], sconfig)) < ET_OK) {
  if (status == ET_ERROR_EXISTS) {
   /* my_stat contains pointer to existing station */;
   printf("  et_client: station already exists\n"  );
  }
  else if (status == ET_ERROR_TOOMANY) {
   printf("  et_client: too many stations created\n"  );
   goto error;
  }
  else {
   printf("  et_client: error in station creation\n"  );
   goto error;
  }
 }
```

```
et_station_config_destroy(sconfig);

/* attach to newly created station */
if (et_station_attach(id, my_stat, &attach1) < 0) {
  printf("  et_client: error in station attach\n"  );
  goto error;
}

/* loop, while ET system is alive, to read and write events */
while (et_alive(id)) {
  /* example of reading array of up to "  CHUNK"   events */
  status = et_events_get(id, attach1, pe, ET_SLEEP, NULL, CHUNK, &numread);

  if (status == ET_OK) {
    /* everything is OK */ ;
  }
  else if (status == ET_ERROR_DEAD) {
    printf("  et_client: detach\n"  );
    goto end;
  }
  else if (status == ET_ERROR_TIMEOUT) {
    printf("  et_client: got timeout\n"  );
    goto end;
  }
  else if (status == ET_ERROR_EMPTY) {
    printf("  et_client: no events\n"  );
    goto end;
  }
  else if (status == ET_ERROR_BUSY) {
    printf("  et_client: station is busy\n"  );
    goto end;
  }
  else if (status == ET_ERROR_WAKEUP) {
    printf("  et_client: someone told me to wake up\n"  );
    break;
  }
  else if (status != ET_OK) {
    printf("  et_client: get error\n"  );
    goto error;
  }

  /* print data */
  if (0) {
    int pri, len, *data;
    for (j=0; j< numread; j++) {
      et_event_getdata(pe[j], (void **) &data);
      et_event_getpriority(pe[j], &pri);
      et_event_getlength(pe[j], &len);
      et_event_getcontrol(pe[j], con);
      printf("  et_client data = %d, pri = %d, len = %d\n"  , *data, pri, len);
      for (i=0; i < ET_STATION_SELECT_INTS; i++) {
        printf("           con[%d] = %d\n"  , i, con[i]);
            }
    }
  }

  /* example of putting array of events */
  status = et_events_put(id, attach1, pe, numread);
```

```
      if (status == ET_ERROR_DEAD) {
        printf("  et_client: detach\n"  );
        goto end;
      }
      else if (status != ET_OK) {
        printf("  et_client: put error\n"  );
        goto error;
      }
      totalread += numread;

      end:
        /* print something out after having read NUMEVENTS events */
        if (totalread >= NUMEVENTS) {
          totalread = 0;
          printf("   et_client: %d events\n"  , NUMEVENTS);
        }
         /* if ET system is dead, wait here until it comes back */
        if (!et_alive(id)) {
          et_wait_for_alive(id);
        }
    } /* while(alive) */

    error:
      free(pe);
      printf("  et_client: ERROR\n"  );
      exit(0);
}
```

Chapter 6 Modifying The ET System

For the reader that needs to tune the ET system for better or even different performance, this is the chapter that needs to be read.

6.1 Versions

The header file “et_private.h” defines the macro ET_VERSION whose value denotes the version of the ET software. When a user calls et_open, the routine checks to see if its version and the version of the ET system it is opening is the same. If not, an error is returned. Thus, when a user makes fundamental changes to the ET software and recompiles it, the value of ET_VERSION should also be changed to some value over 1,000. GIving the version a large number allows the author and distributors of ET to use the version number for successive releases of the software without conflicting with the version a user makes with specific modifications. In this way, incompatible version of ET will always give users a warning.

Modifying the definitions of constants defined in “et.h”, such as ET_STATION_SELECT_INTS, ET_ATTACHMENTS_MAX, ET_FILENAME_LENGTH, or ET_STATNAME_LENGTH, may cause problems if the user is not careful. Difficulties may arise when more than one ET library exist - each with different definitions of one of the above constants. When network communications occur between clients using one library and ET systems using another library, it is likely that one of the processes involved will crash. Thus, for these modifications, be sure to change ET_VERSION.

6.2 Event Selection

6.2.1 Selection Integers

For users that need additional control over the flow of their events, take a look at the file “ et.h”It is possible to increase the value of the macro ET_STATION_SELECT_INTS and recompile ET (provided of course you have the source code). What this buys one is the simultaneously increase of both the number of select words (actually integers) for each station and also the corresponding number of control

words (integers) of each event. Thus, one is not stuck trying to cram as much matching or flow information into the default four integers as possible.

Changing the value of ET_STATION_SELECT_INTS and recompiling can cause fatal errors if not done properly. If an ET system and all its users are not using either the same shared library or one compiled with an identical code, then network communications will fail with unpredictable results. The way to avoid potential problems of this type is to assign another version number to modified ET systems (libraries) by changing the value of ET_VERSION in " et_private.h" (see above).

6.2.2 Selection Functions

This modification suffices for only the simple need of more control information. What if the user needs to change the manner in which a station selects events altogether? The solution - mentioned in sections 3.2 and 4.3 - is for the user to write a routine which does the selection. An example is provided in the source code. Look in the /.../et/src directory and at two files. The first, shown below, is et_userfunction.c :

```
#include <et.h>

int et_users_function(et_sys_id id, et_stat_id stat_id, et_event *pe)
{
  int select[ET_STATION_SELECT_INTS],
      control[ET_STATION_SELECT_INTS];

  et_station_getselectwords(id, stat_id, select);
  et_event_getcontrol(pe, control);

  /* access event control ints thru control[N] */
  /* access station selection ints thru select[N] */

  /* return 0 if it is NOT selected, 1 if it is */

  if (some condition) {
    return 1;
  }
  return 0;
}
```

The first argument is the ET system id which gives the user access to all system information. The second is the station the user is selecting events for, and the last is a pointer to event that the user is wondering whether to select or not. Simply return one (1) if the event is selected, and zero (0) if it is not.

Notice that the routines “et_station_getselectwords” and “ et_event_getcontrol”will prove extremely useful as they allow the user access to all the selection and control integers. The name of this function is completely up to the user. The only obvious restriction is that it shouldn’t conflict with names in the ET library (look in et.h and et_private.h). The name of the file is also up to the user provided corresponding changes to the Makefile are made.

The second file of interest is the Makefile. There is one place where et_userfunction.c is compiled into an object file and another where the object file is compiled into a shared library. The name of the shared library is again up to the user. Simply change the value of “ LIB_USER” in the Makefile.

The names of your function and shared library are parameters in the definition of a station and are thus subject to a length limit. The function name is limited to ET_FUNCNAME_LENGTH - 1 chars and the lib name is limited to ET_FILENAME_LENGTH - 1 chars. These limits are enforced in the routines et_station_config_setfunction and et_station_config_setlib.

6.3 Setting Heartbeat and Heartmonitor Periods

There are two time periods that are adjustable by modifying their values in “et_private.h” and recompiling ET. The first of these two periods is the time between heartbeats. As the reader should be aware of by now, each process opening an ET system has a thread start up which provides a heartbeat. By default it is set to 0.5 seconds:

```
#define ET_BEAT_SEC   0

#define ET_BEAT_NSEC  500000000
```

The second is time period between readings of the system heartbeat if you are a user or client heartbeats if you are the system. Remember that upon opening an ET system, another thread starts which monitors the appropriate heartbeats. The default monitor period is 1.6 seconds:

```
#define ET_MON_SEC    1

#define ET_MON_NSEC   600000000
```

The **CRUCIAL** point to remember is that the heartbeat must be faster than the heartmonitor. If the heartmonitor finds that the system heartbeat has not changed in successive monitorings, then it declares that the ET system is dead. The same is true for the system monitoring clients. If your process declares that the ET system is dead, no further dealings with it are possible until the system heartbeat returns.

Notice that the default has a large safety margin built in. The hearts are beating more than three times faster than the monitors are looking at them. This ensures that flakiness in unix’s handling of timing, sleeping, and the scheduling of processes will not interfere.

The advantage of decreasing the beat and monitor times is that the system and user processes have a much quicker response to the world. The disadvantage is that it slows down the performance of the whole system. The author has run with a beat time of 0.3 seconds and a monitor time of 1 second with no problems.

The reader should be aware that on Solaris systems the clock is 100Hz, meaning that when a thread or process encounters a " sleep" or "nanosleep" command or is swapped out, it does nothing for a minimum of 0.01 seconds. However, a Solaris workstation can be setup to use a high resolution clock of 1000Hz which gives one a lot better control over timing - talk to your system administrator. The down side of speeding up the clock is increased overhead and possibly slower performance.

6.4 Setting the Number of Attachments and Processes

In specifying the configuration of a system, which is passed on to the routine "et_system_start", the user can specify the maximum number or attachments and the maximum number of processes which can use the ET system being created. Both of these values are limited however. They cannot exceed the values set by the macros ET_ATTACHMENTS_MAX and ET_PROCESSES_MAX. The reason for doing it that way is that programming is greatly simplified.

By looking in the file "et_private.h", the reader can see that the default value of ET_ATTACHMENTS_MAX is 50 and that the macro ET_PROCESSES_MAX is set to ET_ATTACHMENTS_MAX. If more attachments or processes are desired, then these 2 values can be increased and ET must be recompiled. (Be sure to change ET_VERSION as well).

6.5 Setting Defaults

Although a user can set ET system parameters such as the number of events and their size, it may be nice if some of these parameters could be made the default. This is possible by editing a few lines in the file " et.h"The value of a station's cue and prescale along with the value of a system's number of events, max number of temporary events, size of events, and max number of stations can be set to a user's preferred default by changing (respectively): ET_STATION_CUE, ET_STATION_PRESCALE, ET_SYSTEM_EVENTS, ET_SYSTEM_NTEMPS, ET_STATION_ESIZE, ET_SYSTEM_NSTATS. A recompilation is necessary.

Chapter 7

Remote ET

7.1 Remote Node Operation Overview

It is possible to have an ET system on one machine and its clients (called remote clients) on another. Remote clients can call all the routines that local clients can. Of course, the speed of transferring events over the network is quite a bit slower than the speed of accessing shared memory.

The way this is done is that each ET system has a server "built in" as it were. That is, there are two threads in the ET system's process which facilitate accessing that system from another computer. One of these threads responds to the broadcasts of remote clients trying to find an ET system of a particular name somewhere on the network. The response is simply sending back the port number of the socket that the other thread is listening on. This second thread, when connected with a client, is the one which handles all the receiving and sending of events and other information with the client.

It is this arrangement that makes it possible to run an ET system on Linux. At this point, Linux does not allow the sharing of pthread mutexes and condition variables between processes. This makes it impossible to access the shared memory of the ET system safely between processes. However, this problem can be circumvented by treating local Linux clients as remote clients. The server built into the ET system handles all ET routines that require handling these mutexes and sends this client pointers to events which the client can then access in shared memory. This makes running ET systems on Linux slower than those running on Solaris. However, on the author's Linux dual 200MHz pentium pro machine, an ET system which consists of a single producer and does no data copying can handle events at 200kHz.

7.2 System Connection

In order to have things work seamlessly, the user needs to make some decisions. First of all, the decision needs to be made whether clients connect to ET systems using a direct connection by specifying host and port, or possibly by using broadcasting or multicasting.

When a client does not know the host and port, either broadcasting or multicasting is a must. Although multicasting is the way of the future, some operating systems (eg. Solaris 2.6) do not yet fully support it. To be more specific. ET systems that run on Solaris 2.6 and earlier versions cannot use multicasting. However, remote clients running on them can (as long as they use the default value for TTL). For network applications that use the new IPv6 standard, multicasting must be used as broadcasting is not supported. On Linux (Redhat 6.0) multicasting works as far as it's been tested. Broadcasting, on the other hand is supported by all operating systems.

7.2.1 Direct Connection

There are times when using either broadcasting or multicasting are inconvenient or impossible. For example, if an ET system and a client are on different subnets, broadcasting from one to the other is stopped by any routers unless such are reprogrammed to allow broadcasting to get through - a hassle in any case. And, as was mentioned above, multicasting is not yet supported by all operating systems. In situations such as these, a direct connection can be made.

When the ET system is started up, it's configuration can be set by using the "et_system_config_..." set of routines. A call to "et_system_config_setserverport" sets the port number of the ET system's tcp server thread in that particular configuration. If the port is unavailable when actually starting the ET system by a call to "et_system_start" using that same configuration, the process will exit with an error message. Thus, using this routine guarantees that the ET system will have its server thread at that port. Note that if the server port is NOT explicitly set in this way, then by default **ET_SERVER_PORT** (defined as 11111 in "et.h") is used as the port number. If this port is busy, it's incremented by one and tried again and so on until there is success. If it has tried 2000 port numbers with no success, the process exits with an error message.

Remote clients need to know the port number and the host name that the ET system is on. Then using "et_open_config_setserverport" the port can be set, using "et_open_config_sethost" the host can be set, and using "et_open_config_setcast" a direct connection can be specified. It is irrelevant as to whether the ET system has been specified as set for broadcasting or multicasting (the only two options for the system).

7.2.2 Broadcasting

Broadcasting is done to IP addresses of two relevant types. An IP address in dotted-decimal form (eg 128.7.6.21) can be represented as {netid, subnetid, hostid}. The first type of broadcast address is subnet-directed and is of the form {netid, subnetid,-1} where -1 simply means that that part of the address is composed of all 1's in binary. For example, if 128.7.6 is the subnet with a mask of

255.255.255.0, then 128.6.7.255 is the broadcast address to that subnet. A broadcast will be received by all machines on that particular subnet.

The second type of broadcast address is all-subnets-directed and is of the form {netid, -1, -1}. Again, if 128.7 is the netid, then the broadcast address is 128.7.255.255. This type of broadcast will be received by all machines on all the subnets at that site. Keep in mind that some older systems still send packets to 255.255.255.255 when broadcasting. If this seems confusing, ask your system administrator to tell you the broadcast address of your subnet or of all your subnets.

To set an ET system to respond to broadcasts and not multicasts, use the " et_system_config_setcast"routine to set the configuration to a setting of **ET_BROADCAST**. (A client can still connect directly to this ET system regardless of this configuration parameter's value). The question then arises how the ET system's subnet is specified. The subnet that the ET system listens on for broadcasts can be set directly by calling" et_system_config_setaddress"However, if the user does not know the local subnet address and does not set it, it will be found automatically (see "et_mybroadcastaddr" in "et_network.c" for details). Normally automatic finding of subnets is no problem, but difficulties can arise when a host is on more that one subnet. If difficulties do arise, explicitly setting the subnet address is the way out.

Configuring a client to use broadcasting and not multicasting is very similar to setting up the ET system to do the same. Use the " et_open_config_setcast"routine to set the configuration to a setting of ET_BROADCAST. Call "et_open_config_setaddress" to set the subnet address to broadcast on (one of the local subnets of the client). A subnet address is found automatically if it is not set explicitly. Keep in mind that if an ET system is set up to listen for broadcasts, then clients need to be set up for broadcasting.

7.2.3 Multicasting

In multicasting a client sends out a packet to a special multicast IP address. The listeners (ET systems) sign up to receive any packets send to that address and only computers hosting such listeners will receive the packets - not all machines on the subnet as is the case in broadcasting. Multicasting has the ability to go beyond the local subnet and thus is more flexible than broadcasting. The following table lists all available multicast addresses as well as "TTL" values reproduced from Unix Network Programming, Volume 1 by Richard Stevens:

Table 1. Multicast Addresses

Scope	IPv6 scope	IPv4	
		TTL scope	administrative scope
node-local	1	0	
link-local	2	1	224.0.0.0 to 224.0.0.225
site-local	5	<32	239.255.0.0 to 239.255.255.255
org.-local	8		239.192.0.0 to 239.195.255.255
global	14	<255	224.0.1.0 to 238.255.255.255

Although this author is NOT an expert ..., the use of TTL values and ranges of addresses is meant to set the range or the scope of the multicasts. The use of setting the TTL value for scoping is accepted and even recommended practice with a default value of one meaning the local subnet only. However, administrative scoping is preferred when possible. The range 239.0.0.0 to 239.25.255.255 is the administratively scoped IPv4 multicast space. “Addresses in this range are assigned locally by an organization but are not guaranteed to be unique across organizational boundaries. An organization must configure its boundary routers (multicast routers at the boundary of the organization) not to forward multicast packets destined to any of these addresses” .

In short, pick an address between 239.0.0.0 and 239.25.255.255 for use at one particular site. If this is confusing, talk to your system administrator and ask for a safe multicast address for your use.

According to Harold in Java Network Programming, the Internet Assigned Numbers Authority (IANA) is responsible for handing out permanent multicast addresses as needed and do so manually as demand is still small. The following is a table, taken from the mentioned book, showing some of the taken multicast addresses:

Table 2. Multicast Addresses In Use

Domain Name	IP Address	Purpose
BASE-ADDRESS.MCAST.NET	224.0.0.0	Reserved base address - never assigned
ALL-SYSTEMS.MCAST.NET	224.0.0.1	All systems on the local subnet
ALL-ROUTERS.MCAST.NET	224.0.0.2	All routers on the local subnet

Table 2. Multicast Addresses In Use

Domain Name	IP Address	Purpose
DVMRP.MCAST.NET	224.0.0.4	All Distance Vector Multicast Routing Protocol routers on this subnet
MOBILE-AGENTS.MCAST.NET	224.0.0.11	Mobile-Agents on the local subnet
DHCP-AGENTS.MCAST.NET	224.0.0.12	Allows client to locate DHCP server on local subnet
PIM-ROUTERS.MCAST.NET	224.0.0.13	All Protocol Independent Multicasting routers on this subnet
RSVP-ENCAPSULATION.MCAST.NET	224.0.0.14	RSVP-ENCAPSULATION on this subnet
NTP.MCAST.NET	224.0.1.1	Network Time Protocol
SGI-DOG.MCAST.NET	224.0.1.2	Silicon Graphics Dogfight game
NSS.MCAST.NET	224.0.1.6	Name Service Center
AUDIONEWS.MCAST.NET	224.0.1.7	Audio News multicast
SUB-NIS.MCAST.NET	224.0.1.8	Sun's NIS+ Information Service
MTP.MCAST.NET	224.0.1.9	Multicast Transport Protocol
	224.0.1.10 - 224.0.1.19	Stuff
EXPERIMENT.MCAST.NET	224.0.1.20	Experiments that do NOT go beyond the local subnet.
	224.0.1.23 - 224.0.1.32	Stuff
	224.0.6.000 - 224.0.6.127	ISIS project for robust software development
	224.0.9.000 - 224.0.9.255	Internet Railroad project - 45Mbit/sec backbone

Table 2. Multicast Addresses In Use

Domain Name	IP Address	Purpose
	224.2.0.0 - 224.2.255.255	MBONE

To set an ET system to respond to multicasts, use the "et_system_config_setcast" routine to set the configuration to a setting of **ET_MULTICAST**. (A client can still connect directly to this ET system regardless of this configuration parameter's value). The address that the ET system listens on for multicasts can be set by calling " et_system_config_setaddress" .

Configuring a client to use multicasting and not broadcasting is very similar to setting up the ET system to do the same. Use the " et_open_config_setcast" routine to set the configuration to a setting of ET_MULTICAST. Call "et_open_config_setaddress" to set the address to multicast on. Keep in mind that if an ET system is set up to listen for multicasts, then clients need to be set up for multicasting.

7.2.4 Port Selection for Broad/Multicasting

In addition to choosing either broadcasting or multicasting and choosing the IP address, the user must also choose the port number for these IP communications. The Internet Assigned Numbers Authority (IANA) states that the range of port numbers from 0 to 1023 are controlled and assigned by the IANA. Thus, these are off limits. The ports 1024 to 49151 are NOT controlled by the IANA and are available for use, but the IANA registers and lists the uses of these ports as a convenience to the internet community. For example, ports 6000 to 6063 are assigned for an X window server for both TCP and UDP. Generally, the higher numbered ports are less likely to be used. Finally, ports 49152 to 65535 are called dynamic or private or ephemeral ports. The IANA says nothing about these.

Use the routine "et_system_config_setport" to configure an ET system to listen for broad/multicast on a particular port. Use " et_open_config_setport" to configure a client to send broad/multicasts to a particular port. These must be the same value for things to work. By default, if not set explicitly, they are both set to **ET_BROADCAST_PORT** (defined as 11111 in " et.h").

7.2.5 Defaults & Macros

When defining a configuration to use in opening an ET system, the defaults are to use broadcasting (ET_BROADCAST) to port ET_BROADCAST_PORT (defined as 11111 in "et.h") on a subnet address that is automatically found. If the automatic finding of the subnet fails, a value of **ET_BROADCAST_ADDR** is used (defined as "129.57.35.255" in " et.h" - the author's personal subnet). The macro **ET_MULTICAST_PORT** is also similarly defined to be 11111, while

the macro **ET_MULTICAST_ADDR** is defined to be "239.200.0.0". The value of **ET_MULTICAST_TTL** is one. All of these macros are only defined for the users' convenience.

7.2.6 Examples

When setting up an ET system, the following sequence of calls will set things up for broadcasting:

