HP OpenView
Event Correlation Services
Developer's Guide and Reference

For HP-UX, Solaris, Linux, Windows® 2000, Windows® XP, and Windows® 2003 operating systems

Manufacturing Part Number: T2490-90015

May 2004

Legal Notices

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett- Packard product and replacement parts can be obtained from your local Sales and Service Office.

HEWLETT-PACKARD COMPANY

3404 E. Harmony Road

Fort Collins, CO 80528 U.S.A.

Use of this manual and flexible disk(s), tape cartridge(s), or CD-ROM(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Trademark Notices

Contents

Contents

Contents

Contents

Contact Information

Contacts

Please visit our HP OpenView web site at:

`http://openview.hp.com/`

There you will find contact information as well as details about the products and services HP OpenView has to offer.

Support

The “hp OpenView support” area of the HP OpenView web site includes:

- Downloadable documentation
- Troubleshooting information
- Patches and updates
- Problem reporting
- Training Information
- Support program information

1 Introduction

Purpose

The HP OV ECS Developer's Guide contains the information you require to use the HP OpenView Event Correlation Services Event Input/Output and Annotation Application Programming Interfaces (APIs). This guide contains information on:

- The connection architecture.
- Developing event input and output processes.
- Developing an annotate server process.

NOTE Ignore all references to the ECS Designer for the rest of this document. The ECS Designer is sold as a separate product and must be ordered separately.

Audience

This manual is written for network application developers who will design and build event input, output and annotate server processes, or add these services to existing applications. Readers of this document are assumed to have the following background:

- A detailed understanding of the event types generated within the network.
- For ASCII events, a general knowledge of Message Description Language (MDL).
- For SNMP and CMIP events, a knowledge of the HP OpenView Distributed Management platform and how it manages MIBs.
- Familiarity with the C programming language and application development techniques
- A general understanding of ECS circuit design and use of the ECS Designer.

2 Introduction

This chapter introduces the HP OV ECS Event I/O and Annotation APIs and describes their high-level architecture.This chapter has the following sections.

Example application code (in both source and compiled form) is provided to illustrate use of the APIs. The same example code is used in this guide.

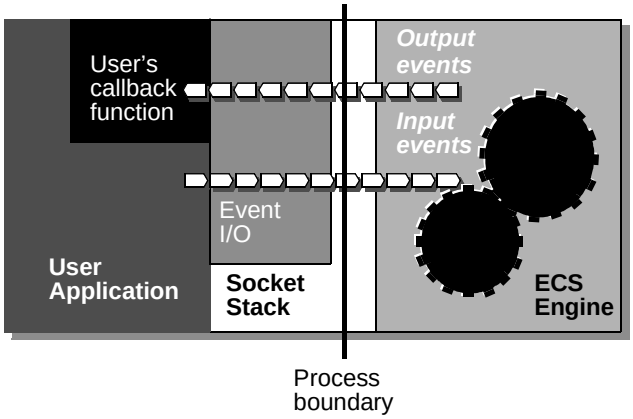
NOTE The HP OV ECS APIs described in this guide (ESOK, EIO, ANNO,EDI and EV) are not thread-safe and are designed to be used in single-threaded applications only.

Integrating ECS With Your Application

The ECS event I/O API enables you to integrate the high-speed, real-time event correlation mechanism provided by the HP OV ECS engine into a new or existing system.

The Event I/O API allows the input and output of arbitrary length event PDUs (Protocol Data Units) between your application(s) and an ECS Engine running as a separate process on the same machine. Your application can send events into an engine, or receive events from an engine, or both. As illustrated in Figure 2-1, the event I/O API is the glue that binds your application process with the ECS Engine process.

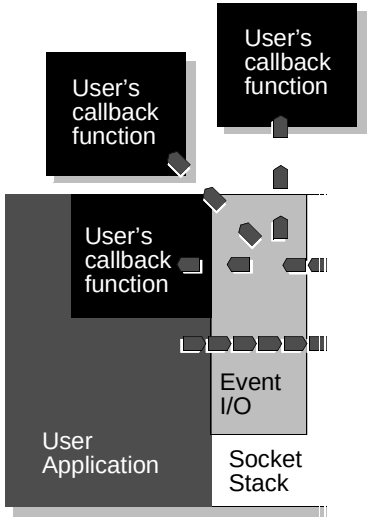
Figure 2-1 ECS Integration Architecture



The callback function shown in Figure 2-1 is registered with the Event I/O API by your application. It can be registered to receive all the output events (as shown here) or a subset of the output events (as illustrated in Figure 2-2). Event subsets can be selected by either event encoding type (for example, all "mdl" events) or event encoding type plus event syntax (for example, a specific ASCII[1]/TL1 message format).

1. CMIP, ASCII and X733 endecoders are no longer supported. Ignore all such references.

Figure 2-2 Callback Functions Register for an Event Subset



The callback functions can be in the same process or different processes. Callback functions in the same process can share a common socket stack connection to the ECS Engine.

One of the callback functions can be registered as a default callback. If an event does not match the registration conditions established for any other callback then the default callback receives it.

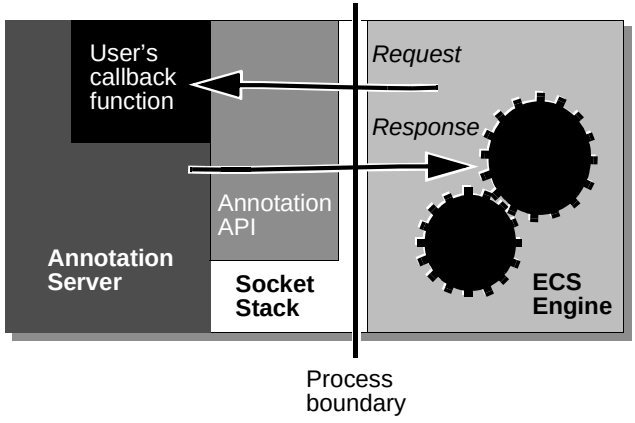
Event Annotation

Annotation is a method by which an ECS circuit can perform actions and obtain information from outside the ECS Engine and its associated Data and Fact Stores. For example:

- An alarm management circuit may need to query an external server for retrieving the name of the person responsible for maintaining a faulty node.
- A fault management application may need to gather additional information from the network before suppressing a fault report.

An annotation server is a user-supplied process external to the ECS Engine that receives annotation requests and sends back annotation responses. The annotation API allows you to link one or more annotation servers directly to one or more ECS Engines to exchange annotation requests and responses.

Figure 2-3 Annotation Architecture



Compare the annotation architecture illustrated in Figure 2-3 with the event I/O architecture shown in Figure 2-1. Both architectures rely on the Socket Stack to transport data between the ECS Engine process and the user's application process. In the same way that the event I/O can register a callback for specific events, the annotation server can register

its callback to receive specific annotation requests, based on the name of the circuit and the name of the node. Alternatively, an annotation server can register for *all* annotation requests emitted from an engine.

Request and response data is easily manipulated using the supplied API functions. However, it is essential that the circuit designer and annotation server developer agree on the number and types of request and response data components.

Interprocess Communication

Underlying both the event I/O and the annotation APIs is the socket stack. In Figure 2-1 and Figure 2-3 you can see that the socket stack is partly exposed to the user application (or annotation server) to provide a common, flexible interprocess communication mechanism.

Basically, there are three things you need to do using the socket stack API:

- open a socket stack connection
- regularly call the socket stack process function `ESOK_process()`
- close the socket stack connection.

All other interaction with the ECS Engine (between opening and closing of the socket stack) is through the event I/O and/or annotate API, as appropriate.

The need to call `ESOK_process()` arises because the design of the socket stack does not rely on threads or any other form of concurrent processing. You must, therefore, explicity provide the stack with processing time by calling the socket stack API function `ESOK_process()`.

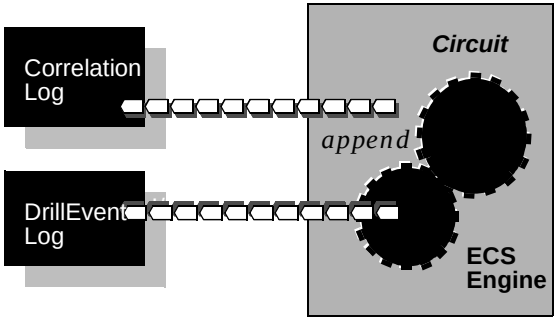
Drill Down

A correlated event[1] may have a number of correlating events[2] or none of them. A correlating event can also be a correlating event for some other correlated events. This implies that an event can be both a correlating event as well as correlated event at the same time.

Any application receiving a correlated event from ECS may be intrested in knowing all the correlating events for the correlated event. This is called drilling down. Conversely, an application maybe interested in finding out all the correlated events for a given correlating event. This is called drilling up.

In order to do a drilldown or a drillup on a given event , the correlation relationship related to the event need to be captured. ECS provides ECDL function append for specifying relationship between correlated event and correlating event. append can be used in a node's condition parameter.

Figure 2-4 Drill Down

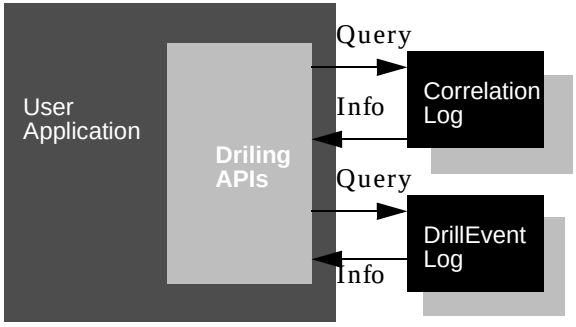


1. An event output by correlation is called a correlated event.
2. An input event contributing to correlation of another event is called correlating event.