```
et_sys_id id;
et_sysconfig config;

/* initialize configuration */
et_system_config_init(&config);
/* remote users use broadcast instead of multicast to
find me */
et_system_config_setcast(config, ET_BROADCAST);
/* remote users broadcast to this port */
et_system_config_setport(config, ET_BROADCAST_PORT);
/* listen to broadcasts on this subnet */
et_system_config_setaddress(config, "129.57.35.255");
/* start ET system */
et_system_start(&id, config);
/* release configuration's allocated memory */
et_system_config_destroy(config);
```

When setting up an ET system that uses multicasting, try the following:

```
et_sys_id id;
et_sysconfig config;

/* initialize configuration */
et_system_config_init(&config);
/* remote users' use multicast to find me */
et_system_config_setcast(config, ET_MULTICAST);
/* remote users multicast to this port */
et_system_config_setport(config, ET_MULTICAST_PORT);
 /* listen to multicasts to this address */
et_system_config_setaddress(config, ET_MULTICAST_ADDR);
/* start ET system */
et_system_start(&id, config);
/* release configuration's allocated memory */
et_system_config_destroy(config);
```

When setting up an ET system with the server on a specific port, try the following:

```
et_sys_id id;
```

```
et_sysconfig config;

/* initialize configuration */
et_system_config_init(&config);
/* remote users use broadcast instead of multicast to find me */
et_system_config_setcast(config, ET_BROADCAST);
/* remote users broadcast to this port */
et_system_config_setport(config, ET_BROADCAST_PORT);
/* set port of tcp server thread */
et_system_config_setserverport(config, 11222);
/* listen to broadcasts on this subnet */
et_system_config_setaddress(config, "129.57.35.255");
/* start ET system */
et_system_start(&id, config);
/* release configuration's allocated memory */
et_system_config_destroy(config);
```

When setting up a client to open an ET system on an unknown host (may be local or remote), and it's trying to find a system that's using broadcasting on port ET_BROADCAST_PORT at subnet address 129.57.35.255, then include the following code:

```
et_sys_id id;
et_openconfig config;

/* initialize configuration */
et_open_config_init(&config);
/* ET is on an unknown host that may be anywhere */
et_open_config_sethost(config, ET_HOST_ANYWHERE);
/* use broadcast instead of multicast to find ET system */
et_open_config_setcast(config, ET_BROADCAST);
/* remote users broadcast to this port */
et_open_config_setport(config, ET_BROADCAST_PORT);
/* on this subnet address */
et_open_config_setaddress(config, "129.57.35.255");
/* open the ET system */
et_open(&id, "et_name", config);
/* release configuration's allocated memory */
et_open_config_destroy(config);
```

When setting up a client that knows the name of the remote host (ethost.mylab.org) running the ET system, and it's trying to find an ET system that's using multicasting on port ET_MULTICAST_PORT at address ET_MULTICAST_ADDR, then include the following code:

```
et_sys_id id;
et_openconfig config;

/* initialize configuration */
```

```
et_open_config_init(&config);
/* ET is running on ethost */
et_open_config_sethost(config, "ethost.mylab.org");
/* use multicast to find ET system */
et_open_config_setcast(config, ET_MULTICAST);
/* remote users multicast to this port */
et_open_config_setport(config, ET_MULTICAST_PORT);
/* remote users multicast to this address */
et_open_config_setaddress(config, ET_MULTICAST_ADDR);
/* open the ET system */
et_open(&id, "et_name", config);
/* release configuration's allocated memory */
et_open_config_destroy(config);
```

When setting up a client to open an ET system on an known host (129.182.54.67), and it's trying to directly connect to it on server port 12345, then include the following code:

```
et_sys_id id;
et_openconfig config;

/* initialize configuration */
et_open_config_init(&config);
/* ET is on an unknown host that may be anywhere */
et_open_config_sethost(config, "129.182.54.67");
/* use a direct connection to the ET system */
et_open_config_setcast(config, ET_DIRECT);
/* ET system's server is on this port */
et_open_config_setserverport(config, 12345);
/* open the ET system */
et_open(&id, "et_name", config);
/* release configuration's allocated memory */
et_open_config_destroy(config);
```

7.3 Remote Programming Details

7.3.1 Errors

As mentioned previously, **ET_ERROR_NOREMOTE** is the error returned when calling a routine which is not supported for remote use. Currently, however, there are no routines which return this error. Some remote user errors are given by **ET_ERROR_REMOTE** - those errors which are unique to a remote user and do not occur locally. In practice, this error is returned when memory cannot be allocated by the remote user. If there are errors in reading or writing over the network, the errors generated will be **ET_ERROR_READ** or **ET_ERROR_WRITE**.

7.3.2 Remote Behavior on a Local Host

It is possible to tell clients to run the code that a remote client runs even if it is running on the same computer as the ET system. In this case, all communication with the ET system is done through sockets with no usage of the shared memory. This is done by calling "et_open_config_setmode" with the **ET_HOST_AS_REMOTE** option. The default mode is **ET_HOST_AS_LOCAL**.

7.3.3 Modifying Events

After opening an ET system, creating a station, and attaching to it, users are ready to start reading events. There are a few details to keep in mind when doing so remotely.

Remote users can gain quite a bit of efficiency by minimizing communication with the ET system. The minimizing of communication is done transparently and is the default mode of operation. That is, when a remote user calls " et_event(s)_get" the ET system copies the events and sends them over the network to the user but also immediately puts them back into the ET system with a call to "et_event(s)_put". There may be times, however, when a user first wishes to modify the events and then send them back over the network to the ET system and which then puts them back in. To aid in this effort an extra flag is introduced, **ET_MODIFY**. By ORing this flag to ET_SLEEP, ET_TIMED, or ET_ASYNC, the user announces an intention to modify the requested event. Thus, when the ET server initially gets the event for the remote user, it does NOT put it back into the ET system immediately afterwards. It waits until the user has called " et_event(s)_put" before doing that. Without this flag, the server puts the events back into the ET system immediately.

There may be occasions when the remote user doesn't want to modify the data but only the header information such as the priority, control works, and such. In that case it makes no sense to send all the data back to the ET system when putting the event back. By using the flag **ET_MODIFY_HEADER** instead of ET_MODIFY, only the header information will be sent back - speeding up communication greatly.

7.3.4 Creating New Events

When producing events remotely, the user's call to "et_event(s)_new" causes the server to send a bunch of pointers to these new events to the remote user. I know this does not appear to make any sense; however, it does cause remote and local programs to behave in the same manner. What happens on the receiving end is the user simply allocates memory locally to store the event(s). When the user calls " et_event(s)_put" the events are sent over the network to the server while the locally allocated memory is freed.

7.3.5 Multi-Threading

If the user's remote client is a multi-threaded program, some caution needs to be used. A problem may arise if one thread calls " et_open" returning an ET system id which is then used by other threads. If more than one thread call routines which simultaneously communicate to the server on the same socket, then the server will become confused. To avoid this problem, each thread that wants access to an ET system needs to do its own " et_open"and thus communicate on

7.4 Swapping Data

Transferring data between machines where one is big endian (the most significant byte is placed in the lowest memory address) and the other is little endian (the least significant byte is placed in the lowest memory address), requires the data to be " swapped"Since in general a user may not be knowledgeable about the machine on which a particular event was originally produced, a simple call to the function "**et_event_needtoswap(et_event *pe, int *swap)**" will reveal whether the data needs to be swapped or not. If the return value placed in *swap* is **ET_NOSWAP**, no swapping is necessary; however, if the return value is **ET_SWAP**, then the opposite is true.

The ET system automatically keeps track of the endianness of an event's data. However, the user may want to forcibly set the data's endianness for some reason. In that case, a call to "**et_event_setendian(et_event *pe, int endian)**" can be made. The endianness can be set to ET_ENDIAN_BIG, ET_ENDIAN_LITTLE, ET_ENDIAN_LOCAL (same endian as local host), ET_ENDIAN_NOTLOCAL (opposite endian as local host), or ET_ENDIAN_SWITCH (switch the endian from whatever it is). This routine does NOT swap the data but simply keeps track of the data's endianness in the event's header. A user may also read the endianness of an event's data by a call to "**et_event_getendian(et_event *pe, int *endian)**". It returns either ET_ENDIAN_BIG or ET_ENDIAN_LITTLE.

The routine "**et_event_CODAswap(et_event *pe)**" is provided for those who need to swap data in CODA format. The data is manipulated in the existing event's data buffer so that the function irreversibly mangles the data.

Users of data formats other than CODA must provide their own swapping routines.

Another routine of interest is "**et_system_getlocality(et_sys_id id, int *locality)**". This returns the value **ET_REMOTE** in the variable "locality" if the ET system is remote, **ET_LOCAL** if it is local, and **ET_LOCAL_NOSHARE** if it is local but is on a machine which does not allow sharing of pthread mutexes across processes (e.g. Linux).

7.5 Transferring Events Between 2 ET Systems

While it is certainly possible for a user to copy events from one ET system and place them in another with "normal" ET function calls, the ET system provides a more efficient way to do this. By using ET's bridging software, unnecessary coping of the data may be eliminated from the procedure. Regardless of whether the ET systems are on the same or different computers or if the process running the bridging routine is on one or the other or on yet a third machine, the transfer should take place smoothly. It will save time except perhaps when both ET systems and the bridging process are on the same machine in which case only a single copy of the data is made - no different than when using the "normal" ET function calls. A call to the following function will take care of all the details:

et_events_bridge(et_sys_id id_from, et_sys_id id_to, et_att_id att_from, et_att_id att_to, et_bridgeconfig bconfig, int num, int *ntransferred).

The arguments are respectively: the ID of the ET system from which the events are copied, the ID of the ET system to which the events are going, the attachment to a station on the "from" ET system, the attachment to a station on the "to" ET system (usually an attachment to GrandCentral), a configuration argument, the total number of events desired to be transferred, and the total number of events that were actually transferred at the routine's return. The configuration argument may be NULL in which case defaults are used.

The configuration for bridging events is very similar to the configuration for opening a system or creating a system. There are a number of functions used to create and define the config argument. It is initialized by a call to **et_bridge_config_init** (et_bridgeconfig *config). When the user is finished using the configuration, **et_bridge_config_destroy** (et_bridgeconfig config) must be called in order to properly release all memory used.

After initialization, calls can be made to functions which set various properties of the specific configuration. Calls to these setting functions will fail unless the configuration is first initialized. The functions used to SET these properties are listed below along with an explanation for each:

1. **et_bridge_config_setmodefrom**(et_bridgeconfig config, int val) : setting val to ET_SLEEP, ET_TIMED, or ET_ASYNC determines the mode of getting events from the "from" ET system. The default is ET_SLEEP.
2. **et_bridge_config_setmodeto**(et_bridgeconfig config, int val) : setting val to ET_SLEEP, ET_TIMED, or ET_ASYNC determines the mode of getting new events from the "to" ET system. The default is ET_SLEEP.

3. **et_bridge_config_setchunkfrom**(et_bridgeconfig config, int val) : setting val sets the maximum number of events to get from the "from" ET system in a single call to et_events_get - the chunk size if you will. The default is 100.
4. **et_bridge_config_setchunkto**(et_bridgeconfig config, int val) : setting val sets the maximum number of new events to get from the "to" ET system in a single call to et_events_new - the chunk size if you will. The default is 100.
5. **et_bridge_config_settimeoutfrom**(et_bridgeconfig config, struct timespec val) : setting val sets the time to wait for the "from" ET system when the mode is set to ET_TIMED. The default is 0 sec.
6. **et_bridge_config_settimeoutto**(et_bridgeconfig config, struct timespec val) : setting val sets the time to wait for the "to" ET system when the mode is set to ET_TIMED. The default is 0 sec.
7. **et_bridge_config_setfunc**(et_bridgeconfig config, ET_SWAP_FUNCPTR func) : setting func to a function pointer (function name) means that the function will be called to swap data whenever it's determined to be necessary. Using this feature is a convenient way of swapping data while it's being moved from one ET system to another with no intervention from the user needed. The function must be of the form: int func(et_event *src, et_event *dest, int bytes, int same_endian) . It returns ET_OK if successful otherwise ET_ERROR. The arguments consists of: src which is a pointer to the event whose data is to be swapped, dest which is a pointer to the event where the swapped data goes, bytes which tells the length of the data in bytes, and same_endian which is a flag equalling one if the machine and the data are of the same endian and zero otherwise. This function must be able to work with src and dest being the same event. With this as a prototype, the user can write a routine which swaps data in the appropriate manner. Notice that the first two arguments are pointers to events and not data buffers. This allows the writer of such a routine to have access to any of the event's header information. In general, such functions should NOT call et_event_setendian in order to change the registered endian value of the data. This is already taken care of in et_events_bridge. The default is NULL which means no swapping is done.
8. **et_bridge_CODAswap**(et_event *src, et_event *dest, int bytes, int same_endian) : this is a function that can be used in et_bridge_config_setfunc if the user wants to swap CODA format data.

There are corresponding "et_bridge_config_get..." functions to get the configuration values of everything except the swapping function.

Chapter 8 Useful Macros

8.1 Event Priority

1. **ET_LOW** - low event priority
2. **ET_HIGH** - high event priority

8.2 Event Data Status

3. **ET_DATA_OK** - event data is OK
4. **ET_DATA_CORRUPT** - event data is corrupt
5. **ET_DATA_POSSIBLY_CORRUPT** - event data may be corrupt

8.3 String Lengths

6. **ET_FILENAME_LENGTH** - maximum length of ET system file name + 1
7. **ET_FUNCNAME_LENGTH** - maximum length of user's event selection function name
8. **ET_STATNAME_LENGTH** - maximum length of station name

8.4 Waiting Modes for Events

9. **ET_ASYNC** - event get routines' async wait mode
10. **ET_SLEEP** - event get routines' sleep wait mode
11. **ET_TIMED** - event get routines' timed wait mode
12. **ET_MODIFY** - remote event will be modified and put back. This flag gets ORed with one of ET_SLEEP, ET_TIMED, or ET_ASYNC.
13. **ET_MODIFY_HEADER** - remote event will have its header information modified and put back. This flag gets ORed with one of ET_SLEEP, ET_TIMED, or ET_ASYNC.

8.5 Station Related

8.5.1 General

14. **ET_GRANDCENTRAL** - station id for grandcentral
15. **ET_STATION_SELECT_INTS** - number of event selection integers associated with each station

8.5.2 Station Status

16. **ET_STATION_UNUSED** - station status when unused
17. **ET_STATION_CREATING** - station status when it is being created
18. **ET_STATION_IDLE** - station status when it has been created but has no processes attached to it
19. **ET_STATION_ACTIVE** - station status when it is active

8.5.3 Number of Users per Station

20. **ET_STATION_USER_MULTI** - mode allowing many users to attach to a particular station
21. **ET_STATION_USER_SINGLE** - mode allowing one user only to attach to a particular station

8.5.4 Station Blocking Modes

22. **ET_STATION_NONBLOCKING** - mode which does not block the flow of events through the ET system at a particular station
23. **ET_STATION_BLOCKING** - mode which blocks the flow of events through the ET system so all must pass through a particular station

8.5.5 Event Selection Modes

24. **ET_STATION_SELECT_ALL** - station mode in which all events selected
25. **ET_STATION_SELECT_MATCH** - station mode in which events whose control integer array match the station's selection integer array in a predefined manner, are selected (see next chapter for details)
26. **ET_STATION_SELECT_USER** - station mode in which events that meet user-defined criteria are selected (see next chapter for details)

8.5.6 Event Restore Modes

27. **ET_STATION_RESTORE_OUT** - mode which restores events in a dead user process back to the ET system by placing them in the station's output list
28. **ET_STATION_RESTORE_IN** - mode which restores events in a dead user process back to the ET system by placing them in the station's input list
29. **ET_STATION_RESTORE_GC** - mode which restores events in a dead user process back to the ET system by placing them in grandcentral station's input list

8.5.7 Default Values

30. **ET_STATION_CUE** - default number of events to cue in a nonblocking station (10)
31. **ET_STATION_PRESCALE** - default prescale value for blocking stations (1) gets every event

8.6 System Related

32. **ET_SYSTEM_EVENTS** - default number of events in a system (300)
33. **ET_SYSTEM_NTEMPS** - default maximum number of temporary events in a system (300)
34. **ET_SYSTEM_ESIZE** - default size of normal events in bytes (1000)
35. **ET_SYSTEM_NSTATS** - default naximum number of stations in a system (10)

8.7 Errors

36. **ET_OK** - error status of ok
37. **ET_ERROR** - error status of error
38. **ET_ERROR_TOOMANY** - error status of too many already
39. **ET_ERROR_EXISTS** - error status of exists already
40. **ET_ERROR_WAKEUP** - error status indicating a user call of et_wakeup_attachments or et_wakeup_all was made in order to wakeup an attachment trying to get events.
41. **ET_ERROR_TIMEOUT** - error status of timeout
42. **ET_ERROR_EMPTY** - error status of no events in station input list
43. **ET_ERROR_BUSY** - error status of another process currently accessing a station's input or output list.

44. **ET_ERROR_DEAD** - error status indicating ET system is dead.
45. **ET_ERROR_READ** - error reading from socket in remote client.
46. **ET_ERROR_WRITE** - error writing to socket in remote client.
47. **ET_ERROR_REMOTE** - error in remote client. Used to differentiate between an error generated locally in the remote client from an error (ET_ERROR) generated and returned by the ET server or a valid value returned from a routine.
48. **ET_ERROR_NOREMOTE** - error indicating that the routine is not supported on a remote client.

8.8 Debug Output Levels

49. **ET_DEBUG_NONE** - no debug output
50. **ET_DEBUG_SEVERE** - severe errors only debug output
51. **ET_DEBUG_ERROR** - all errors debug output
52. **ET_DEBUG_WARN** - all errors and warnings debug output
53. **ET_DEBUG_INFO** - all errors, warnings, and informational debug ouput

8.9 Remote Client Related

54. **ET_SWAP** - swapping of data is necessary since client and server are on different endian machines
55. **ET_NOSWAP** - no swapping of data is necessary since client and server are on the same endian machines
56. **ET_BROADCAST** - discover ET system by broadcasting
57. **ET_MULTICAST** - discover ET system by multicasting