The ECS engine collects drill information of each event and logs it to appropriate log files. ECS engine provides a default Correlation log[1] and a default DrillEvent log[2]. Applications may register their own Correlation and DrillEvent logs for a stream. Both these logs can be registered independently. Only the events output on the stream will be logged in them. Logging can be optionally enabled or disabled independently on any stream (If applications did not specify their own log files, default stream logs will be used). ECS engine's default drill logging can also be optionally enabled or disabled.

ECS also provides Drill API's for offline reading of generated drill log files. Refer "Drill API" on page 97 for more details.

Figure 2-5 Drill Down

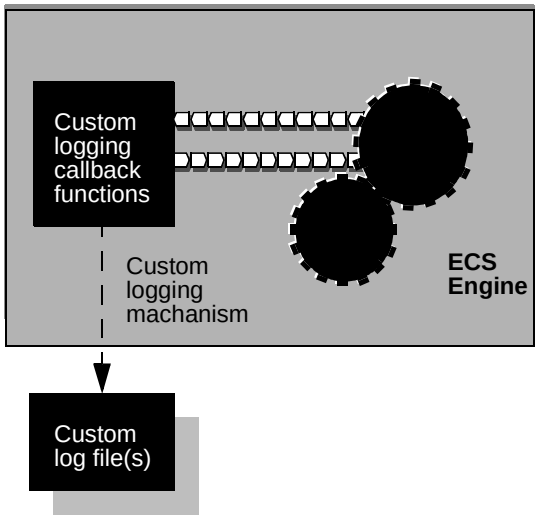


ECS also provides Custom logging framework to allow the integrating applications to provide their own logging mechanisms for drilling events.Refer to "Custom Logging framework" on page 102 for more details.

The framework and the ECDL built-in functions are the external interfaces provided to facilitate the development of custom drilling applications.

1. Correlation log contains relation tree for an event
2. DrillEvent log contains the event for all events in correlation log.

Figure 2-6 Drill Down



Circuit Serialization

When an is event emitted from one circuit and fed to other circuits it is called Circuit Serialization

Events that are emitted from a circuit could be

- one of the events that were input to the circuit
- a new event that was created using the create node in the circuit
- modified primitive event

Primitive events can be fed back to all other circuits including or excluding the circuit which emmited the event. All circuits should be in the same engine. For more information on how Circuit Serilaization works refer to “Circuit Serialization” on page 109.

Multiple Event Creation

At the arrival of an event at the Input port, of the create node, it creates a new event using the specfied Encoding Type and Event Syntax as specified in the Create Node configuration parameters. There is a default event that is always returned to the engine in the case of the Create node. The Create Node can be enhanced to create multiple events upon the arrival of an input event.

Multiple events can be created using multiple (event type,event syntax) pairs where each pair corresponds to one event. The events output from the create node are then modified and then sent to the engine.

The `create_events( )` API is used to create multiple events from a Create node. This API is used in the create spec of a create node. It takes a list of tuples as argument and returns a list of many arguments. For more information on Multiple Event Creation refer “Multiple Event Creation” on page 111

C API Library and Header Files

The Event I/O, Annotate and drill down libraries and header files are installed by selecting the appropriate option during installation. See the *HP OpenView Communications Event Correlation Services Installation Guide* for details.

There are libraries called `libecsio.a` (`libecsio.lib` on Windows) and `libdrill.a` (`libdrill.lib` on Windows) containing all the API functions discussed in this guide and it is installed in:

UNIX

`$OV_LIB`

Windows

`%OV_LIB%`

Header files are installed in subdirectories under:

UNIX

`$OV_HEADER/ecs`

Windows

`%OV_HEADER%\ecs`

You need only include `EIO.h` for event I/O, `ANNO.h` for an annotation server, or both if you are writing an application that supports both functions. All other header files are included as required.

You need to include the drill_api.h for an application using drilling APIs. custom_struct.h needs to be included for developing for custom logging shared library.

Sample source code is in:

UNIX

`$OV_PROG_SAMPLES/ecs/event_io`

Windows

`%OV_PROG_SAMPLES%\ecs\event_io`

UNIX

`$OV_PROG_SAMPLES/ecs/annotate`

Windows

```
%OV_PROG_SAMPLES%\ecs\annotate
```

and

UNIX

```
$OV_PROG_SAMPLES/ecs/drill_down
```

For last minute changes see the following file:

UNIX

```
$OV_RELNOTES/ECS
```

Windows

```
%OV_RELNOTES%\ECS.txt
```

3 The Socket Stack API

This chapter describes the socket stack architecture (ESOK) on which the event I/O (EIO) and annotate (ANNO) APIs are based. It explains the basic steps needed to initialize, open, and reset a socket stack connection, and the limitations you must be aware of when establishing multiple connections.

The Socket Stack Architecture

The ECS Engine event I/O and annotation server interfaces are based on a socket stack architecture that supports multiple connections between one or more engines, and one or more separate user processes. The socket stack architecture is also used to control the engine using the `ecsmgr` program.

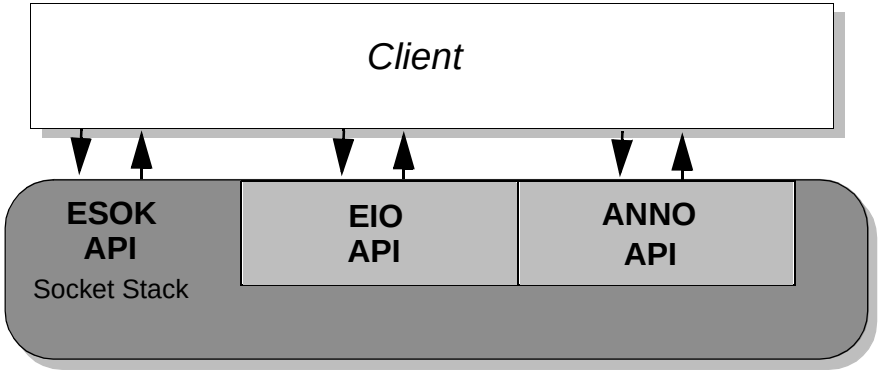
Event I/O and annotate processes are separate to the ECS Engine process. A single process can handle input, output, annotation, or any combination of these facilities. Event I/O and annotation can be added to existing applications that require an interface to ECS, or they can be provided by new applications.

All processes that interface with the ECS Engine must run on the same machine as the engine. Network sockets are not supported.

The APIs are arranged in two levels:

- The socket stack itself, whose primary purpose is to move raw data to and from the engine. This is the ESOK set of API functions.
- Event I/O and annotate APIs that simplify implementation of event I/O and annotate servers on top of the lower-level socket stack. Two APIs are provided: EIO for event I/O and ANNO for annotate servers.

Figure 3-1 Architecture of the Socket Stack APIs



Socket connection limits

As illustrated in Figure 3-2, there are limits to the number of connections that can be established between an ECS Engine and supporting applications. In the context of this illustration, an application is any annotation server or event I/O code that opens a connection. These limits are explained in detail in the following paragraphs.

Up to eight socket stack connections can be opened. A define for this value (ESOK_MAXCON) is provided. The ECS Engine management process (ecsmgr) uses one of these connections, leaving seven connection IDs to be shared between all event I/O and annotate API functions. An error (ESOK_RESOURCE_LIMIT) is returned by ESOK_open( ) if the maximum number of connections would be exceeded.

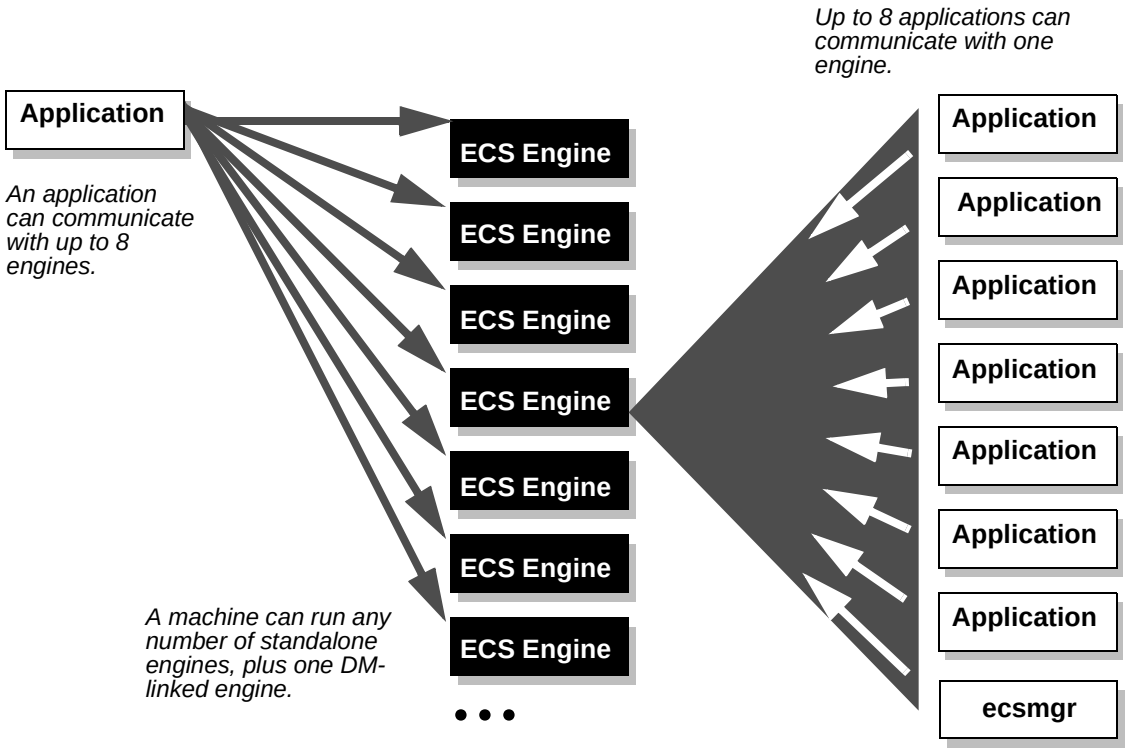
Figure 3-2 shows seven applications communicating with one ECS Engine, plus ecsmgr. The ECS management program (ecsmgr) uses the same socket stack mechanism for communication with the engine as any other application.

WARNING **It is your responsibility to ensure that a spare connection is available for ecsmgr. If a connection is not available you will be unable to use ecsmgr until a connection is freed.**

Event I/O connection Limits

In addition to the socket connection limit, there is a limit of eight event I/O and eight annotate connections with a single ECS Engine. That is, one engine can support a maximum of eight event I/O connections, whether they are on one socket stack connection or spread over several socket stack connections, and the same limitation applies to the annotate API. The EIO_open( ) function returns EIO_RESOURCE_LIMIT if the maximum number is exceeded. A define (EIO_MAXCON) is provided for safe coding.

Figure 3-2 Connection and API Limits



The Socket Stack API (ESOK)

The socket stack API (ESOK) enables a socket-based connection to be established between your application and the ECS Engine. It establishes a generic low-level connection that is shared by:

- Event I/O (EIO) API
- Annotate (ANNO) API
- Management program (`ecsmgr`)
- The CGI interface for the NNM ECS Correlation Services GUI

The socket stack API is capable of supporting connections between multiple applications and multiple engines running on the one machine.

To establish a connection that can be used by the event I/O API or the annotate API your program must complete the following steps. All these steps are described in Chapter 4, “Event I/O,” on page 39 and the steps specific to annotation are described in Chapter 3, “The Socket Stack API,” on page 29.

1. Initialize the socket stack.
2. Open a socket stack to obtain a connection ID (`cid`) for a specific engine instance.
3. Initialize the event I/O and/or annotate interface.
4. Open an event I/O or annotate connection on top of the `cid`.

   The first four steps are described for event I/O in “Initialization” on page 48. The next steps are described separately for each of the three architectures
5. For an event I/O application, register one or more callback function(s) to receive events output from the ECS Engine. This step is only required if you want this process to receive events from the engine. For an annotation server, register a function that will be called when an annotation request is generated.

6. For each event I/O callback function that is registered to a specific event I/O connection ID (`eid`), set a filter to ensure that the callback is registered to receive just those events you want. Callbacks registered as the default cannot be filtered. For an annotation server, you establish an annotation connection ID (`annid`) with a specified ECS Engine, circuit, or Annotation node.
7. Loop while passing data to the engine and calling the `ESOK_process()` function to process data through the stack and cause the callbacks to be called when necessary.
8. Close the event I/O and/or annotate API connection. If you want to close all connections specific to this `eid`, or `annid` (or there is only one connection) then this step can be omitted, in which case all connections specific to this task are closed when the connection is reset.
9. Close the `cid`. If you want to close all connections based on this socket stack connection (or there is only one connection) then this step can be omitted, in which case all connections are reset when the socket stack is reset.
10. Reset the event I/O or annotate connection.
11. Reset the socket stack.

If your application will dynamically open connections (`cids`, `eids` or `annids`) you should take special care not to exceed the connection limits discussed in “The Socket Stack Architecture” on page 31. In addition, you must ensure that a `cid` is always available for `ecsmgr` to use when necessary.

Designing an ECS Engine Interface

The select() loop

The major issue when designing a process to use the ECS socket stack interface is whether to base the design around a `select()` loop or not. Using a `select()` loop is a simple and efficient way to ensure that the socket stack gets called promptly, only when there is processing to be done, in a real-time environment.

However, when processing events outside real-time (such as correlating event history files), the event input process can be simplified to a loop that simply tests for end of file.

The following pseudocode outlines the main elements of a `select()` loop:

Figure 3-3 Elements of the select() loop.

```
forever                                  Loop forever...
{
   ESOK_getFdSet(...)                    Set mask to include socket stack fds.
   FD_SET( my_fd, ...)                   Include fd we are reading events from.
   select(...)                           Wait for activity on one of the fds...
   if( FD_ISSET(my_fd) )                 If an event is ready for input...
   {
     read( my_fd, ...)                   Read the event and send it to the ECS
     EIO_sendEvent(...)                  Engine through the EIO interface.
   }                                     Could call ANNO_sendEvent instead.
   ESOK_process(...)                     Call the socket stack to give it
}                                        processing time and (possibly)
                                         call the receive callback function.
```

The sequence of events inside the `select()` loop sets up a bitmap, where each bit in the bitmap represents a file descriptor. A 1-bit represents a file descriptor you are interested in and a 0-bit indicates that you are not interested in that file descriptor. The `select()` call returns when there is activity on a file descriptor you are interested in.

There are two types of file descriptors you must watch:

- File descriptors in use by the ECS Engine.
- File descriptors used to read events from other sockets.

You obtain a mask of file descriptors currently used by the engine by calling `ESOK_getFdSet()`. Additional file descriptor bits are set explicitly (using the `FD_SET()` macro if available).

Shutting down

When a `select()` loop is used, the program's main loop is usually infinite. This is required because it is not possible to know when the last event has been processed. For example, a circuit can introduce an arbitrarily long delay while waiting to see if a cancelling event arrives. At the least you should provide a signal-based interrupt handler so that the event I/O, annotate and stack connections can be reset before exiting. More sophisticated mechanisms to close individual connections can be implemented as required.

See Also

For more information on `select()`, see:

- "Using a Select Loop" on page 58 for an explanation of the `select()` loop used in the supplied sample program `ecsio.c`
- The following reference pages:

HP-UX:	*select*(2)
Linux:	*select*(2)
Solaris:	*select*(3c)
Windows:	Online documentation provided with your development environment.

4 Event I/O

This chapter discusses how to design and construct event input and output processes HP OpenView Event Correlation Services Engine. The process is illustrated with three simple approaches that highlight the issues involved.

Designing Event I/O

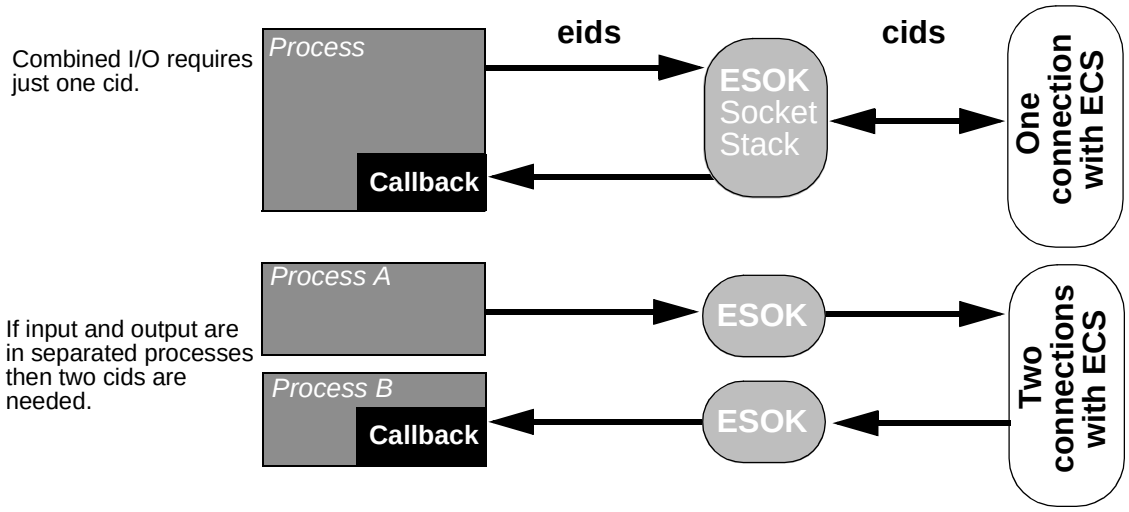
Before designing your event I/O code, you must decide the following:

- How many separate processes do you need and how will these processes connect with ECS Engine(s)? Each process uses additional resources. For example, if you have two independent processes: one for event input to the engine and another to handle event output, the same connection cannot be shared by both.
- How will the `create_time` event header attribute be set? Creation time can be an important factor in event correlation.
- Is any special event handling needed (for example, to handle binary-encoded SNMP traps)?
- Will a `select()` loop be used? (See "Designing an ECS Engine Interface" on page 36.)

Connection resources

The number of connections supported by an ECS Engine is limited. Each additional process that communicates with the engine requires an additional connection.

Figure 4-1 Connection Resources and Processes



Use the smallest number of separate processes to communicate with the engine. This will conserve scarce connection resources, as well as improving efficiency by eliminating the overhead incurred for each new process.

Header attributes Decide how ECS event header attributes will be determined. Some attributes can be set through the event I/O API. In conjunction with the ECS circuit designer and the developer of the MDL message definition, you must decide if and when to override header attributes such as `create_time` and `event_type`.

Event handling Different event protocols will require specific handling in the event I/O process. For example, SNMP events use a different endecoder to ASCII events. Consequently, they require different handling within the event I/O process.

SNMP Traps

The event I/O APIs can be used to process SNMP Traps. However, if the pmd-linked version of ECS is used, then SNMP Traps must be processed through the postmaster and cannot be processed through the event I/O APIs.

The SNMP Trap is supplied as an ASN.1 syntax BER-encoded SNMP Trap-PDU Message (as described in RFC1157). Obtaining this PDU from the network is the developer's responsibility. When designing event I/O to handle SNMP events, be aware that the SNMP event PDU string (`pdu`) will contain embedded nulls. This means that you cannot use functions that assume null-terminated strings, such as *strlen*(3C). Instead, you must find out the length of the PDU by other means.