58. **ET_DIRECT** - connect directly (no broad/multicast) to ET system by specifying ET server port number and host
59. **ET_BROADCAST_PORT** - port number for broadcasting (default = 11111)
60. **ET_MULTICAST_PORT** - port number for multicasting (default = 11111)
61. **ET_SERVER_PORT** - port number of the ET system for communicating with remote users (default = 11111)
62. **ET_BROADCAST_ADDR** - broadcasting IP address (default = daq group subnet = "129.57.35.255")
63. **ET_MULTICAST_ADDR** - multicasting IP address (default = " 239.200.0.0")

64. **ET_MULTICAST_TTL -** multicast TTL value (default = 1)
65. **ET_HOST_LOCAL -** ET system's host is local
66. **ET_HOST_REMOTE -** ET system's host is remote
67. **ET_HOST_ANYWHERE -** ET system's host may be local or remote
68. **ET_HOST_AS_LOCAL** - ET client is treated as local if it is local
69. **ET_HOST_AS_REMOTE** - ET client is treated as remote even if it is local

Appendix A User Routines

A.1 General Functions

A.1.1 int et_open

Purpose:

Given an ET system on the same host, this routine will map the system's shared memory into the user's space. It also starts up a thread to produce a heartbeat and a second thread to monitor the ET system's heartbeat. If the ET system is remote, a network connection is made to it.

Arguments:

(et_sys_id *id, char *filename, et_openconfig config)

1. *id* is a pointer that gets filled in with the unique id of the ET system being opened. It can be thought of as a pointer to a handle.
2. *filename* is the unique filename of the ET system
3. *config* is the desired configuration of the way the ET system is opened and is defined by routines starting with " et_open_config_ ..." .

Returns:

1. ET_OK if successful
2. ET_ERROR if failure to open the ET file
3. ET_ERROR_TIMEOUT if the ET system is still not active before the routine returns.
4. ET_ERROR_REMOTE for a remote user if it cannot get the ET system's port number, the host has a strange byteorder, the tcp connection fails, or there's not enough memory.
5. ET_ERROR_READ for a remote user's network read error
6. ET_ERROR_WRITE for a remote user's network write error

Notes:

The ET system is implemented as a single memory mapped file of the name *filename*. This routine should only be called once, before all other ET routines are used, or after a system has been closed with a call to *et_close or et_forcedclose*. A successful return from this routine assures connection to an ET system which is up and running. IT IS CRUCIAL THAT THE USER GET A RETURN VALUE OF " ET_OK" IF THE USER WANTS AN ET SYSTEM GUARANTEED TO FUNCTION.

The user may open an ET system on a remote host. ET decides whether the user is on the same as or a different machine than the system. If the determination is made that the user is on another computer, then network connections are made to that system.

A.1.2 int et_close

Purpose:

Given a local ET system that has been opened with a call to *et_open*, this routine will stop all ET-related threads and unmap the system's memory from the user's space making it inaccessible. It also frees memory allocated in *et_open* to create the system's id. For a remote user, all this routine does is close the connection between the user and ET system as well as free the memory allocated in creating the system's id.

Arguments:

(et_sys_id id)

1. *id* is the id of the ET system being closed.

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_REMOTE for a local user on Linux (or other non-mutex-sharing system) if cannot ummap shared memoryr

Notes:

This routine should only be called once for a particular ET system after the associated call to *et_open*. In addition, all attachments of the process calling this routine must be detached first or an error will be returned.

A.1.3 int et_forcedclose

Purpose:

Given a local ET system that has been opened with a call to *et_open*, this routine will stop all ET-related threads and unmap the system's memory from the user's space making it inaccessible. For a remote user, this routine closes the connection between the user and ET system. But before it does any of this, it detaches all attachments belonging to the process calling it. It also frees memory allocated in *et_open* to create the system's id.

Arguments:

(et_sys_id id)

1. *id* is the id of the ET system being closed.

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_DEAD for a local user if ET system is dead
4. ET_ERROR_REMOTE for a local user on Linux (or other non-mutex-sharing system) if cannot ummap shared memoryr

Notes:

This routine should only be called once for a particular ET system after the associated call to *et_open*.

A.1.4 int et_alive

Purpose:

This routine tells the user whether the ET system is dead or alive.

Arguments:

(et_sys_id id)

1. *id* is the id of the ET system of interest

Returns:

1. 1 if ET system given by *id* is alive
2. 0 if ET system given by *id* is dead
3. ET_ERROR_READ for a remote user's network read error
4. ET_ERROR_WRITE for a remote user's network write error

Notes:

This routine behaves differently depending on whether it is run locally on Solaris, locally on Linux, or remotely. If the user is running it locally on Solaris, a thread of the user's process is constantly checking to see if the ET system is alive and provides a valid return value to *et_alive* when last it was monitored (up to three heartbeats ago). If the user is on a remote node, the ET system's server thread is contacted. If that communication succeeds, then the ET system is alive by definition, otherwise it is dead. If the user is local on Linux, a hybrid approach is used. First the ET system's server thread is contacted. If that fails, the actual heartbeat in the shared memory is monitored (taking 1.5 times the heartbeat time).

A.1.5 int et_wait_for_alive

Purpose:

This routine waits until the ET system has a heartbeat before it returns. It checks once every minimum sleep period.

Arguments:

(et_sys_id id)

1. *id* is the id of the ET system of interest

Returns:

1. ET_OK
2. ET_ERROR_READ for a remote user's network read error
3. ET_ERROR_WRITE for a remote user's network write error

Notes:

This routine behaves differently depending on whether it is run locally or remotely. If the user is running it locally, it constantly checks to see if the ET system is alive and waits before returning until it is. If the user is on a remote node, the ET system's server thread is contacted. If that communication succeeds, then the ET system is alive by definition, otherwise it immediately returns an error.

A.2 Open Configuration Functions

A.2.1 int et_open_config_init

Purpose:

This routine initializes a configuration used by a process to open an ET system. This MUST be done prior to setting any configuration parameters or all setting routines will return an error.

Arguments:

(et_openconfig* sconfig)

1. *sconfig* is pointer to an open configuration variable

Returns:

1. ET_OK if successful
2. ET_ERROR if it fails to allocate memory for configuration data storage

A.2.2 int et_open_config_destroy

Purpose:

This routine frees the memory allocated when a configuration is initialized by " et_open_config_init" .

Arguments:

(et_openconfig sconfig)

1. *sconfig* is an open configuration

Returns:

1. ET_OK

A.2.3 int et_open_config_setwait

Purpose:

This routine sets the means to wait for a valid ET system to be detected.

Arguments:

(et_openconfig sconfig, int val)

1. *sconfig* is an open configuration
2. *val* is the method to wait for a valid ET system to be detected. Setting "val" to ET_OPEN_WAIT makes "et_open" block by waiting until the given ET system is fully functioning or a set time period has passed before returning. Setting val to ET_OPEN_NOWAIT makes "et_open" return immediately after determining whether the ET system is alive or not. The default is ET_OPEN_NOWAIT.

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not ET_OPEN_WAIT or ET_OPEN_NOWAIT

Notes:

If the ET system is local, both ET_OPEN_NOWAIT and ET_OPEN_WAIT mean " et_open"will send a UDP broad/multicast packet to see if the system responds. If not, it tries to detect a heartbeat which necessitates waiting at least one heartbeat. With a remote system, broad/multicasting to find it may take up to several seconds. Usually, if the system is up and running, this will take a fraction of a second. If a direct remote connection is being made, it is tried once in the ET_OPEN_NOWAIT mode, but is tried repeatedly at 10 Hz until the set timeout in the ET_OPEN_WAIT mode.

A.2.4 int et_open_config_getwait

Purpose:

This routine gets the means to wait for a valid ET system to be detected.

Arguments:

(et_openconfig sconfig, int *val)

1. *sconfig* is an open configuration
2. *val* is a pointer that gets filled with ET_OPEN_WAITor ET_OPEN_NOWAIT

Returns:

1. ET_OK

A.2.5 int et_open_config_settimeout

Purpose:

This routine sets the maximum time to wait for a valid ET system to be detected.

Arguments:

(et_openconfig sconfig, struct timespec val)

1. *sconfig* is an open configuration
2. *val* is maximum amount of time to wait for an alive ET system to be detected if the wait mode is ET_OPEN_WAIT. If the time is set to zero (the default), an infinite time is indicated. Note that in local systems, " et_open"waits in integral units of the system's heartbeat time (ET_BEAT_SEC & ET_BEAT_NSEC set in et_private.h). If broad/multicasting to find a remote ET system, it is possible to take up to several seconds to determine whether the system is alive or not. In this case, the time limit may be exceeded.

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized

A.2.6 int et_open_config_gettimeout

Purpose:

This routine gets the maximum time to wait for a valid ET system to be detected.

Arguments:

(et_openconfig sconfig, struct timespec *val)

1. *sconfig* is an open configuration
2. *val* is a pointer that gets filled with the time

Returns:

1. ET_OK

A.2.7 int et_open_config_sethost

Purpose:

This routine sets host or computer on which the ET system is running.

Arguments:

(et_openconfig sconfig, char *val)

1. *sconfig* is an open configuration
2. *val* is the name of the computer (or host) on which the ET system resides. For opening a local system only, set val to ET_HOST_LOCAL (the default) or "localhost" (including quotes). For opening a system on an unknown remote computer only, set it to ET_HOST_REMOTE. For an unknown host which may be local or remote, set it to ET_HOST_ANYWHERE. Otherwise set val to the name or dotted-decimal IP address of the desired host. If the ET_DIRECT option is taken in *et_open_config_setcast*, be aware that this routine must use the ET system's actual host name or "localhost" but must ***NOT*** be ET_HOST_LOCAL, ET_HOST_REMOTE, or ET_HOST_ANYWHERE.

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is NULL or too long

A.2.8 int et_open_config_gethost

Purpose:

This routine gets the host or computer on which the ET system is running.

Arguments:

(et_openconfig sconfig, char *val)

1. *sconfig* is an open configuration
2. *val* is a character array that gets filled with the host name

Returns:

1. ET_OK

A.2.9 int et_open_config_setmode

Purpose:

This routine sets whether the ET system being opened treats the user running this routine as local (if it is local) or remote even if it is local..

Arguments:

(et_openconfig sconfig, int val)

1. *sconfig* is an open configuration
2. *val* can be set to ET_HOST_AS_REMOTE if the local user is to be treated as remote. This means all communication is done through the ET server using sockets. The alternative is to set *val* to ET_HOST_AS_LOCAL (default) which means local users are treated as local with the ET system memory being mapped into the user's space.

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not ET_HOST_AS_REMOTE or ET_HOST_AS_LOCAL

A.2.10 int et_open_config_getmode

Purpose:

This routine gets the mode which tells whether local users are treated as local or remote.

Arguments:

(et_openconfig sconfig, int *val)

1. *sconfig* is an open configuration
2. *val* is a pointer that gets filled with either ET_HOST_AS_REMOTE or ET_HOST_AS_LOCAL

Returns:

1. ET_OK

A.2.11 int et_open_config_setdebugdefault

Purpose:

This routine sets the default level of debugging output.

Arguments:

(et_openconfig sconfig, int val)

1. *sconfig* is an open configuration
2. *val* can be set to ET_DEBUG_NONE which means no output, ET_DEBUG_SEVERE for output describing severe errors, ET_DEBUG_ERROR for output describing all errors, ET_DEBUG_WARN for output describing warnings and errors, and ET_DEBUG_INFO for output describing all information, warnings, and errors.

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not one of the listed values

A.2.12 int et_open_config_getdebugdefault

Purpose:

This routine gets the default level of debugging output.

Arguments:

(et_openconfig sconfig, int *val)

1. *sconfig* is an open configuration
2. *val* is a pointer that gets filled with either ET_DEBUG_NONE, ET_DEBUG_SEVERE, ET_DEBUG_ERROR, ET_DEBUG_WARN, or ET_DEBUG_INFO

Returns:

1. ET_OK

A.2.13 int et_open_config_setcast

Purpose:

This routine sets the method for a remote user to discover the ET system to be opened.

Arguments:

(et_openconfig sconfig, int val)

1. *sconfig* is an open configuration
2. *val* is the name of the method for a remote user to discover the ET system to be opened. Set it to ET_BROADCAST for using UDP broadcasting (the default), ET_MULTICAST for using UDP multicasting, or ET_DIRECT for a direct connection to the ET system by specifying host and port..

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not equal to ET_BROADCAST, ET_MULTICAST, or ET_DIRECT

Notes:

To avoid broad/multicasting to find the ET system (actually the port of the tcp server thread), use the ET_DIRECT option. This does not broad/multicast and connects directly to the ET system. One reason for wanting to avoid this is if the system and user are on different subnets and routers will not pass the UDP packets between them. If using ET_DIRECT, "et_open_config_sethost" must use the ET system's actual host name or "localhost" only. Use " et_open_config_setserverport'to specify the port number of the ET server thread.

A.2.14 int et_open_config_getcast

Purpose:

This routine gets the method for a remote user to discover the ET system to be opened.

Arguments:

(et_openconfig sconfig, int *val)

1. *sconfig* is an open configuration
2. *val* is a pointer that gets filled with either ET_BROADCAST or ET_MULTICAST

Returns:

1. ET_OK

A.2.15 int et_open_config_setaddress

Purpose:

This routine sets the IP subnet address used for broadcasting or the address for multicasting in discovering the ET system to be opened.

Arguments:

(et_openconfig sconfig, char *val)

1. *sconfig* is an open configuration
2. *val* is the IP address for broadcast or multicast communications in dotted-decimal form. If this routine is never called, the address is automatically set to the local subnet broadcast address. However, if finding the subnet broadcast address fails, then the address is set to ET_BROADCAST_ADDR (defined in "et.h" to be the author's subnet broadcast address). If multicasting is used, the address must be explicitly set by this routine.

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is NULL or too long or too short

A.2.16 int et_open_config_getaddress

Purpose:

This routine gets the IP subnet address used for broadcasting or the address for multicasting in discovering the ET system to be opened.

Arguments:

(et_openconfig sconfig, char *val)

1. *sconfig* is an open configuration
2. *val* is a character array that gets filled with the address

Returns:

1. ET_OK

A.2.17 int et_open_config_setport

Purpose:

This routine sets the port number for the IP broadcasting or multicasting used to discover the ET system being opened.

Arguments:

(et_openconfig sconfig, unsigned short val)

1. *sconfig* is an open configuration
2. *val* is the port number of the broadcast or multicast communications. The default is ET_BROADCAST_PORT, defined in "et.h" as 11111. It may also be set to ET_MULTICAST_PORT or to any port number desired by the user.

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is less than 1024

A.2.18 int et_open_config_getport

Purpose:

This routine gets the port number for the IP broadcasting or multicasting used to discover the ET system being opened.

Arguments:

(et_openconfig sconfig, unsigned short *val)

1. *sconfig* is an open configuration
2. *val* is a pointer that gets filled with the port number

Returns:

1. ET_OK

A.2.19 int et_open_config_setserverport

Purpose:

This routine sets the port number for opening an ET system directly without broadcasting or multicasting. It is used in conjunction with setting the value in *et_open_config_setcast* to be ET_DIRECT.

Arguments:

(et_openconfig sconfig, unsigned short val)