SNMP events always have an encoding_type of `"ber"` and an event syntax of `"Trap-PDU"`. The `event_type` header event attribute is set to the generic trap number.

Using the Event I/O API in a pmd-linked Environment

In a pmd-linked environment, ASCII events can be processed through the event I/O APIs while SNMP and CMIP events are processed through the postmaster. Such an architecture makes it possible to convert between all three event formats, using correlation circuits that recognize patterns of events in one format and create events in an alternative format. See the appropriate module guide(s) for further details:

- *HP OpenView Event Correlation Services SNMP Module*
- *HP OpenView Event Correlation Services CMIP Module*
- *HP OpenView Event Correlation Services ASCII Module*

pmd	SNMP and CMIP events *cannot* be processed through the eve APIs of a pmd-linked engine.

ECS Event Header Attributes

There are seven ECS event header attributes. These attributes are added to all events when they enter the ECS Engine and are stripped and discarded when the event leaves the engine. These attributes are summarized in Table 4-1.

Table 4-1 Event Header Attributes

Header Attribute	Description	Remarks
encoding_type	The endecoder module used to decode and encode the event.	*These three parameters collectively define the event's structure.*
event_syntax	The ber mib or mdl syntax that defines the event structure.	
event_type	Defines the detailed event structure.	
create_time	The time at which the event was created.	
arrival_time	The time at which the event was received by the ECS Engine.	*These three parameters are visible only inside the ECS Engine. They are not accessible in the API.*
unique_id	A value that uniquely identifies an event.	
event_class	Classifies an event as primitive, temporary or composite	

Some of these header attributes can be controlled from the event I/O API. How and why you would want to do this is explained in the following sub-sections.

Refer to the appendix for a complete description of the `EIO_sendEvent()` and `EIO_addFilter()` API calls discussed below.

encoding_type

The `encoding_type` is passed as a parameter to the `EIO_sendEvent()` function. The following string values for encoding types are supported by this API:

Table 4-2 Valid encoding_type Values

Event format	encoding_type
SNMP events	"ber"
8-bit text (and ASCII) events	"mdl"

No other values are valid in the event I/O API.

The encoding_type is also a parameter to EIO_addFilter(). The value specified here determines which receive function is called for a given encoding type.

The encoding_type is also a parameter to the receive function itself. The value passed is always the same as that specified for the corresponding EIO_addFilter().

event_syntax

The event_syntax is a parameter to EIO_sendEvent(). The value of event_syntax varies depending on the event, as follows:

SNMP Events SNMP events have an event syntax of "Trap-PDU". When calling EIO_sendEvent() with an SNMP trap you must set the event_syntax parameter to the string value "Trap-PDU".

ASCII Events ASCII events have an event syntax that matches the syntax name in the MDL message definition. For example, the SimpleEvent message definition contains the following statement that defines the syntax name:

```
syntax SimpleEvent
    attributes
        ...
```

For further details on MDL see *HP OpenView Event Correlation Services ASCII Module.*

create_time

Event **create_time** is a header attribute added to all ECS events that is used by the ECS circuit to compensate for delays in the delivery of events.

There are three ways that the `create_time` header attribute can be set:

- The `create_time` parameter to `EIO_sendEvent()` overrides any other settings.
- If the `create_time` parameter to `EIO_sendEvent()` is set to 0 then an attempt is made to obtain the creation time from the event, as described below.
- Unless otherwise specified, an event's `create_time` is set to the time that it arrived at the ECS Engine.

If the `create_time` is obtained from the event then the method used depends on the event as follows:

SNMP SNMP events do not contain a record of the time at which they were created. If you are able to calculate a creation time then you could pass this value as a parameter to `EIO_sendEvent()`. Otherwise, if a value of 0 (zero) is passed to `EIO_sendEvent()` then the `create_time` is set to the time that the event arrived at the ECS Engine.

ASCII The MDL message definition can define create_time in terms of some other event attribute(s). If a creation time is not defined in the message definition (that is, if there is no create_time part in the MDL) then it is set to the event's arrival time at the ECS Engine, unless overridden by `EIO_sendEvent()` `create_time` parameter.

Compiling

The supplied library (`libecsio.a` (`libecsio.lib` on Windows)) must be linked with your event I/O code.

The only header files that you need to include in your source code are listed in Table 4-3.

Table 4-3 C Header Files

API	#include file
ESOK only	`sockstack.h`
EIO	`EIO.h` (includes `sockstack.h`)

Additional header files are automatically included as required.

Writing an Input Process

This section describes the construction of a simple ECS event input process. This process initializes the socket stack, opens a connection, sends a series of events that it reads from standard input, and resets.

Sample code for this process is in:

HP-UX, **Solaris**, **Linux**

```
$OV_PROG_SAMPLES/ecs/event_io/ecsin.c
```

Windows

```
%OV_PROG_SAMPLES%\ecs\event_io\ecsin.c
```

Initialization

Before communicating with the ECS Engine you must initialize the socket stack and open a connection. This is done in four steps:

1. Initialize the socket stack.
2. Open a socket stack to obtain a connection ID (`cid`) for a specific engine instance and a specific stream.
3. Initialize the event I/O interface.
4. Open an event I/O connection (`eid`) on top of the `cid`.

The following code fragment establishes an event I/O connection with the engine identified by `instance`. The event I/O connection ID is represented by `eid`.

```
int instance = 1;
ESOK_Remote remote;
ESOK_ConnectionId cid;
EIO_ConnectionId eid;

...

/*
** Build a socket connection structure from the instance number
** of the engine that we want to communicate with.
*/

ESOK_buildRemote( instance, &remote );
```

```
/*
** Initialize the stack, open it, and then open a connection with
** the ECS engine
*/

ESOK_stackInit ();
EIO_stackInit ();
ESOK_open( &remote, &cid );  /* open a socket connection */
EIO_open (cid, 0, &eid);     /* open an EIO connection
                              ** with default (0) stream   */
```

For clarity, error checking has been eliminated from this example. Normally, you would check that each `ESOK_` and `EIO_` function returns `ECS_SUCCESS`.

The function `ESOK_buildRemote()` fills in a structure of type `ESOK_Remote`which is then passed to `ESOK_open()` to establish the socket stack connection. The `instance` parameter to `ESOK_buildRemote()` identifies the ECS Engine to connect with. Multiple `cid` connections can be established with the same engine instance. To establish connections with multiple engines you must call `ESOK_buildRemote()` with the appropriate `cid` instance parameter for each engine instance.

If you used *malloc*(3C) to allocate storage for the remote structure, it can be freed immediately after a successful call to `ESOK_open()`.

The socket stack connection (`cid`) is passed to `EIO_open()` along with the address of the `eid` to be initialized. The `eid` is used in all further communication with the engine, in much the same way as a file handle is used for file I/O. Multiple `eid` connections can be established on the same or different `cid`s.

Instance numbers

By default the instance number for the first instance of an ECS Engine on a given machine is 1. However, if an engine is already running on the same machine then the second and subsequent instances must each be assigned a unique positive instance number when each instance is started, using the command:

`ecsd -i` *n*

Where *n* is the instance number. See *HP OpenView Event Correlation Services Administrators Guide* for details.

Sending Events

Once an Event I/O connection ID (eid) has been established we can use it to send events to the ECS Engine, as in the following code fragment that simply reads 'events' from stdin one line at a time, and sends them to the engine identified by eid:

```
int instance = 1;
ESOK_Remote remote;
ESOK_ConnectionId cid;
EIO_ConnectionId eid;
int32 create_time    = 0;
char* syntax         = NULL;
char* encoding_type  = NULL;
char  pdu [MAX_PDU_LEN];
int32 pdu_length     = 0;
int rc;

...
/*
** Build a socket connection structure from the instance number
** of the engine that we want to communicate with. Unles otherwise
** specified (on the command line) the instance number is always 1
*/

ESOK_buildRemote( instance, &remote );
/*
** Initialize the stack, open it, and then open a connection with
** the ECS engine
*/

ESOK_stackInit ();
EIO_stackInit ();
ESOK_open( &remote, &cid );  /* open a socket connection */
EIO_open (cid, 0, &eid);    /* open an EIO connection
                            ** with default (0) stream   */

/*
** Read an event from stdin, process it, and continue reading the
** next event until there are no more events to read. Events are
** assumed to be separated by newline characters.
*/

create_time = 0;       /* create_time will come from the event */
encoding_type = "mdl"; /* Always MDL for ASCII events          */
syntax        = "SimpleEvent"; /* SimpleEvents only      */
```

```
while (gets (pdu))
{
    pdu_length = strlen( pdu );
    if (pdu_length <= 0)
    {
    continue; /* Don't send the empty line as an event */
    }
    EIO_sendEvent (eid, pdu, pdu_length, create_time,
                   encoding_type, syntax);
    /*
    ** Call ESOK_process() so that the socket stack has a chance
    ** to do some processing. Specify a wait of 0 ms so that it
    ** returns immediately if there is nothing to do.
    */
    ESOK_process (0);
}
```

The maximum size of the event buffer pointed at by `pdu` is subject to any limits imposed by the underlying transport mechanism, and to the limits imposed by the MDL endecoder (currently 8K bytes).

In this example, the actual size of the event is determined with a simple call to *strlen*(3C). However, SNMP events need not be null-terminated and may contain embedded nulls, and require a more sophisticated approach to determining the event size.

The `create_time` parameter may be used to override the event `create_time` header attribute. A value of 0 (zero) causes `create_time` to be calculated in the normal way. See “create_time” on page 46 for details.

The `encoding_type` parameter must point to a string value that the engine can use to identify the appropriate endecode module. See Table 4-2 for a list of valid string values.