1. *sconfig* is an open configuration
2. *val* is the port number of the ET system's tcp server thread. The default value is ET_SERVER_PORT which is set to 11111 in *et.h*.

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is less than 1024

Notes:

To avoid broadcasting or multicasting to find the ET system (actually the port number of the tcp server thread), use *et_open_config_setcast* set to the ET_DIRECT option. This does no broad/multicast and connects directly to the ET system. A possible reason for wanting to avoid this is if the ET system and user are on different subnets and routers will not pass the UDP packets between them. Setting the server port value with this routine assumes that its value is known. One way to ensure this, is to use the routine "et_system_config_setserverport" in the program which is starting up the ET system (e.g. et_start.c). This will definitively set the port number to the requested value or exit the program if it cannot. In this way, the port can be set by the ET system and the information shared with the user for use in this routine.

A.2.20 int et_open_config_getserverport

Purpose:

This routine gets the port number for opening an ET system directly without broadcasting or multicasting.

Arguments:

(et_openconfig sconfig, unsigned short *val)

1. *sconfig* is an open configuration
2. *val* is a pointer that gets filled with the port number

Returns:

1. ET_OK

A.2.21 int et_open_config_setTTL

Purpose:

This routine sets the “ttl” value for multicasting. This value determines how many “ hops” through routers the packet makes.

Arguments:

(et_openconfig sconfig, int val)

1. *sconfig* is an open configuration
2. *val* is the “ ttl”value for multicasting. It defaults to a value of one which restricts multicasting to the local subnet.

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is less than 0 or greater than 254

A.2.22 int et_open_config_getTTL

Purpose:

This routine gets the “ ttl” value for multicasting.

Arguments:

(et_openconfig sconfig, int *val)

1. *sconfig* is an open configuration
2. *val* is a pointer that gets filled with the ttl value

Returns:

1. ET_OK

A.3 System Functions

A.3.1 int et_system_start

Purpose:

This routine creates a new ET system. The process that executes this routine becomes the ET system process.

Arguments:

(et_sys_id *id, et_sysconfig sconfig)

1. *id* is a pointer to an ET system id - a value-result argument - in which is returned the id of the new system which gets created
2. *sconfig* is the desired configuration of the new ET system (set by routines starting with *et_system_config_*).

Returns:

1. ET_OK
2. ET_ERROR if the system could not be created

A.3.2 int et_system_close

Purpose:

This routine closes an ET system that was started with a call to *et_system_start*. All ET system threads are stopped and the shared memory is unmapped. Only the ET system process may execute this routine.

Arguments:

(et_sys_id id)

1. *id* is the id of the ET system

Returns:

1. ET_OK
2. ET_ERROR if not ET system process

A.3.3 int et_system_getlocality

Purpose:

This routine returns a value indicating whether this routine is being executed on the same machine as the ET system or remotely.

Arguments:

(et_sys_id id, int *locality)

1. *id* is the id of the ET system of interest
2. *locality* is a pointer to an integer that gets filled in with the value ET_LOCAL if the user is on the same machine as the ET system, ET_REMOTE if the user is on another machine, and ET_LOCAL_NOSHARE if the user is on the same machine but one whose operating system does not allow the sharing of pthread mutexes (eg. Linux).

Returns:

1. ET_OK
2. ET_ERROR if *locality* is NULL

A.3.4 int et_system_setdebug

Purpose:

This routine sets the debug output of the caller's process for a particular ET system.

Arguments:

(et_sys_id id, int debug)

1. *id* is the id of the ET system of interest
2. *debug* can be either ET_DEBUG_NONE which means no output, ET_DEBUG_SEVERE for output describing severe errors, ET_DEBUG_ERROR for output describing all errors, ET_DEBUG_WARN for output describing warnings and errors, and ET_DEBUG_INFO for output describing all information, warnings, and errors.

Returns:

1. ET_OK
2. ET_ERROR if *debug* is an invalid value

A.3.5 int et_system_getdebug

Purpose:

This routine gets the value of the caller's debug level for a particular ET system.

Arguments:

(et_sys_id id, int *debug)

1. *id* is the id of the ET system of interest
2. *debug* is a pointer to an integer that gets filled in with the debug level of the ET system. See above for possible values.

Returns:

1. ET_OK
2. ET_ERROR if *debug* is NULL

A.3.6 int et_system_getnumevents

Purpose:

This routine tells the total number of events that an ET system has.

Arguments:

(et_sys_id id, int *numevents)

1. *id* is the id of the ET system of interest
2. *numevents* is a pointer to an integer that gets filled in with the total number of events that the ET system has.

Returns:

1. ET_OK
2. ET_ERROR if *numevents* is NULL

A.3.7 int et_system_geteventsize

Purpose:

This routine tells the size in bytes of standard events in an ET system.

Arguments:

(et_sys_id id, int *eventsize)

1. *id* is the id of the ET system of interest
2. *eventsize* is a pointer to an integer that gets filled in with the size in bytes of standard events in the ET system.

Returns:

1. ET_OK
2. ET_ERROR if *eventsize* is NULL

A.3.8 int et_system_gettempsmax

Purpose:

This routine gives the maximum number of temporary events allowed in an ET system.

Arguments:

(et_sys_id id, int *tempsmax)

1. *id* is the id of the ET system of interest
2. *tempsmax* is a pointer to an integer that gets filled in with the maximum number of temporary events allowed in an ET system.

Returns:

1. ET_OK
2. ET_ERROR if *tempsmax* is NULL
3. ET_ERROR_READ for a remote user's network read error
4. ET_ERROR_WRITE for a remote user's network write error

A.3.9 int et_system_getstationsmax

Purpose:

This routine gives the maximum number of stations allowed in an ET system.

Arguments:

(et_sys_id id, int *stationsmax)

1. *id* is the id of the ET system of interest
2. *stationsmax* is a pointer to an integer that gets filled in with the maximum number of stations allowed in an ET system.

Returns:

1. ET_OK
2. ET_ERROR if *stationsmax* is NULL
3. ET_ERROR_READ for a remote user's network read error
4. ET_ERROR_WRITE for a remote user's network write error

A.3.10 int et_system_getprocsmax

Purpose:

This routine gives the maximum number of processes that are allowed to open an ET system.

Arguments:

(et_sys_id id, int *procsmax)

1. *id* is the id of the ET system of interest
2. *procsmax* is a pointer to an integer that gets filled in with the maximum number of processes that are allowed to open an ET system.

Returns:

1. ET_OK
2. ET_ERROR if *procsmax* is NULL
3. ET_ERROR_READ for a remote user's network read error
4. ET_ERROR_WRITE for a remote user's network write error

A.3.11 int et_system_getattsmax

Purpose:

This routine gives the maximum number of attachments that are allowed in an ET system.

Arguments:

(et_sys_id id, int *attsmax)

1. *id* is the id of the ET system of interest
2. *attsmax* is a pointer to an integer that gets filled in with the maximum number of attachments that are allowed in an ET system.

Returns:

1. ET_OK
2. ET_ERROR if *attsmax* is NULL
3. ET_ERROR_READ for a remote user's network read error
4. ET_ERROR_WRITE for a remote user's network write error

A.3.12 int et_system_getheartbeat

Purpose:

This routine gives the current value of the heartbeat of an ET system. For a healthy system, this value changes every heartbeat period.

Arguments:

(et_sys_id id, int *heartbeat)

1. *id* is the id of the ET system of interest
2. *heartbeat* is a pointer to an integer that gets filled in with the current heartbeat of an ET system.

Returns:

1. ET_OK
2. ET_ERROR if *heartbeat* is NULL
3. ET_ERROR_READ for a remote user's network read error
4. ET_ERROR_WRITE for a remote user's network write error

A.3.13 int et_system_getpid

Purpose:

This routine gives the pid of an ET system.

Arguments:

(et_sys_id id, pid_t *pid)

1. *id* is the id of the ET system of interest
2. *pid* is a pointer that gets filled in with the pid of an ET system.

Returns:

1. ET_OK
2. ET_ERROR if *pid* is NULL
3. ET_ERROR_READ for a remote user's network read error
4. ET_ERROR_WRITE for a remote user's network write error

A.3.14 int et_system_getprocs

Purpose:

This routine gives the number of processes that currently have an ET system open.

Arguments:

(et_sys_id id, int *procs)

1. *id* is the id of the ET system of interest
2. *procs* is a pointer to an integer that gets filled in with the number of processes currently open to an ET system.

Returns:

1. ET_OK
2. ET_ERROR if *procs* is NULL
3. ET_ERROR_READ for a remote user's network read error
4. ET_ERROR_WRITE for a remote user's network write error

A.3.15 int et_system_getattachments

Purpose:

This routine gives the number of attachments currently in an ET system.

Arguments:

(et_sys_id id, int *atts)

1. *id* is the id of the ET system of interest
2. *atts* is a pointer to an integer that gets filled in with the number of attachments currently in an ET system.

Returns:

1. ET_OK
2. ET_ERROR if *atts* is NULL
3. ET_ERROR_READ for a remote user's network read error
4. ET_ERROR_WRITE for a remote user's network write error

A.3.16 int et_system_getstations

Purpose:

This routine gives the current number of stations in an ET system that are either idle or active (but not unused or being created).

Arguments:

(et_sys_id id, int *stations)

1. *id* is the id of the ET system of interest
2. *stations* is a pointer that gets filled in with the number of stations in an ET system that are either idle or active.

Returns:

1. ET_OK
2. ET_ERROR if *stations* is NULL
3. ET_ERROR_READ for a remote user's network read error
4. ET_ERROR_WRITE for a remote user's network write error

A.3.17 int et_system_gettemps

Purpose:

This routine gives the number of temporary events currently in an ET system.

Arguments:

(et_sys_id id, int *temps)

1. *id* is the id of the ET system of interest
2. *temps* is a pointer to an integer that gets filled in with the number of temporary events currently in an ET system.

Returns:

1. ET_OK
2. ET_ERROR if *temps* is NULL
3. ET_ERROR_READ for a remote user's network read error
4. ET_ERROR_WRITE for a remote user's network write error

A.3.18 int et_system_gethost

Purpose:

This routine gives the fully qualified name of the ET system's host computer.

Arguments:

(et_sys_id id, char *host)

1. *id* is the id of the ET system of interest
2. *host* is a pointer that gets filled in with the ET system's host name. To be safe the array should be at least ET_MAXHOSTNAMELEN characters long.

Returns:

1. ET_OK
2. ET_ERROR if *host* is NULL

A.3.19 int et_system_getserverport

Purpose:

This routine gives the number of the ET system's TCP server thread.

Arguments:

(et_sys_id id, unsigned short *port)

1. *id* is the id of the ET system of interest
2. *port* is a pointer to an unsigned short integer that gets filled in with the port number of the ET system's TCP server thread.

Returns:

1. ET_OK
2. ET_ERROR if *port* is NULL

A.4 System Configuration Functions

A.4.1 int et_system_config_init

Purpose:

This routine initializes a system configuration. This MUST be done prior to setting any configuration parameters or all setting routines will return an error.

Arguments:

(et_sysconfig* sconfig)

1. *sconfig* is pointer to a system configuration variable

Returns:

1. ET_OK if successful
2. ET_ERROR if it fails to allocate memory for configuration data storage

A.4.2 int et_system_config_destroy

Purpose:

This routine frees the memory allocated when a configuration is initialized by " et_system_config_init" .

Arguments:

(et_sysconfig sconfig)

1. *sconfig* is a system configuration

Returns:

1. ET_OK

A.4.3 int et_system_config_setevents

Purpose:

This routine sets a system configuration's total number of events.

Arguments:

(et_sysconfig sconfig, int val)

1. *sconfig* is a system configuration
2. *val* must be greater than zero

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not an allowed value

A.4.4 int et_system_config_getevents

Purpose:

This routine gets a system configuration's current total number of events.

Arguments:

(et_sysconfig sconfig, int *val)

1. *sconfig* is a system configuration
2. *val* is a pointer that gets filled with the total number of events

Returns:

1. ET_OK

A.4.5 int et_system_config_setsize

Purpose:

This routine sets a system configuration's event size in bytes.

Arguments:

(et_sysconfig sconfig, int val)

1. *sconfig* is a system configuration
2. *val* must be greater than zero

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not an allowed value

A.4.6 int et_system_config_getsize

Purpose:

This routine gets a system configuration's current event size.

Arguments:

(et_sysconfig sconfig, int *val)

1. *sconfig* is a system configuration
2. *val* is a pointer that gets filled with the event size in bytes

Returns:

1. ET_OK

A.4.7 int et_system_config_settemps

Purpose:

This routine sets a system configuration's total number of temporary events.

Arguments:

(et_sysconfig sconfig, int val)

1. *sconfig* is a system configuration
2. *val* must be greater than zero

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not greater than zero

Notes:

The number of temp events must not be greater than the total number of events. This in not checked in this routine, but is checked when attempting to create an ET system.

A.4.8 int et_system_config_gettemps

Purpose:

This routine gets a system configuration's current total number of temporary events.

Arguments:

(et_sysconfig sconfig, int *val)

1. *sconfig* is a system configuration
2. *val* is a pointer that gets filled with the number of temporary events

Returns:

1. ET_OK

A.4.9 int et_system_config_setstations

Purpose:

This routine sets a system configuration's limit on how many stations can be created.

Arguments:

(et_sysconfig sconfig, int val)

1. *sconfig* is a system configuration
2. *val* must be greater than zero

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not greater than zero

A.4.10 int et_system_config_getstations

Purpose:

This routine gets a system configuration's limit on how many stations can be created.

Arguments:

(et_sysconfig sconfig, int *val)

1. *sconfig* is a system configuration
2. *val* is a pointer that gets filled with the maximum number of stations that can be created

Returns:

1. ET_OK

A.4.11 int et_system_config_setprocs

Purpose:

This routine sets a system configuration's limit on how many user processes can open the system.

Arguments:

(et_sysconfig sconfig, int val)

1. *sconfig* is a system configuration
2. *val* must be > 0 and <= ET_PROCESSES_MAX

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not greater than zero or not <= ET_PROCESSES_MAX

Notes:

The default value of ET_PROCESSES_MAX is 20 but may be changed in et_private.h, requiring a recompilation of ET.

A.4.12 int et_system_config_getprocs

Purpose:

This routine gets a system configuration's limit on how many user processes can open the system.

Arguments:

(et_sysconfig sconfig, int *val)

1. *sconfig* is a system configuration
2. *val* is a pointer that gets filled with the configuration's limit on the number of processes that can open the system.

Returns:

1. ET_OK

A.4.13 int et_system_config_setattachments

Purpose:

This routine sets a system configuration's limit on how many user attachments to stations can exist at one time.

Arguments:

(et_sysconfig sconfig, int val)

1. *sconfig* is a system configuration
2. *val* must be > 0 and <= ET_ATTACHMENTS_MAX

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not greater than zero or not <= ET_ATTACHMENTS_MAX

Notes:

The default value of ET_ATTACHMENTS_MAX is 20 but may be changed in et_private.h, requiring a recompile of the ET system library.

A.4.14 int et_system_config_getattachments

Purpose:

This routine gets a system configuration's limit on how many user attachments to stations are possible.

Arguments:

(et_sysconfig sconfig, int *val)

1. *sconfig* is a system configuration
2. *val* is a pointer that gets filled with the system configuration's limit on the number of attachments.

Returns:

1. ET_OK

A.4.15 int et_system_config_setfile

Purpose:

This routine sets a system configuration's ET system file name.

Arguments:

(et_sysconfig sconfig, int *val)

1. *sconfig* is a system configuration
2. *val* is a pointer to a string or null-terminated character array containing the file name

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is NULL, or *val* is too long

A.4.16 int et_system_config_getfile

Purpose:

This routine gets a system configuration's ET system file name

Arguments:

(et_sysconfig sconfig, int *val)

1. *sconfig* is a system configuration
2. *val* is a pointer to a char array that gets filled with the system file name.

Returns:

1. ET_OK

A.4.17 int et_system_config_setcast

Purpose:

This routine sets a system configuration's method for a remote user to discover this ET system.

Arguments:

(et_sysconfig sconfig, int val)

1. *sconfig* is a system configuration
2. *val* is the name of the method for a remote user to discover this ET system. Set it to ET_BROADCAST for using UDP broadcasting (the default), or set it to ET_MULTICAST for using UDP multicasting.

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not equal to ET_BROADCAST or ET_MULTICAST

Notes:

If a user is discovering an ET system directly (with ET_DIRECT), this configuration setting is ignored.

A.4.18 int et_system_config_getcast

Purpose:

This routine gets a system configuration's method for a remote user to discover this ET system.

Arguments:

(et_sysconfig sconfig, int *val)

1. *sconfig* is a system configuration
2. *val* is a pointer that gets filled with either ET_BROADCAST or ET_MULTICAST

Returns:

1. ET_OK

A.4.19 int et_system_config_setaddress

Purpose:

This routine sets an ET system configuration's IP subnet address used for receiving broadcasts or the address for receiving multicasts when remote users discover this ET system.

Arguments:

(et_sysconfig sconfig, char *val)

1. *sconfig* is a system configuration
2. *val* is the IP address for broadcast or multicast communications in dotted-decimal form. If this routine is never called, the address is set to the local subnet address (found automatically).

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is NULL or too long or too short

A.4.20 int et_system_config_getaddress

Purpose:

This routine gets an ET system configuration's IP subnet address used for receiving broadcasts or the address for receiving multicasts when remote users discover this ET system.

Arguments:

(et_sysconfig sconfig, char *val)

1. *sconfig* is a system configuration
2. *val* is a character array that gets filled with the address

Returns:

1. ET_OK

A.4.21 int et_system_config_setport

Purpose:

This routine sets a system configuration's port number for the IP broadcasting or multicasting used to discover this ET system by remote users.

Arguments:

(et_sysconfig sconfig, unsigned short val)

1. *sconfig* is a system configuration
2. *val* is the port number of the broadcast or multicast communications. The default is ET_BROADCAST_PORT, defined in "et.h" as 11111. It may also be set to ET_MULTICAST_PORT or to any port number desired by the user.

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is less than 1024

A.4.22 int et_system_config_getport

Purpose:

This routine gets a system configuration's port number for the IP broadcasting or multicasting used to discover this ET system by remote users.

Arguments:

(et_sysconfig sconfig, unsigned short *val)

1. *sconfig* is a system configuration
2. *val* is a pointer that gets filled with the port number

Returns:

1. ET_OK

A.4.23 int et_system_config_setserverport

Purpose:

This routine sets a system configuration's port number for the ET system (tcp server thread) used to communicate with remote users.

Arguments:

(et_sysconfig sconfig, unsigned short val)

1. *sconfig* is a system configuration
2. *val* is the port number of theET system server for remote users.

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is less than 1024

Notes:

If this routine is called and used to explicitly set the ET system's port, it either uses that port or the ET system process exits with an error. If this routine is **NOT** called, the system tries to use ET_SERVER_PORT (defined in *et.h* as 11111) as its port. If that fails it adds one to that value and tries again and so on until a couple thousand values have been tried at which time the process exits if not successful.

A.4.24 int et_system_config_getserverport

Purpose:

This routine gets a system configuration's port number for the ET system (tcp server thread) used to communicate with remote users.

Arguments:

(et_sysconfig sconfig, unsigned short *val)

1. *sconfig* is a system configuration
2. *val* is a pointer that gets filled with the port number

Returns:

1. ET_OK

A.5 Event Functions

A.5.1 int et_event_new

Purpose:

This routine is used when a process wants a blank or fresh event from the ET system into which it can place data.

Arguments:

(et_sys_id id, et_att_id att, et_event **pe, int wait, struct timespec *time, int size)

1. *id* is the id of the ET system of interest
2. *att* is the attachment id. This is obtained by attaching the user process to a station with *et_station_attach*.
3. *pe* is a pointer to a pointer to an event. Declare it a pointer to an event such as " et_event *pe;" . And pass it as " &pe" .
4. *wait* is either ET_SLEEP, ET_ASYNC, or ET_TIMED. The sleep option waits until an event is available before it returns and therefore may " hang"The timed option returns after a time set by the last argument. Finally, the async option returns immediately whether or not it was successful in obtaining a new event for the caller.
5. *time* is used only with the *wait* = ET_TIMED option, where it gives the time to wait before returning. For other options it will be ignored.
6. *size* is the number of bytes desired for the event's data

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_REMOTE for a memory allocation error of a remote user
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error
6. ET_ERROR_DEAD if ET system is dead
7. ET_ERROR_WAKEUP if told to stop sleeping while trying to get an event
8. ET_ERROR_TIMEOUT if timeout on ET_TIMED option
9. ET_ERROR_BUSY if cannot get access to events due to activity of other processes when in ET_ASYNC mode.
10. ET_ERROR_EMPTY if no events available in ET_ASYNC mode

Notes:

Performance will generally be best with the ET_SLEEP wait mode. It will slow with ET_TIMED, and will crawl with ET_ASYNC. All this routine does for a remote user is allocate memory in which to place event data. The error of ET_ERROR_WAKEUP is returned when the ET system dies, or a user calls *et_wakeup_all* or *et_wakeup_attachment* on the attachmen, whilet waiting to read an event.

A.5.2 int et_events_new

Purpose:

This routine is used when a process wants an array of blank or fresh event from the ET system into which it can place data.

Arguments:

(et_sys_id id, et_att_id att, et_event *pe[], int wait, struct timespec *time, int size, int num, int *nread)

1. *id* is the id of the ET system of interest
2. *att* is the attachment id. This is obtained by attaching the user process to a station with *et_station_attach*.
3. *pe* is an array of pointers to events.
4. *wait* is either ET_SLEEP, ET_ASYNC, or ET_TIMED. The sleep option waits until an event is available before it returns and therefore may " hang"The timed option returns after a time set by the last argument. Finally, the async option returns immediately whether or not it was successful in obtaining a new event for the caller.
5. *time* is used only with the *wait* = ET_TIMED option, where it gives the time to wait before returning. For other options it will be ignored.
6. *size* is the number of bytes desired for the event's data
7. *num* is the number of desired events
8. *nread* returns the number of events actually read

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_REMOTE for a memory allocation error of a remote user
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error
6. ET_ERROR_DEAD if ET system is dead
7. ET_ERROR_WAKEUP if told to stop sleeping while trying to get events
8. ET_ERROR_TIMEOUT if timeout on ET_TIMED option
9. ET_ERROR_BUSY if cannot get access to events due to activity of other processes when in ET_ASYNC mode.
10. ET_ERROR_EMPTY if no events available in ET_ASYNC mode

Notes:

See *et_event_new*. If all processes in an ET system use block transfers such as this one, a speed increase of over 2X the single transfer rate is likely. On Linux a 10 fold increase is possible.

A.5.3 int et_event_get

Purpose:

This routine is used when a consumer wants to read a single event from the ET system.

Arguments:

(et_sys_id id, et_att_id att, et_event **pe, int wait, struct timespec *time)

1. *id* is the id of the ET system of interest
2. *att* is the attachment id.
3. *pe* is a pointer to a pointer to an event. Declare it a pointer to an event such as " et_event *pe;" . And pass it as " &pe" .
4. *wait* is either ET_SLEEP, ET_ASYNC, or ET_TIMED. The sleep option waits until an event is available before it returns and therefore may " hang"The timed option returns after a time set by the last argument. Finally, the async option returns immediately whether or not it was successful in obtaining a new event for the caller. For remote users, the mentioned macros may be ORed with ET_MODIFY. This indicates to the ET server that the user intends to modify the data and so the server must NOT place the event immediately back into the ET system, but must do so only when *et_event_put* is called.
5. *time* is used only with the *wait* = ET_TIMED option, where it gives the time to wait before returning. For other options it will be ignored.

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_REMOTE for a memory allocation error of a remote user
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error
6. ET_ERROR_DEAD if ET system is dead
7. ET_ERROR_WAKEUP if told to stop sleeping while trying to get an event
8. ET_ERROR_TIMEOUT if timeout on ET_TIMED option
9. ET_ERROR_BUSY if cannot get access to events due to activity of other processes when in ET_ASYNC mode.
10. ET_ERROR_EMPTY if no events available in ET_ASYNC mode

Notes:

For remote users, not specifying ET_MODIFY greatly increases ET system efficiency as extra communication between user & system and extra copying of the event data are avoided. The error of ET_ERROR_WAKEUP is returned when the ET system dies, or a user calls *et_wakeup_all* or *et_wakeup_attachment* on the attachment, while waiting to get an event.

A.5.4 int et_events_get

Purpose:

This routine is used when a process wants to read multiple events from the ET system.

Arguments:

(et_sys_id id, et_att_id att, et_event *pe[], int wait, struct timespec *time, int num, int *nread)

1. *id* is the id of the ET system of interest
2. *att* is the attachment id.
3. *pe* is an array of pointers to events.
4. *wait* is either ET_SLEEP, ET_ASYNC, or ET_TIMED. The sleep option waits until an event is available before it returns and therefore may " hang"The timed option returns after a time set by the last argument. Finally, the async option returns immediately whether or not it was successful in obtaining a new event for the caller. For remote users, the mentioned macros may be ORed with ET_MODIFY. This indicates to the ET server that the user intends to modify the data and so the server must NOT place the event immediately back into the ET system, but must do so only when *et_event_put* is called.
5. *time* is used only with the *wait* = ET_TIMED option, where it gives the time to wait before returning. For other options it will be ignored.
6. *num* is the number of events desired to be read.
7. *nread* returns the number of events actually read.

Returns:

1. ET_OK, if successful
2. ET_ERROR if error
3. ET_ERROR_REMOTE for a memory allocation error of a remote user
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error
6. ET_ERROR_DEAD if ET system is dead
7. ET_ERROR_WAKEUP if told to stop sleeping while trying to get events
8. ET_ERROR_TIMEOUT if timeout on ET_TIMED option
9. ET_ERROR_BUSY if cannot get access to events due to activity of other processes when in ET_ASYNC mode.
10. ET_ERROR_EMPTY if no events available in ET_ASYNC mode

Notes:

See *et_event_get*. If all processes in an ET system use block transfers such as this one, a speed increase of over 2X the single transfer rate is possible.

A.5.5 int et_event_put

Purpose:

This routine is used when a process wants to return a single, previously read or new event into the ET system so processes downstream can use it or so it can be returned to grandcentral station.

Arguments:

(et_sys_id id, et_att_id att, et_event *pe)

1. *id* is the id of the ET system of interest
2. *att* is the attachment id.
3. *pe* is a pointer to an event. Declare it as "et_event *pe;" and pass it as "pe".

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

Notes:

Only the attachment (*att*) used to get an event can put that event into the ET system. If a process which did not *get* tries to *put*, an error will be returned. This is implemented in order to prevent a user from accidentally putting many identical events into the system thereby causing data and system corruption.

A.5.6 int et_events_put

Purpose:

This routine is used when a process wants to return multiple, previously read or new events into the ET system so processes downstream can use it or so it can be returned to grandcentral station.

Arguments:

(et_sys_id id, et_att_id att, et_event *pe[], int num)

1. *id* is the id of the ET system of interest
2. *att* is the attachment id.
3. *pe* is an array of pointers to events.
4. *num* is the number of events to be written.

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_REMOTE for a memory allocation error of a remote user
5. ET_ERROR_READ for a remote user's network read error
6. ET_ERROR_WRITE for a remote user's network write error

Notes:

Only the attachment (*att*) used to get events can put those events into the ET system, otherwise an error will be returned. If any one of the events in the array is not owned by *att*, an error will result.

A.5.7 int et_event_dump

Purpose:

This routine is used when a process wants to get rid of a single, previously read or new event so that no user processes downstream will ever see it. It is placed directly into the ET system's grandcentral station which recycles it.

Arguments:

(et_sys_id id, et_att_id att, et_event *pe)

1. *id* is the id of the ET system of interest
2. *att* is the attachment id.
3. *pe* is a pointer to an event. Declare it as "et_event *pe;", and pass it as "pe".

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

A.5.8 int et_events_dump

Purpose:

This routine is used when a process wants to get rid of multiple, previously read or new events so that no user processes downstream will ever see them. They are placed directly into the ET system's grandcentral station which recycles them.

Arguments:

(et_sys_id id, et_att_id att, et_event *pe[], int num)

1. *id* is the id of the ET system of interest
2. *att* is the attachment id.
3. *pe* is an array of pointers to events.
4. *num* is the number of events to be written.

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_REMOTE for a memory allocation error of a remote user
5. ET_ERROR_READ for a remote user's network read error
6. ET_ERROR_WRITE for a remote user's network write error

A.5.9 int et_events_bridge

Purpose:

This routine transfers events between two ET systems. Events are copied from the "from" ET system and placed into the "to" ET system. A function may be provided to swap the data during the transfer.

Arguments:

(et_sys_id id_from, et_sys_id id_to, et_att_id att_from, et_att_id att_to, et_bridgeconfig bconfig, int num, int *ntransferred)

1. *id_from* is the ID of the ET system from which the events are copied
2. *id_to* is the ID of the ET system in which the events are placed
3. *att_from* is the attachment to a station on the "from" ET system
4. *att_to* is the attachment to a station on the "to" ET system (usually GrandCentral)
5. *bconfig* is the configuration of the remaining transfer parameters (see section on Bridge Functions).
6. *num* is the total number of events desired to be transferred
7. *ntransferred* is the total number of events that were actually transferred at the routine's return

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_REMOTE for a memory allocation error of a remote user
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error
6. ET_ERROR_DEAD if ET system is dead
7. ET_ERROR_WAKEUP if told to stop sleeping while trying to get an event
8. ET_ERROR_TIMEOUT if timeout on ET_TIMED option
9. ET_ERROR_BUSY if cannot get access to events due to activity of other processes when in ET_ASYNC mode.
10. ET_ERROR_EMPTY if no events available in ET_ASYNC mode

Notes:

For the best performance, the process calling this routine should be on the same machine as either the "from" or "to" ET systems. Some experimentation is in order to determine which of the two machines

will run the bridging faster. The author's experience suggests that placing the process on the machine with the most processors or computing power will probably give the best results.