The `syntax` parameter points to a string value that the endecode module uses to identify the particular message definition to use when decoding and encoding this event. In this example, the `"SimpleEvent"` syntax is a hard-coded value. However, it may be necessary to infer the value from the event itself, before calling `EIO_sendEvent()`. For example, if your I/O process deals with several different event encodings, you may need to partially decode the event to determine its syntax.

`EIO_sendEvent()` returns a value of `ECS_SUCCESS` if the event is sent successfully. In this example the loop is broken if any other value is returned.

The socket stack and event I/O implementation does not rely on threads, signals or any other multi-tasking mechanism. It is therefore essential for your event I/O process to make regular calls to ESOK_process() to provide the API code with execution time. ESOK_process() takes just one parameter, which determines how many milliseconds to wait for some activity to occur. A value of –1 will cause it to wait forever. A value of 0 (zero) will cause it to return immediately if there is no processing to be done.

Resetting the Connection

Your input process may run *forever*, waiting for events to arrive over a network connection and dispatching them to the ECS Engine for correlation.

Alternatively, it may process a finite number of events and exit. Calling EIO_sendEvent() is not sufficient to ensure that all events have been processed through the socket stack. You must arrange for ESOK_process() to be called *after* the last event has been sent, and you must continue calling ESOK_process() until all events have been processed through the socket stack. This raises the problem of knowing when all the events sent through the stack have been processed and when to stop calling ESOK_process(). This is conveniently achieved by waiting for ESOK_stackEmpty() to return non-zero, as in the following code fragment:

```
while ( ! ESOK_stackEmpty() )
    ESOK_process(0);
```

Waiting for a value of ESOK_NOTHING_DONE to be returned from ESOK_process() is not sufficient by itself.

When all events have been processed through the socket stack you can close connections and reset the stacks, as in the following code fragment:

```
EIO_close( eid );
EIO_stackReset();
ESOK_close( cid );
ESOK_stackReset();
exit (0);
```

If there is only one connection (as in this example) calling EIO_close() and ESOK_close() is unnecessary because the corresponding EIO_stackReset() or ESOK_stackReset() calls will close any outstanding connections.

However, if you had established several `eid`s on a single `cid`, the `EIO_close()` function allows you to selectively close connections. Alternatively, where you have established different `cid`s (possibly with different ECS Engine instances) `ESOK_close()` allows you to close a connection with one engine, without affecting others.

Error handling

Most API functions return `ECS_SUCCESS` if they complete successfully. You should generally check for this value and take corrective action if any other value is received. See the function reference (manpages) for specific return codes and suggested actions.

In general, return codes less than `ECS_SUCCESS` are errors and codes greater than `ECS_SUCCESS` are warnings that indicate that the function completed but the result may not be as expected.

Writing an Output Process

Obtaining events output from an ECS Engine requires registration of a callback function. The callback function is called by the API interface code whenever an event is available, during the processing phase initiated by the `ESOK_process()` function.

The output process used to illustrate this is contained in the file:

HP-UX, **Solaris**, **Linux**

```
$OV_PROG_SAMPLES/ecs/event_io/ecsout.c
```

Windows

```
%OV_PROG_SAMPLES%\ecs\event_io\ecsout.c
```

It initializes a connection, opens it, registers a simple callback function that writes events to stdout, and loops while calling `ECS_process()` and waiting for events to arrive.

Event output processes cannot come and go in the same way that an event input process can. Where an input process may be able to detect the end of a stream of events it is not possible for the output process to know if more events will arrive in the future. Because of this, output processes must run 'forever', or at least as long as output is of use.

Initialization

Initialization is identical to that described for event input in "Initialization" on page 48. If input and output are being done in the same process then the same `eid` connection can be shared for input and output.

Receiving Events

To receive events you must do four things:

- Call `EIO_Open` to open a connection with a specific stream on a specific engine.
- Add one or more filters to discriminate between different events.
- Register at least one callback function to receive the events.

- Call ESOK_process() to process events through the socket stack.

The following example registers a callback function and adds a single filter that allows the callback function to be called for all events with an encoding_type of "mdl":

```
char* syntax         = NULL;
char* encoding_type  = NULL;

...

/*************************************************************
* A filter must be added before events can be received.      *
* Here, we set the filter to allow mdl events of all sorts to be*
* accepted                                                   *
*************************************************************/

encoding_type = “mdl”;      /* Must be MDL for ASCII events */
syntax        = NULL;       /* all event syntaxes           */
EIO_addFilter (eid, encoding_type, syntax);

/*************************************************************
* Register a callback function. The function address passed  *
* here is called whenever an event is ready for output from  *
* the ECS engine. If eid is set to EIO_ALLCON this function  *
* will be called for all PDUs that don’t have a specific     *
* receive function specified.                                *
*************************************************************/

EIO_registerReceiveFn( eid, recvFn );

/*************************************************************
* Call ESOK_process() to process events.                     *
* Loop while we receive good return codes from ESOK_process().  *
* If any other value is received then the loop is exited     *
*                                                            *
* Loop forever (until process is interrupted by Ctrl+C, etc)    *
*************************************************************/
mSecs = 0;
while ( (rc = ESOK_process( mSecs )) == ECS_SUCCESS
            || rc == ESOK_NOTHING_DONE);
```

The call to ESOK_process() provides the API with an opportunity to call the registered callback function (recvFn) when appropriate. The while loop runs for as long as ESOK_process() returns a non-error value. If it returns an error code the loop is exited and you should test the value of rc to determine why ESOK_process() failed.

The receive callback function recvFn must be supplied by you. You can register one callback function for each eid. If you want to register several callback functions (for example, to handle different event types in different ways) you must open an eid for each one.

There are two ways that you can control when a callback function gets called:

- Specify a valid eid and specify a filter. The filter controls the encoding_type and event_syntax for which this callback function is invoked.
- Or, specify a value of EIO_ALLCON for the eid parameter. This registers the callback as a default callback function.

In the second case, the callback function registered with an eid of EIO_ALLCON is called if an event is not processed by any of the other registered callbacks.

If EIO_ALLCON is specified for the eid then a filter cannot be specified. Otherwise, if an eid is specified then you must call EIO_addFilter() before any events can be received.

For example, there may be several receive functions registered for different eids, each of which is filtered on a different encoding_type and event_syntax. In addition, a default receive function could be registered with an eid value of EIO_ALLCON that receive all events that do not match any of the filtered eids.

A trivial receive function that simply writes the event to stdout is:

```
int32
recvFn(EIO_ConnectionId id,
        const void* pdu,
        int32 pdu_length,
        int32 create_time,
        const char* encoding_type,
        const char* syntax )
{
    fwrite (pdu, 1, pdu_length, stdout);
    return ECS_SUCCESS;
}
```

The parameter streamId is reserved for future use.

fwrite() is used in preference to printf() because the string pdu may not be null-terminated.

Resetting the Connection

As mentioned previously, generally you cannot know when all events have been processed and so the main loop for the event output process loops *forever*.

The sample program shown in `ecsout.c` takes a simple approach and simply resets all connections if a signal is received. A more sophisticated approach might implement a control interface that could close individual connections.

Stream

To register for the default stream, use `EIO_open(cid, 0, &eid)`.

To register for a stream called “myStream”, use:

```
EIO_open(cid, “myStream”, &eid)
```

Using a Select Loop

The `select()`system call can be used to construct a convenient and efficient multi-tasking system based around file activity. This method replaces the simple loop used in previous examples with a slightly more complicated loop that includes a call to `select()`. If there is no activity on any of the file descriptors we are interested in then `select()` blocks until there is (or until it times out).

The sample source code in the following files illustrates how the `select()` call is used:

HP-UX, **Solaris**, **Linux**

`$OV_PROG_SAMPLES/ecs/event_io/ecsio.c`

Windows

`%OV_PROG_SAMPLES%\ecs\event_io\ecsio.c`

In this example, events are read from `stdin` and sent to the ECS Engine. A child process is used to simulate a simple network connection so we can monitor the input file descriptor in the `select()` loop. The process also registers a callback function (`recvFn()`) that is called whenever an event is available from the engine.

```
/*************************************************************
* This process is at the other end of the socketpair with child().
* It does all the work of routing events received from
* child() into the ECS engine via the ECS socket stack,
* setting up a callback function to receive events back
* from the ECS engine, and controlling the ECS socket stack
* by calling ESOK_process().
*************************************************************/

int
parent( ESOK_Remote *pRemote )
{
  ESOK_ConnectionId cid;
  EIO_ConnectionId  eid;
  int32 create_time;
  char* syntax;
  char* encoding_type;

  char  pdu [MAX_PDU_LEN];
```

```
int32 pdu_length;

int rc = 0;

int32 nfds = 0;
int32 msize = 0;
int32 nbits = 0;
fd_set readFds;
fd_set writeFds;
fd_set exceptFds;
const fd_set* fds = 0;

FD_ZERO(&readFds);
FD_ZERO(&writeFds);
FD_ZERO(&exceptFds);

/*
** Initialize the stack, open it, and then open a connection
** with the ECS engine
*/

ESOK_stackInit ();

EIO_stackInit ();

ESOK_open(pRemote, &cid); /* open a socket connection  */

EIO_open (cid, “default”, &eid); /* open an EIO connection   */
                                 ** with a named stream
                                 ** (“default” in this case) */

/*
** A filter must be added before events can be received.
** Here, we set the filter to allow mdl events of all sorts
** to be accepted. The filter is applied to the open
** connection identified by eid.
*/

encoding_type = “mdl”;   /* Must be MDL for ASCII events */
syntax        = “SimpleEvent”; /* SimpleEvents only      */

EIO_addFilter (eid, encoding_type, syntax );

/*
** Register a callback function. The function address passed
** here is called whenever an event is ready for output from
```

```
  ** the ECS engine. If eid is set to IO_ALLCON this function
  ** will be called for all PDUs that don't have a specific
  ** receive function specified.
  */

  EIO_registerReceiveFn( EIO_ALLCON, recvFn );

  /*
  ** Loop around while we keep track of i/o activity by examining
  ** the result of a select() call. If there is there is data
  ** waiting on the input stack, read it, and route it to the ECS
  ** engine. Call ESOK_process() to ensure that the stack gets
  ** processor time.
  **
  ** IMORTANT: we cannot know when the 'last' event has been
  ** received from ECS and so cannot deduce when to exit this loop.
  **
  ** In this test program press Ctrl+C to kill it.
  */

#ifdef SYS_X86_NT35
  fds = (fd_set *)malloc(sizeof(fd_set));
#endif

  for (pdu_length=1; ; )     /* forever */
  {
    /* get file descriptors in use by ECS engine */
    msize = ESOK_getFdSet(ESOK_READMASK, &nfds, &fds);
    memcpy(&readFds, fds, msize);

    msize = ESOK_getFdSet(ESOK_WRITEMASK, &nfds, &fds);
    memcpy(&writeFds, fds, msize);

    msize = ESOK_getFdSet(ESOK_EXCEPTIONMASK, &nfds, &fds);
    memcpy(&exceptFds, fds, msize);

    /* Set additional file descriptors we want to select on.
    ** Because select() returns immediately if a file cannot
    ** be read, only set the bit for this fd if the last read
    ** returned something, as indicated by pdu_length.
    */

    if( pdu_length > 0 && !FD_ISSET(fildes[FD_PARENT], &readFds))
      FD_SET(fildes[FD_PARENT], &readFds);

      /* Wait for some activity on nominated fds */
```

```
      nbits=select(nfds, &readFds, &writeFds, &exceptFds, 0);

      /* If there is an incoming event on our socket fd,
      ** read one event and send it to the ECS engine
      */
      if( FD_ISSET(fildes[FD_PARENT], &readFds) )
      {
        pdu_length = read_pdu(fildes[FD_PARENT], pdu, MAX_PDU_LEN);
          if( pdu_length > 0 )
          {
            ++numEventsSent;
            EIO_sendEvent( eid, pdu, pdu_length, create_time,
                           encoding_type, syntax);
                     }
          else
#ifndef SYS_X86_NT35
            close( fildes[FD_PARENT] );
#else
            closesocket(fildes[FD_PARENT]);
#endif
      }

      /*
      ** ESOK_process() must be called in this loop to ensure that
      ** the ECS stack and the receive function get a share of
      ** processor time.
      */
      ESOK_process(0);
   }

/*execution never reaches this point, exit is via signal handler*/
}
```

The ESOK_getFdSet() API call fills in a pointer to the file descriptor bitmask and the size of the bitset. The size of the bitset is variable. A bit is set for each file descriptor in use by the ESOK API. Three separate calls are needed, one each for read, write and exception file descriptors.

Once the ESOK API file descriptors have been set, the FD_SET() macro is used to set the bit in the read mask that corresponds to the file descriptor fd that we read events from. In this example we use pdu_length as a flag to indicate whether end-of-file has occurred and only set the bit if we have not yet reached the end of the file.

The call to select() blocks until there is activity on one of the file descriptors in one of the three masks.

The `FD_ISSET()` macro returns true if `fd` is ready for reading, in which case we read an event from `fd` and send it to the ECS Engine.

Note that `ESOK_process()` is called whenever `select()` returns, even if we do not send an event to the engine. This is required for two reasons:

- To process events in the socket stack.
- To provide an opportunity for the callback function to be invoked (if events are received from the engine by this process).

See Also

For more information on `select()`, see the following references:

HP-UX:	*select*(2)
Solaris:	*select*(3c)
Windows:	Online documentation provided with your development environment.

5 Annotation Servers

The Annotate node allows an external mechanism to be used to obtain data asynchronously from outside the ECS Engine. The data could typically be corporate data such as equipment inventory, network topology or management information.

The Annotate node depends on the existence of an appropriate annotation server process. The Annotate node sends a request to the server and waits for a response. Both the request and the response are complex data types capable of conveying arbitrary amounts and types of data.

Refer to the *HP OpenView Event Correlation Services Designer's Reference* for a description of the configuration and functionality of the Annotate node.

This chapter provides a detailed discussion of the annotation mechanism from the perspective of the developer of an annotation server, followed by a discussion of how an annotation server should be created:

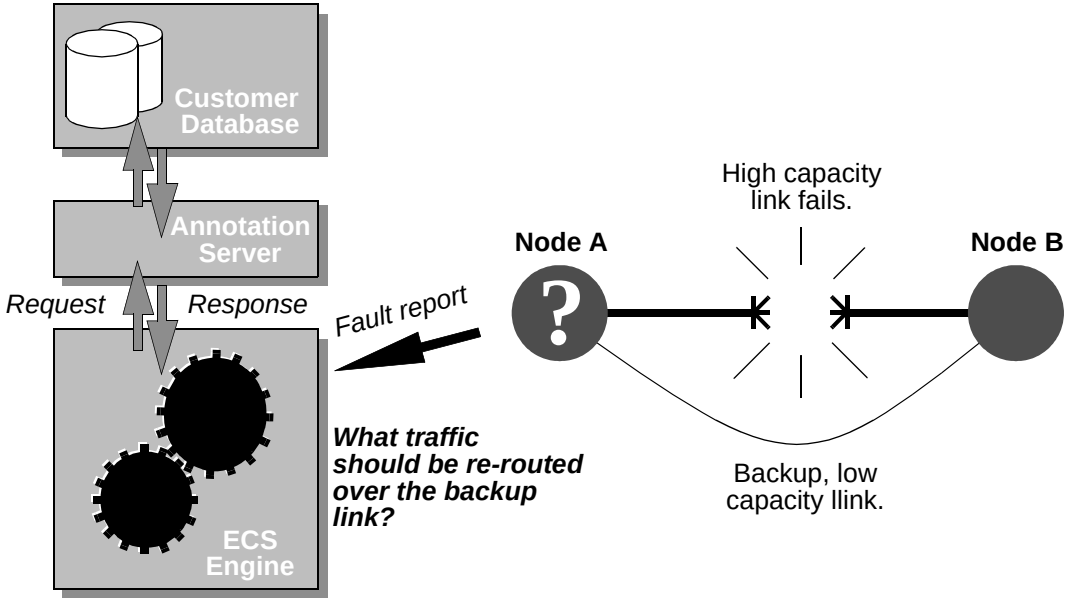
Annotation Concepts

The HP OpenView Event Correlation Services (ECS) annotation mechanism allows a request for data to be issued by an Annotate node in an ECS circuit and sent to some process that is *external* to the engine. The requested data may be obtained from a remote system, totally unrelated to ECS, and a response returned to the Annotate node that issued the request.

Consider the situation where a communication facility fails, and that facility provides services to many customers. The question arises: which services should be re-routed to use the limited backup capacity? This raises two further problems: which customers qualify for special treatment, and finally, which services were in use by them.

The event stream itself contains no information about customers or the services they are entitled to—additional data is required.

Figure 5-1 Example Link Failure Problem



The problem and a sample architecture for an ECS-based solution is illustrated in Figure 5-1. Assume that systems exist somewhere that relate equipment IDs to services and customers. Those systems could be queried from within the ECS circuit so that the circuit could automatically generate events to re-route high-priority services for specific customers.

The circuit designer uses an Annotate node as a gateway through which a request for information is issued. The request is sent to a user-developed application referred to as an **annotation server**. This process performs whatever operations are required to obtain the requested data, possibly querying other systems such as equipment databases and billing systems, and returns a response to the requesting Annotate node.

The annotate API (ANNO) provides the developer with a way to exchange requests and responses with one or more ECS Engine running on the same machine. Figure 2-3 on page 19 shows the basic arrangement. In addition to multiple engines, the API supports multiple circuits and multiple Annotate nodes, and guarantees that responses always find their way back to the Annotate node that sent the original request.

The only limitation is that annotation servers and the ECS Engine(s) that they serve must be located on the same machine.

The Annotation Mechanism

The annotate API (ANNO) is a library of functions that you link into your annotation server. The mechanism is similar to the event I/O API (see Chapter 4, “Event I/O,” on page 39 for details). Like the event I/O API, the annotate API depends on the lower-level socket stack API to communicate between the ECS Engine and the annotation server. See Chapter 3, “The Socket Stack API,” on page 29 for details.