A.5.10 int et_event_getdata

Purpose:

This routine provides the pointer to an event's data.

Arguments:

(et_event *pe, void **data)

1. *pe* is a pointer to event.
2. *data* is pointer to a pointer to the event's data

Returns:

1. ET_OK
2. ET_ERROR if error

A.5.11 int et_event_setdatastatus

Purpose:

This routine sets the status to an event's data.

Arguments:

(et_event *pe, int status)

1. *pe* is a pointer to event.
2. *status* may be ET_DATA_OK, ET_DATA_CORRUPT, or ET_DATA_POSSIBLY_CORRUPT. Currently, all data is ET_DATA_OK unless a user's process exits or crashes while having events obtained from the ET system but not put back. In that case, the ET system recovers the events and places them in either GrandCentral station, the attachment's station's input list, or output list depending on the station's configuration (see "et_station_config_setrestore"). If the events are NOT put back into GrandCentral station to be recycled but are placed in the station's input or output list, the data status will become ET_DATA_POSSIBLY_CORRUPT. This simply warns the user that a process previously crashed with the event and may have corrupted its data.

Returns:

1. ET_OK
2. ET_ERROR if error

A.5.12 int et_event_getdatastatus

Purpose:

This routine provides the status of an event's data.

Arguments:

(et_event *pe, int *status)

1. *pe* is a pointer to event.
2. *status* is a pointer which gets filled with the status of an event's data. Ths status can be ET_DATA_OK, ET_DATA_POSSIBLY_CORRUPT, or ET_DATA_CORRUPT.

Returns:

1. ET_OK
2. ET_ERROR if error

A.5.13 int et_event_setlength

Purpose:

This routine records the length of data written into an event.

Arguments:

(et_event *pe, int len)

1. *pe* is a pointer to event.
2. *len* is the length in bytes of data written into the event.

Returns:

1. ET_OK if successful
2. ET_ERROR if error

A.5.14 int et_event_getlength

Purpose:

This routine provides the length of data written into an event.

Arguments:

(et_event *pe, int *len)

1. *pe* is a pointer to event.
2. *len* is a pointer which gets filled with the length in bytes of data written into the event.

Returns:

1. ET_OK
2. ET_ERROR if error

A.5.15 int et_event_setpriority

Purpose:

This routine sets the priority of an event.

Arguments:

(et_event *pe, int pri)

1. *pe* is a pointer to event.
2. *pri* is the priority of either ET_HIGH or ET_LOW.

Returns:

1. ET_OK if successful
2. ET_ERROR if error

A.5.16 int et_event_getpriority

Purpose:

This routine provides the priority of an event.

Arguments:

(et_event *pe, int *pri)

1. *pe* is a pointer to event.
2. *pri* is a pointer which gets filled with the priority of the event.

Returns:

1. ET_OK
2. ET_ERROR if error

A.5.17 int et_event_setcontrol

Purpose:

This routine set the control array of an event.

Arguments:

(et_event *pe, int con[], int num)

1. *pe* is a pointer to event.
2. *con* is an integer array.
3. *num* is the number of elements in the array.

Returns:

1. ET_OK if successful
2. ET_ERROR if error

A.5.18 int et_event_getcontrol

Purpose:

This routine provides the control array of an event.

Arguments:

(et_event *pe, int con[])

1. *pe* is a pointer to event.
2. *con* is an integer array.

Returns:

1. ET_OK if successful
2. ET_ERROR if error

A.5.19 int et_event_setendian

Purpose:

This routine sets the endian value of an event's data. Although an ET system automatically keeps track of the endianness of an event's data, this routine can override and directly set it.

Arguments:

(et_event *pe, int endian)

1. *pe* is a pointer to event.
2. *endian* may be set to ET_ENDIAN_BIG, ET_ENDIAN_LITTLE, ET_ENDIAN_LOCAL (same endian as local host), ET_ENDIAN_NOTLOCAL (opposite endian as local host), or ET_ENDIAN_SWITCH (switch the endian from whatever it is).

Returns:

1. ET_OK if successful
2. ET_ERROR if error

A.5.20 int et_event_getendian

Purpose:

This routine provides the endian value of an event's data.

Arguments:

(et_event *pe, int *endian)

1. *pe* is a pointer to event.
2. *endian* is a pointer which gets filled with the either ET_ENDIAN_BIG or ET_ENDIAN_LITTLE.

Returns:

1. ET_OK
2. ET_ERROR if error

A.5.21 int et_event_needtoswap

Purpose:

This routine indicates whether an event's data needs to be swapped or not.

Arguments:

(et_event *pe, int *swap)

1. *pe* is a pointer to event.
2. *swap* is a pointer which gets filled in with either ET_SWAP or ET_NOSWAP.

Returns:

1. ET_OK if successful
2. ET_ERROR if error

A.5.22 int et_event_CODAswap

Purpose:

This routine swaps the data of an event in CODA format. The data is swapped in the event's data buffer and therefore irreversibly mangles the data. It also takes care of header information concerning the data's endian value so " et_event_setendian"does NOT need to be called.

Arguments:

(et_event *pe)

1. *pe* is a pointer to event.

Returns:

1. ET_OK
2. ET_ERROR if error

A.6 Station Functions

A.6.1 int et_station_create

Purpose:

This routine creates a station provided that it does not already exist and the maximum number of stations do not yet exist. The station's status is set to ET_STATION_IDLE. This changes to ET_STATION_ACTIVE when a process attaches to it. The ET system is immediately notified of the new station upon creation and will transfer events in and out as soon as it is active.

Arguments:

(et_sys_id id, et_stat_id *stat_id, char *stat_name, et_stat_config sconfig)

1. *id* is the id of the ET system of interest
2. *stat_id* is a pointer to a station id number and returns the value of the newly created station's id.
3. *stat_name* is the station name
4. *sconfig* is a station configuration handle

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_EXISTS if station already exists (*stat_id* is set to the existing station's id)
5. ET_ERROR_TOOMANY if the maximum number of stations already exist
6. ET_ERROR_REMOTE for a memory allocation error of a remote user
7. ET_ERROR_READ for a remote user's network read error
8. ET_ERROR_WRITE for a remote user's network write error

A.6.2 int et_station_remove

Purpose:

This routine deletes a station provided it is not grandcentral and provided there are no attached processes. The station's status is set to ET_STATION_UNUSED.

Arguments:

(et_sys_id id, et_stat_id stat_id)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote users network read error
5. ET_ERROR_WRITE for a remote user's network write error

A.6.3 int et_station_attach

Purpose:

This routine will attach the user to a station -- meaning that the user is free to read and write events from that station or to request new events. It returns a unique attachment id in the second argument which is to be used in all transactions with the station.

Arguments:

(et_sys_id id, et_stat_id stat_id, et_att_id *att)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number
3. *att* is a pointer to a attachment id which gets filled in by this routine.

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_REMOTE for memory allocation of a remote user
5. ET_ERROR_READ for a remote user's network read error
6. ET_ERROR_WRITE for a remote user's network write error

Notes:

When a user process attaches to a station, it is marked as an active station, which means it will start receiving events. To remove an attachment, call the routine " et_station_detach" .

A.6.4 int et_station_detach

Purpose:

This routine will detach a user attachment from a station meaning that the process can no longer read or write events from that station. It undoes what *et_station_attach* does.

Arguments:

(et_sys_id id, et_att_id att)

1. *id* is the id of the ET system of interest
2. *att* is the attachment id.

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

Notes:

If this routine detaches the last attachment to a station, it marks the station as idle. In other words, the station stops receiving events since no one is there to read them. All events remaining in the station's input list (after the detachment) will be moved to the output list and sent to other stations. One must detach all attachments to a station before the station can be removed.

A.6.5 void et_wakeup_attachment

Purpose:

This routine will wake up a single attachment that is blocked, waiting to read events from a station. If the user is remote, nothing is done.

Arguments:

(et_sys_id id, et_att_id att)

1. *id* is the id of the ET system of interest
2. *att* is the attachment id.

Returns:

1. ET_OK if successful
2. ET_ERROR if bad *att* argument
3. ET_ERROR_WRITE if a remote user's network write error

A.6.6 void et_wakeup_all

Purpose:

This routine will wake up all attachments that are blocked, waiting to read events from a station. If the user is remote, nothing is done.

Arguments:

(et_sys_id id, et_stat_id stat_id)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.

Returns:

1. ET_OK if successful
2. ET_ERROR if bad *stat_id* argument
3. ET_ERROR_WRITE if a remote user's network write error

A.6.7 int et_station_isattached

Purpose:

Is *att* attached to station *stat_id*?

Arguments:

(et_sys_id id, et_stat_id stat_id, et_att_id att)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *att* is the attachment id.

Returns:

1. 1 if attached
2. 0 if not attached
3. ET_ERROR for bad argument(s)
4. ET_ERROR_DEAD if ET system is dead
5. ET_ERROR_READ for a remote user's network read error
6. ET_ERROR_WRITE for a remote user's network write error

A.6.8 int et_station_exists

Purpose:

Given the name of a station, this routine tells whether the station exists or not. If it does, it gives a its id.

Arguments:

(et_sys_id id, et_stat_id *stat_id, char *stat_name)

1. *id* is the id of the ET system of interest
2. *stat_id* is a pointer to a station id which is filled if *stat_name* exists.
3. *stat_name* is the station name

Returns:

1. 1 if station exists
2. 0 if station does not exist
3. ET_ERROR for bad *stat_name* argument
4. ET_ERROR_DEAD if ET system is dead.
5. ET_ERROR_REMOTE for a memory allocation error of a remote user
6. ET_ERROR_READ for a remote user's network read error
7. ET_ERROR_WRITE for a remote user's network write error

A.6.9 int et_station_name_to_id

Purpose:

Given the name of a station, this routine will return its id.

Arguments:

(et_sys_id id, et_stat_id *stat_id, char *name)

1. *id* is the ET system id
2. *stat_id* is a pointer that get filled in with the station's id number.
3. *name* is the station name

Returns:

1. ET_OK
2. ET_ERROR if no station by that name exists or *stat_name* is null
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_REMOTE for a memory allocation error of a remote user
5. ET_ERROR_READ for a remote user's network read error
6. ET_ERROR_WRITE for a remote user's network write error

A.6.10 int et_station_getattachments

Purpose:

Gives the number of attachments to a station. This returns an error for unused stations.

Arguments:

(et_sys_id id, et_stat_id stat_id, int *numatts)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *numatts* is a pointer to int which gets filled in with the number of attachments

Returns:

1. ET_OK if successful
2. ET_ERROR if station is unused or bad *stat_id* argument
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

A.6.11 int et_station_getstatus

Purpose:

Return a station's status.

Arguments:

(et_sys_id id, et_stat_id stat_id, int *status)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *status* is a pointer which gets filled in with the status of the station

Returns:

1. ET_OK if successful
2. ET_ERROR for bad *stat_id* argument
3. ET_ERROR_DEAD if ET system is dead.
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

A.6.12 int et_station_getinputcount

Purpose:

Gives the number of events in a station's input list. This number changes rapidly and is likely to be out-of-date immediately.

Arguments:

(et_sys_id id, et_stat_id stat_id, int *cnt)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *cnt* is a pointer to int which gets filled in with the number of events in the station's input list

Returns:

1. ET_OK if successful
2. ET_ERROR for bad *stat_id* argument
3. ET_ERROR_DEAD if ET system is dead.
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

A.6.13 int et_station_getoutputcount

Purpose:

Gives the number of events in a station's output list. This number changes rapidly and is likely to be out-of-date immediately.

Arguments:

(et_sys_id id, et_stat_id stat_id, int *cnt)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *cnt* is a pointer which gets filled in with the number of events in the station's output list

Returns:

1. ET_OK if successful
2. ET_ERROR for bad *stat_id* argument
3. ET_ERROR_DEAD if ET system is dead.
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

A.6.14 int et_station_getblock

Purpose:

Gets the value of a station's blocking mode.

Arguments:

(et_sys_id id, et_stat_id stat_id, int *block)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *block* is a pointer which gets filled in with the blocking mode

Returns:

1. ET_OK if successful
2. ET_ERROR if station is unused or bad *stat_id* argument
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

A.6.15 int et_station_getuser

Purpose:

Gets the value of a station's user mode.

Arguments:

(et_sys_id id, et_stat_id stat_id, int *user)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *user* is a pointer which gets filled in with the user mode

Returns:

1. ET_OK if successful
2. ET_ERROR if station is unused or bad *stat_id* argument
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

A.6.16 int et_station_getrestore

Purpose:

Gets the value of a station's restore mode.

Arguments:

(et_sys_id id, et_stat_id stat_id, int *restore)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *restore* is a pointer which gets filled in with the restore mode

Returns:

1. ET_OK if successful
2. ET_ERROR if station is unused or bad *stat_id* argument
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

A.6.17 int et_station_getselect

Purpose:

Gets the value of a station's select mode.

Arguments:

(et_sys_id id, et_stat_id stat_id, int *select)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *select* is a pointer which gets filled in with the select mode

Returns:

1. ET_OK if successful
2. ET_ERROR if station is unused or bad *stat_id* argument
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

A.6.18 int et_station_getcue

Purpose:

Gets a station's cue value.

Arguments:

(et_sys_id id, et_stat_id stat_id, int *cue)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *cue* is a pointer which gets filled in with the cue value

Returns:

1. ET_OK if successful
2. ET_ERROR if station is unused or bad *stat_id* argument
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

A.6.19 int et_station_getprescale

Purpose:

Gets a station's prescale value.

Arguments:

(et_sys_id id, et_stat_id stat_id, int *prescale)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *prescale* is a pointer which gets filled in with the prescale value

Returns:

1. ET_OK if successful
2. ET_ERROR if station is unused or bad *stat_id* argument
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

A.6.20 int et_station_getlib

Purpose:

Gets a station's shared library name.

Arguments:

(et_sys_id id, et_stat_id stat_id, int *lib)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *lib* is a character array which gets filled in with the lib name

Returns:

1. ET_OK if successful
2. ET_ERROR if station is unused or bad *stat_id* argument
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

Notes:

A station has a shared library name associated with it only in the ET_STATION_SELECT_USER select mode in which the user supplies his own function to select events.

A.6.21 int et_station_getfunction

Purpose:

Gets a station's function name.

Arguments:

(et_sys_id id, et_stat_id stat_id, int *function)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *function* is a character array which gets filled in with the function name

Returns:

1. ET_OK if successful
2. ET_ERROR if station is unused or bad *stat_id* argument
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

Notes:

A station has a function name associated with it only in the ET_STATION_SELECT_USER select mode in which the user supplies his own function to select events.

A.6.22 int et_station_getselectwords

Purpose:

Gets a station's array of selection integers (words) used to select events.

Arguments:

(et_sys_id id, et_stat_id stat_id, int *select)

1. *id* is the id of the ET system of interest
2. *stat_id* is the station's id number.
3. *select* is an integer array which gets filled in with the station's selection array

Returns:

1. ET_OK if successful
2. ET_ERROR if station is unused or bad *stat_id* argument
3. ET_ERROR_DEAD if ET system is dead