Figure 5-2 The Annotation Mechanism

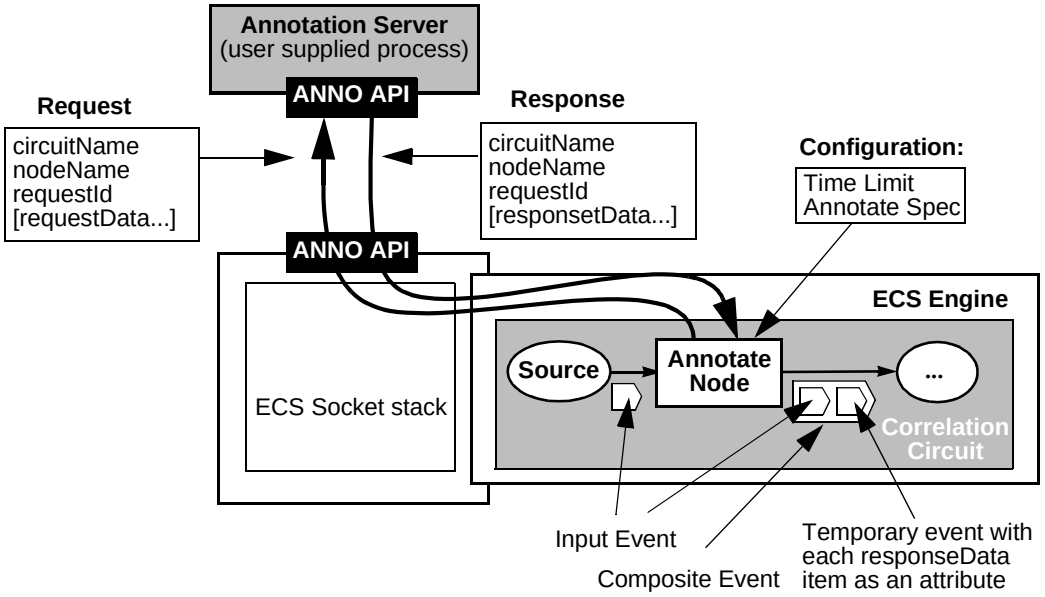


Figure 5-2 shows the annotation mechanism in detail. When an event enters the Annotate node through its Input port, the ECS Engine calls a user-supplied function that you register through the annotate API in the annotation server. The interprocess communication between the engine and the annotation server (a Unix domain socket) is transparent. The callback function receives a pointer to a data structure containing information passed by the circuit designer in the `Annotate Spec` parameter of the Annotate node. This data is a C-language representation of the list defined by the circuit designer.

The annotation server identifies annotation requests by a unique `requestId`. The `requestId` must be returned with the response to ensure that the response is sent to the appropriate Annotate node in the appropriate circuit.

The annotation server decodes the annotation request and determines the action required. The annotation server may discriminate between requests in a number of ways:

- by the name of the circuit
- by the Annotate node name
- on the basis of the data contained in the request itself.

To service requests from a specified circuit and/or Annotate node you specify the circuit name and/or Annotate node name when the annotate API connection is established. This is useful when different circuits require different annotation servers, or when several different types of annotation are used in one or more circuits.

NOTE The circuit designer and the annotation server developer must agree beforehand on the data to be passed between the Annotate node and the annotation server.

The Annotate node has two parameters:

- `Time Limit` specifies the maximum time that the Annotate node can detain the input event while awaiting a response from the annotation server.
- `Annotate Spec` defines the `requestData` to be sent to the annotation server, in the form of an ECDL List.

The `requestData` List can contain nested Lists, Tuples and other ECDL data types. The Annotate node attaches some additional information to the List, serializes it, and forwards it to the annotation server through the ECS socket stack (ESOK) and the annotate API (ANNO).

The annotate API deserializes the data and calls the user's callback function, passing it a pointer to the List of data. The annotation server extracts values from the List and uses them to generate, for example, a database query.

The response data (for example, the result of the database query) is sent back using the reverse mechanism. The annotation server assembles a response List and passes it to the annotate API. The List can consist of any number of values in any mixture of the supported data types.

When the Annotate node receives the response, it creates a temporary event whose attributes are composed of the information passed back in the `EV_attrValue` structure. The temporary event is enclosed in a composite event containing the original input event together with the temporary event, and transmitted from the output port. It is the responsibility of downstream nodes to use this data in subsequent correlation decisions.

Annotation Data Types

Annotation request and response data is passed in the form of an EV_AttrValue structure. This is a recursive structure capable of representing a (nested) list of values of various data types. It is a direct representation of the ECDL List constructed by the Annotate Spec parameter of the Annotate node and supports most ECDL data types (see Table 5-1).

Two API functions are provided to simplify dealing with EV_AttrValue structures:

- EV_extractElement() gets a pointer to the data for a given element in the list. You use this function to extract data from the annotation request.
- EV_parseFormattedValue() constructs an EV_AttrValue from a string representation that looks like an ECDL List of values. The formatted list is most easily assembled using *sprintf* (3C).

These API functions allow you to treat the EV_AttrValue as a handle to an opaque data structure representing an ECDL List or Tuple data value. Their use is explained in detail in the following sections.

CAUTION

The EV_AttrValue structure is documented in $OV_HEADER/value_struct.h. However, because of the complexity of this structure and because it is subject to change in future releases, pointers to EV_AttrValues should be regarded as opaque. Direct access to EV_AttrValue internal structure should be unnecessary and is not supported as HP may make changes to this structure at any time without notice.

The Type Identifier in Table 5-1 defines a mapping between an ECDL data type in the original Annotate Spec parameter, and a C data type (or structure).

Some data types are represented as simple values (for example, EV_INTEGER is stored as a single 32-bit integer value). Other data types are represented as structures (For example, EV_TIME consists of two integers: *seconds* and *microseconds*; and EV_LIST and EV_TUPLE are EV_AttrValues).

Table 5-1 EV_AttrValue Data Type Summary

Type Identifier and ECDL Type	Example Response Text String	C Data Type	For deta see...
EV_INTEGER Integer	123	int32 *	"Integer" o page 72
EV_REALReal	12.345	real64 *	"Real" on page 72
EV_BOOLEAN Boolean	true	ecsbool *	"Boolean" page 73
EV_TIME Time	19971231095959.0Z	struct { u_int32 _secs; u_int32 _usecs; } EV_Time *;	"Time" on page 74
EV_DURATION Duration	1.5h *or* 12345.67s *or* 1h 30m 2.1s	struct{ int32 _secs; u_int32 _usecs; } EV_Duration *;	"Duration" page 75
EV_STRING String	"LNKUP"	char *	"String" on page 76
EV_OID Oid	1.234.5.67	u_int32 *	"Oid" on page 77
EV_NULL Void	()	0	"Null" on page 78
EV_LIST List	[1, "Two", 3.4]	EV_AttrValue *	"List" on page 78
EV_TUPLE Tuple	(1, "Two", 3.4)	EV_AttrValue *	"Tuple" on page 79

EV_AttrValue structures can be constructed recursively to any depth using Lists and Tuples in the following ways:

- A List, and each of the elements in the List, is an EV_AttrValue that points to the next element. An element may itself be a List or a Tuple data type.
- A Tuple, and each of the elements in the Tuple, is an EV_AttrValue that can be of any data type including List and Tuple.

Integer

The internal representation of an integer is a signed 32-bit quantity.

ECS Circuit	ECDL type:	Integer
	Example:	`[123]`
Request	Type Identifier:	`EV_INTEGER`
	C Data type:	`int32;`

	Example:	`int32 *pint;`
		`...`
		`case EV_INTEGER:`
		`  pint = (int32*) elementData;`

Response	Format:	`sprintf(buf, "[%d]", *pint );`
	Example:	`[123]`

Real

The internal representation of a real number is as a 64-bit IEEE format double precision floating point number.

ECS Circuit	ECDL type:	Real
	Example:	`[1.23]`
Request	Type Identifier:	`EV_REAL`
	C Data type:	`real64;`

	Example:	`real64 *preal;`
		`...`
		`case EV_REAL:`
		`  preal = (real64*) elementData;`

Response	Format:	`sprintf(buf, "[%f]", *preal);`
	Example:	`[1.23]`

Boolean

The internal representation of a boolean is as an `ecsbool` type with the value 1 for true or 0 for false.

ECS Circuit	ECDL type:	Boolean
	Example:	`[true]`
Request	Type Identifier:	`EV_BOOLEAN`
	C Data type:	`ecsbool;`

	Example:	`ecsbool *pbool;`
		`...`
		`case EV_BOOLEAN:`
		`  pbool = (ecsbool*) elementDa`

Response	Format:	`sprintf(buf, "[%s]",` `  *pbool ? "true" : "false");`
	Example:	`[true]`

ECS Circuit **ECDL type:** Boolean

Example: `[true]`

Time

The internal representation of time is two integers counting seconds and microseconds since Unix Epoch (1 January 1970). All times are assumed to be UTC (Universal Coordinated Time).

ECS Circuit	ECDL type:	Time
	Example:	`[19911231235959.123456Z]`
Request	Type Identifier:	`EV_TIME`
	C Data type:	`typedef struct{ u_int32 _secs; u_int32 _usecs; } EV_Time;`
	Example:	`EV_Time *ptime; ... case EV_TIME: ptime = (EV_Time*) elementData;`
Response	Format:	`/* get time into tm, then... */ sprintf(buf, "[%04d%02d%02d%02d%02d%02d.%06dZ]", tm.tm_year+(tm.tm_year>69?1900:2000) tm.tm_mon+1, /* 1 to 12 */ tm.tm_mday, tm.tm_hour, tm.tm_min, tm.tm_sec, usecs);`
	Example:	`[19971231235959.123456Z]`

When converting to seconds be aware that the microseconds component can be greater than 10^6, so (where `ptime` is a pointer to an `EV_Time` struct):

```
realT = (ptime->_secs * 1000000 + ptime->_usecs)
        / 1000000;
```

When building a response from a `tm` time structure remember to add the century to the year, and to count months from 1 to 12 instead of 0 to 11.

The decimal point separating seconds from microseconds is mandatory. If there are no microseconds then `.0` must be specified. For example, `19971231235959.0Z`. If the decimal point is omitted then a runtime error will be generated. The terminating 'Z' (for Zulu) indicates that the time value is UTC and is required to distinguish a Time value from a Real.

Duration

The internal representation of duration is two integers counting seconds and microseconds.