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error

A.7 Station Configuration Functions

A.7.1 int et_station_config_init

Purpose:

This routine initializes a station configuration. This MUST be done prior to setting any configuration parameters or all setting routines will return an error.

Arguments:

(et_statconfig* sconfig)

1. *sconfig* is pointer to a station configuration variable

Returns:

1. ET_OK if successful
2. ET_ERROR if it fails to allocate memory for configuration data storage

A.7.2 int et_station_config_destroy

Purpose:

This routine frees the memory allocated when a configuration is initialized by " et_station_config_init" .

Arguments:

(et_statconfig sconfig)

1. *sconfig* is a station configuration

Returns:

1. ET_OK

A.7.3 int et_station_config_setblock

Purpose:

This routine sets a station configuration's block mode.

Arguments:

(et_statconfig sconfig, int val)

1. *sconfig* is a station configuration
2. *val* must be either ET_STATION_BLOCKING or ET_STATION_NONBLOCKING

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not one of the allowed values

A.7.4 int et_station_config_getblock

Purpose:

This routine gets a station configuration's current block mode value.

Arguments:

(et_statconfig sconfig, int *val)

1. *sconfig* is a station configuration
2. *val* is a pointer that gets filled with the current value of the configuration block mode

Returns:

1. ET_OK

A.7.5 int et_station_config_setselect

Purpose:

This routine sets a station configuration's select mode.

Arguments:

(et_statconfig sconfig, int val)

1. *sconfig* is a station configuration
2. *val* must be either ET_STATION_SELECT_ALL or ET_STATION_SELECT_MATCH, or ET_STATION_SELECT_USER

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not one of the allowed values

A.7.6 int et_station_config_getselect

Purpose:

This routine gets a station configuration's current select mode value.

Arguments:

(et_statconfig sconfig, int *val)

1. *sconfig* is a station configuration
2. *val* is a pointer that gets filled with the current value of the configuration select mode

Returns:

1. ET_OK

A.7.7 int et_station_config_setuser

Purpose:

This routine sets a station configuration's user mode.

Arguments:

(et_statconfig sconfig, int val)

1. *sconfig* is a station configuration
2. *val* must be either ET_STATION_USER_MULTI or ET_STATION_USER_SINGLE

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not one of the allowed values

A.7.8 int et_station_config_getuser

Purpose:

This routine gets a station configuration's current user mode value.

Arguments:

(et_statconfig sconfig, int *val)

1. *sconfig* is a station configuration
2. *val* is a pointer that gets filled with the current value of the configuration user mode

Returns:

1. ET_OK

A.7.9 int et_station_config_setrestore

Purpose:

This routine sets a station configuration's restore mode.

Arguments:

(et_statconfig sconfig, int val)

1. *sconfig* is a station configuration
2. *val* must be either ET_STATION_RESTORE_OUT or ET_STATION_RESTORE_IN, or ET_STATION_RESTORE_GC

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not one of the allowed values

A.7.10 int et_station_config_getrestore

Purpose:

This routine gets a station configuration's current restore mode value.

Arguments:

(et_statconfig sconfig, int *val)

1. *sconfig* is a station configuration
2. *val* is a pointer that gets filled with the current value of the configuration restore mode

Returns:

1. ET_OK

A.7.11 int et_station_config_setcue

Purpose:

This routine sets the size of a station configuration's input list cue when the block mode is ET_STATION_NONBLOCKING.

Arguments:

(et_statconfig sconfig, int val)

1. *sconfig* is a station configuration
2. *val* must be greater than zero

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not an allowed value

Notes:

The value of the cue must not exceed the total number of events in the system. This is not checked until a station is created with this configuration - at which time cue is set to the total number of events in the system if it is set too high in this routine. **Take notice that setting the value of the cue to the total # of events will, in essence, change the station into one which blocks (block mode of ET_STATION_BLOCKING).** The reason is that all events will now pass through this station.

A.7.12 int et_station_config_getcue

Purpose:

This routine gets a station configuration's current cue value.

Arguments:

(et_statconfig sconfig, int *val)

1. *sconfig* is a station configuration
2. *val* is a pointer that gets filled with the current cue value

Returns:

1. ET_OK

A.7.13 int et_station_config_setprescale

Purpose:

This routine sets the station configuration's prescale when the block mode is ET_STATION_BLOCKING.

Arguments:

(et_statconfig sconfig, int val)

1. *sconfig* is a station configuration
2. *val* must be greater than zero

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized or *val* is not an allowed value

A.7.14 int et_station_config_getprescale

Purpose:

This routine gets a station configuration's current prescale value.

Arguments:

(et_statconfig sconfig, int *val)

1. *sconfig* is a station configuration
2. *val* is a pointer that gets filled with the current prescale value

Returns:

1. ET_OK

A.7.15 int et_station_config_setselectwords

Purpose:

This routine sets the values of integers in the station configuration's array used to select events.

Arguments:

(et_statconfig sconfig, int val[])

1. *sconfig* is a station configuration
2. *val* is an array of integers

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized

A.7.16 int et_station_config_getselectwords

Purpose:

This routine gets a station configuration's current select values.

Arguments:

(et_statconfig sconfig, int val[])

1. *sconfig* is a station configuration
2. *val* is an array that gets filled with the current select values

Returns:

1. ET_OK

A.7.17 int et_station_config_setlib

Purpose:

This routine sets the value of a station configuration's shared library name - used for loading a user's function to select events.

Arguments:

(et_statconfig sconfig, int *val)

1. *sconfig* is a station configuration
2. *val* is a pointer to a string or null-terminated character array containing the shared library name

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized, or *val* is NULL, or *val* is too long

A.7.18 int et_station_config_getlib

Purpose:

This routine gets a station configuration's current shared library name.

Arguments:

(et_statconfig sconfig, int *val)

1. *sconfig* is a station configuration
2. *val* is a pointer to a character array that gets filled with the current shared library name

Returns:

1. ET_OK

A.7.19 int et_station_config_setfunction

Purpose:

This routine sets the value of a station configuration's function name - loaded from the shared library and used for allowing the user to select events.

Arguments:

(et_statconfig sconfig, int *val)

1. *sconfig* is a station configuration
2. *val* is a pointer to a string or null-terminated character array containing the function name

Returns:

1. ET_OK
2. ET_ERROR if the *sconfig* was not initialized, or *val* is NULL, or *val* is too long

A.7.20 int et_station_config_getfunction

Purpose:

This routine gets a station configuration's current function name.

Arguments:

(et_statconfig sconfig, int *val)

1. *sconfig* is a station configuration
2. *val* is a pointer to a character array that gets filled with the current function's name

Returns:

1. ET_OK

A.8 Bridge Functions

A.8.1 int et_events_bridge

Purpose:

This routine transfers events between two ET systems. Events are copied from the "from" ET system and placed into the "to" ET system. A function may be provided to swap the data during the transfer.

Arguments:

(et_sys_id id_from, et_sys_id id_to, et_att_id att_from, et_att_id att_to, et_bridgeconfig bconfig, int num, int *ntransferred)

1. *id_from* is the ID of the ET system from which the events are copied
2. *id_to* is the ID of the ET system in which the events are placed
3. *att_from* is the attachment to a station on the "from" ET system
4. *att_to* is the attachment to a station on the "to" ET system (usually GrandCentral)
5. *bconfig* is the configuration of the remaining transfer parameters
6. *num* is the total number of events desired to be transferred
7. *ntransferred* is the total number of events that were actually transferred at the routine's return

Returns:

1. ET_OK if successful
2. ET_ERROR if error
3. ET_ERROR_REMOTE for a memory allocation error of a remote user
4. ET_ERROR_READ for a remote user's network read error
5. ET_ERROR_WRITE for a remote user's network write error
6. ET_ERROR_DEAD if ET system is dead
7. ET_ERROR_WAKEUP if told to stop sleeping while trying to get an event
8. ET_ERROR_TIMEOUT if timeout on ET_TIMED option
9. ET_ERROR_BUSY if cannot get access to events due to activity of other processes when in ET_ASYNC mode.
10. ET_ERROR_EMPTY if no events available in ET_ASYNC mode

Notes:

For the best performance, the process calling this routine should be on the same machine as either the "from" or "to" ET systems. Some experimentation is in order to determine which of the two machines will run the bridging faster. The author's experience suggests that placing the process on the machine with the most processors or computing power will probably give the best results.

A.8.2 int et_bridge_config_init

Purpose:

This routine initializes a configuration used by the routine " et_events_bridge"in transferring events between two ET systems. This MUST be done prior to setting any configuration parameters or all setting routines will return an error.

Arguments:

(et_bridgeconfig *config)

1. *config* is pointer to a bridge configuration

Returns:

1. ET_OK if successful
2. ET_ERROR if it fails to allocate memory for configuration data storage

A.8.3 int et_bridge_config_destroy

Purpose:

This routine frees the memory allocated when a configuration is initialized by " et_bridge_config_init" .

Arguments:

(et_bridgeconfig config)

1. *config* is a bridge configuration

Returns:

1. ET_OK

A.8.4 int et_bridge_config_setmodefrom

Purpose:

This routine sets the mode of getting events from the "from" ET system.

Arguments:

(et_bridgeconfig config, int val)

1. *config* is a bridge configuration
2. *val* is et to either ET_SLEEP, ET_TIMED, or ET_ASYNC and determines the mode of getting events from the "from" ET system. The default is ET_SLEEP.

Returns:

1. ET_OK
2. ET_ERROR if *config* was not initialized or *val* is not ET_SLEEP, ET_TIMED, or ET_ASYNC

A.8.5 int et_bridge_config_getmodefrom

Purpose:

This routine gets the mode of getting events from the "from" ET system.

Arguments:

(et_bridgeconfig config, int *val)

1. *config* is a bridge configuration
2. *val* is a pointer that gets filled with ET_SLEEP, ET_TIMED, or ET_ASYNC

Returns:

1. ET_OK
2. ET_ERROR

A.8.6 int et_bridge_config_setmodeto

Purpose:

This routine sets the mode of getting nre events from the "to" ET system.

Arguments:

(et_bridgeconfig config, int val)

1. *config* is a bridge configuration
2. *val* is et to either ET_SLEEP, ET_TIMED, or ET_ASYNC and determines the mode of getting new events from the "to" ET system. The default is ET_SLEEP.

Returns:

1. ET_OK
2. ET_ERROR if *config* was not initialized or *val* is not ET_SLEEP, ET_TIMED, or ET_ASYNC

A.8.7 int et_bridge_config_getmodeto

Purpose:

This routine gets the mode of getting new events from the "to" ET system.

Arguments:

(et_bridgeconfig config, int *val)

1. *config* is a bridge configuration
2. *val* is a pointer that gets filled with ET_SLEEP, ET_TIMED, or ET_ASYNC

Returns:

1. ET_OK
2. ET_ERROR

A.8.8 int et_bridge_config_setchunkfrom

Purpose:

This routine sets the maximum number of events to get from the "from" ET system in a single call to “ et_events_get”- the number of events to get in one chunk.

Arguments:

(et_bridgeconfig config, int val)

1. *config* is a bridge configuration
2. *val* is any integer greater than one. The default is 100.

Returns:

1. ET_OK
2. ET_ERROR if *config* was not initialized or *val* less than one.

A.8.9 int et_bridge_config_getchunkfrom

Purpose:

This routine gets the maximum number of events to get from the "from" ET system in a single call to “ et_events_get” .

Arguments:

(et_bridgeconfig config, int *val)

1. *config* is a bridge configuration
2. *val* is a pointer that gets filled with the number

Returns:

1. ET_OK
2. ET_ERROR

A.8.10 int et_bridge_config_setchunkto

Purpose:

This routine sets the maximum number of new events to get from the "to" ET system in a single call to “et_events_new” - the number of events to get in one chunk.

Arguments:

(et_bridgeconfig config, int val)

1. *config* is a bridge configuration
2. *val* is any integer greater than one. The default is 100.

Returns:

1. ET_OK
2. ET_ERROR if *config* was not initialized or *val* less than one.

A.8.11 int et_bridge_config_getchunkto

Purpose:

This routine gets the maximum number of new events to get from the "to" ET system in a single call to “ et_events_new” .

Arguments:

(et_bridgeconfig config, int *val)

1. *config* is a bridge configuration
2. *val* is a pointer that gets filled with the number

Returns:

1. ET_OK
2. ET_ERROR

A.8.12 int et_bridge_config_settimeoutfrom

Purpose:

This routine sets the time to wait for the "from" ET system during all “ et_events_get” calls when the mode is set to ET_TIMED.

Arguments:

(et_bridgeconfig config, int val)

1. *config* is a bridge configuration
2. *val* is the time to wait. The default is 0 seconds.

Returns:

1. ET_OK
2. ET_ERROR if *config* was not initialized.

A.8.13 int et_bridge_config_gettimeoutfrom

Purpose:

This routine gets the time to wait for the "from" ET system during all “ et_events_get” calls when the mode is set to ET_TIMED.

Arguments:

(et_bridgeconfig config, int *val)

1. *config* is a bridge configuration
2. *val* is a pointer that gets filled with the time

Returns:

1. ET_OK
2. ET_ERROR

A.8.14 int et_bridge_config_settimeoutto

Purpose:

This routine sets the time to wait for the "to" ET system during all “ et_events_new” calls when the mode is set to ET_TIMED.

Arguments:

(et_bridgeconfig config, int val)

1. *config* is a bridge configuration
2. *val* is the time to wait. The default is 0 seconds.

Returns:

1. ET_OK
2. ET_ERROR if *config* was not initialized.

A.8.15 int et_bridge_config_gettimeoutto

Purpose:

This routine gets the time to wait for the "to" ET system during all “ et_events_new” calls when the mode is set to ET_TIMED.

Arguments:

(et_bridgeconfig config, int *val)

1. *config* is a bridge configuration
2. *val* is a pointer that gets filled with the time

Returns:

1. ET_OK
2. ET_ERROR

A.8.16 int et_bridge_config_setfunc

Purpose:

This routine sets the function used to automatically swap data from one endian to another when bridging events between two ET systems.

Arguments:

(et_bridgeconfig config, ET_SWAP_FUNCPTR func)

1. *config* is a bridge configuration
2. *func* is the name of the function or function pointer. The default is NULL (no swapping).

Returns:

1. ET_OK
2. ET_ERROR if *config* was not initialized.

Notes:

The function must be of the form: **int func(et_event *src, et_event *dest, int bytes, int same_endian)**. It returns ET_OK if successful otherwise ET_ERROR. The arguments consists of: *src* which is a pointer to the event whose data is to be swapped, *dest* which is a pointer to the event where the swapped data goes, *bytes* which tells the length of the data in bytes, and *same_endian* which is a flag equalling one if the machine and the data are of the same endian and zero otherwise. This function must be able to work with *src* and *dest* being the same event. With this as a prototype, the user can write a routine which swaps data in the appropriate manner. Notice that the first two arguments are pointers to events and not data buffers. This allows the writer of such a routine to have access to any of the event's header information. In general, such functions should NOT call " et_event_setendian"in order to change the registered endian value of the data. This is already taken care of in " et_events_bridge" .

A.8.17 int et_bridge_CODAswap

Purpose:

This function can be used as an argument in the routine et_bridge_config_setfunc to provide automatic swapping of CODA format data when bridging events between two ET systems.

Arguments:

(et_event *src, et_event *dest, int bytes, int same_endian)

1. *src* is a pointer to an event whose data is to be swapped
2. *dest* is a pointer to an event where the swapped data goes
3. *bytes* is the length of the data in bytes
4. *same_endian* is a flag equalling one if the machine and the data are of the same endian and zero otherwise

Returns:

1. ET_OK