ECS Circuit	ECDL type:	Duration
	Example:	`[1h 33m 29.123s]`
Request	Type Identifier:	`EV_DURATION`

	C Data type:	`typedef struct{`
		`  int32 _secs;`
		`  u_int32 _usecs;} EV_Duration;`

	Example:	`EV_Duration *pdur;`
		`...`
		`case EV_DURATION:`
		`  pdur = (EV_Duration*) elementData;`

Response	Format:	`sprintf(buf, "[%dh %dm %d.%ds]",`
		`  tm.tm_hr,`
		`  tm.tm_min,`
		`  tm.tm_sec,`
		`  usecs);`

	Example:	`[1h 33m 29.123s]`

When converting to seconds be aware that the microseconds component can be greater than 10^6, and that `secs` is signed whereas `_usecs` is unsigned, so you need to be careful to calculate the sign if you convert an `EV_Duration` to a floating point number, as in the following example (where `pdur` is a pointer to an `EV_Duration` struct):

```
real64 realD = (pdur->_secs < 0 ) ?
  ((pdur->_secs * 1000000 - pdur->_usecs) / 1000000) :
  ((pdur->_secs * 1000000 + pdur->_usecs) / 1000000);
```

The formula above ensures that `realD`, a floating point number, has the correct sign.

String

Strings are represented as an array of a specified number of bytes. Bytes in the array may contain any value, including null (`'\0'`).

ECS Circuit	ECDL type:	String
	Example:	`["ber"]`
Request	Type Identifier:	`EV_STRING`
	C Data type:	`char;`

	Example:	`char *s;`
		`...`
		`  case EV_STRING:`
		`  s = (char*) elementData;`

Response	Format:	`sprintf(buf,"[\"%s\"]", s);`
	Example:	`["ber"]`

Because string data types can contain embedded nulls, and because there is no guarantee that strings are nul-terminated, you must rely on the `size` parameter returned by `EV_extractElement()` to delimit the character array. Do not use standard C library string functions such as *strlen*(3C) or *strcpy*(3C) on string elements that may contain nulls.

Oid

Oid data types are Object Identifiers with a text representation of the form *n.n.n...* Each number *n* is an integer value between 0 and 2^{32} inclusive. There must be at least three numbers, and each number must be separated from the next with a period (.)

The native representation of an Oid is as an array of unsigned 32-bit integers, where each number *n* is an integer in the array.

ECS Circuit	ECDL type:	Oid
	Example:	`[123.45.6.7.89]`
Request	Type Identifier:	`EV_OID`
	C Data type:	`u_int32[];`

	Example:	`u_int32 id, *oid;`
		`int n;  /* index of id in oid */`
		`char buf[64], *pbuf;`
		`...`
		`case EV_OID:`
		`  oid = (u_int32*) elementData;`
		`  if (n < size/sizeof(u_int32))`
		`  id = oid[ n ];`

Response	Format:	`for(i=0,pbuf=buf;i<len;i++)`
		`{`
		`  if(i != 0)`
		`  pbuf += sprintf(pbuf, ".");`
		`  pbuf += sprintf(pbuf, "%d", oid[i])`
		`}`

	Example:	`[123.45.6.7.89]`

In the C code fragment for the request example, above, the `size` (in bytes) returned by `EV_extractElement()` is divided by `sizeof(u_int32)` to yield the number of elements in the array of integers.

Null

A Void type is required to support optional event attributes and ASN.1 CHOICE values in SNMP and CMIP events. When `type` is `EV_NULL`, the value of `elementData` is always 0 (`NULL`), and `size` is always 0 (zero).

ECS Circuit	ECDL type:	Void
	Example:	`[ () ]`
Request	Type Identifier:	`EV_NULL`
	C Data type:	`void;`

	Example:	`void *ptr;`
		`...`
		`case EV_NULL:`
		`  ptr = elementData; /*NULL*/`

Response	Format:	`sprintf(buf, "[ () ]" );`
	Example:	`[ () ]`

List

The List data type represents an ordered list of values. The number of elements in a List is variable. A List is represented internally as a data structure of type `EV_AttrValue`. To extract a value from a list, you pass a pointer to the `EV_attrValue` and the index number of the element you wish to extract to the `EV_extractElement()` API function. See “Extracting Values from the Annotation Request” on page 81.

The callback function that you define receives a pointer to a List as one of its parameters.

ECS Circuit	ECDL type:	List
	Example:	`[["Severity", 0], []]`
Request	Type Identifier:	`EV_LIST`
	C Data type:	`EV_AttrValue*;`

	Example:	`EV_AttrValue *AV;`
		`...`
		`case EV_LIST:`
		`  AV = (EV_AttrValue*) elementData;`

Response	Format:	`sprintf(buf,"[[\"%s\", %d],[]]", s, i`
	Example:	`[["Severity", 0],[]]`

The ECS Circuit and Response examples above shows a List with two nested lists, the first containing two elements, and the second containing none.

CAUTION Pointers to `EV_AttrValue` structures should always be regarded as opaque. The details of this structure are not defined, and code that references the internal structure is unsupported.

Tuple

The Tuple data type represents a fixed collection of elements. The Tuple is represented internally as a data structure of type `EV_AttrValue`. To extract a value from a Tuple, you pass a pointer to the `EV_attrValue` and the index number of the element you wish to extract to the `EV_extractElement()` API function. See “Extracting Values from the Annotation Request” on page 81.

The distinction between Tuples and Lists is subtle and is only important within an ECS circuit. In the annotation interface, both Lists and Tuples should be handled as opaque pointers of type `EV_AttrValue *`.

ECS Circuit	ECDL type:	Tuple
	Example:	`[("Severity", 0), ()]`
Request	Type Identifier:	`EV_TUPLE`
	C Data type:	`EV_AttrValue*;`

	Example:	`EV_AttrValue *AV;`
		`...`
		`case EV_TUPLE:`
		`  AV = (EV_AttrValue*) elementData;`

Response	Format:	`sprintf(buf,"[(\"%s\", %d),()]", s, i`
	Example:	`[("Severity", 0),()]`

The ECS Circuit and Response examples above show a List with two Tuples, the first containing two elements and the second containing none. The form of the empty Tuple `()` is the same as a Void data type and is in fact a Void. Since Void data types have no defined value you must be careful not to misinterpret empty Tuples.

CAUTION Pointers to `EV_AttrValue` structures should always be regarded as opaque. The details of this structure are not defined, and the code that references the internal structure is not supported.

Receiving Annotation Requests

Whenever an annotation request is received, the annotate API calls the function that you register with the ANNO_registerReceiveFn() function. The callback function must conform with the following prototype:

```
ANNO_receiveFn( const ANNO_ConnectionId cid,
                const ANNO_requestId requestId,
                const EV_AttrValue *annoRequest );
```

The cid identifies the connection on which the request was received, and the requestId uniquely identifies the request from this connection. Both parameters must be passed back with the response so that the interface can align the response with the original request.

The annoRequest pointer points to an EV_AttrValue structure of type EV_LIST. This is the start of the ECDL List data structure passed in the Annotate node's Annotate Spec parameter.

CAUTION You must not change any of the contents of annoRequest and you must not pass the annoRequest pointer back as a response (through ANNO_sendResponse()).

It is essential that the circuit designer and annotation server developer cooperate in designing the list of data passed from the Annotate Spec parameter of the Annotate node.

Extracting Values from the Annotation Request

Values are extracted from the annotation request list by using the EV_extractElement() API function:

```
int32 EV_extractElement(
            const EV_AttrValue *requestList,
            int32 index,
            EV_AttrType *type,
            int32 *size,
            const void **elementData
        )
```

You pass it a pointer to the requestList and an index representing the number of the element that you want to extract from the list (the first element has an index of 1). If you pass an index value greater than the number of elements in the list then a value of EV_NO_SUCH_ELEMENT is returned.

Otherwise, if the element exists, then EV_extractElement() fills in the type, size and elementData, as follows:

- type is an EV_AttrType with one of the values listed in Table 5-1 on page 71 such as EV_INTEGER, EV_STRING, etc. The type may be used in a switch statement to decode generic requestLists, where the type of an element is not known ahead of time. The type indicates the Type Identifier and hence the C data type to which the elementData pointer must be cast.
- size is a 32-bit integer that is set to the size of the data element pointed at by elementData. The size of the data is measured in bytes and is suitable for use with *malloc*(3C) to allocate storage that can be used to hold a copy of the data. For some data types (such as EV_LIST and EV_TUPLE) the size is 0 (zero).
- elementData is the address of a pointer to a void data type that is set to the address of the data. You must cast this pointer to the appropriate C data type (based on the type value) before accessing the data. Note the double indirection.

The following code fragment illustrates how to copy a String element from a requestList into a null-terminated C string called surname.

```
char* surname = 0;
EV_AttrType type;
int32 size;
void* elementData;

/* -- Get the Surname --- */
int index = 1;
if(requestList != 0
 && EV_extractElement(requestList,index,&type,&size,&elementData)
                                          == ECS_SUCCESS
 && type == EV_STRING
  )

{
    surname = (char*) malloc( size+1 );
```

```
    memcpy( surname, elementData, size );
    surname[ size ] = '\0';
}
```

Data of type EV_STRING is not necessarily nul-terminated, and may contain embedded nuls. It is therefore essential to use memcpy() rather than strcpy(). Note also that an additional byte is allocated for the terminating null that we add.

Copying Data

Normally, it is sufficient to retrieve a pointer to the data. However, there are occasions when you *must* copy the data. These are:

- If the data must be kept for a relatively long time, either after a response has been returned or after the request expiryTime (returned by a call to ANNO_getRequestInfo()) has occurred.
- If the value of the data pointer (*elementData) , or the storage it points to (elementData), may be modified.

If either of these conditions is true then you cannot simply copy the data pointer. Some data types (EV_TUPLE and EV_LIST) cannot be copied as they contain embedded pointers. Such data types always have a size of 0 (zero).

Getting Additional Request Information

ANNO_getRequestInfo() can be used to obtain additional information about a specific annotation request. You supply the Connection ID and Request ID, and ANNO_getRequestInfo() fills in the circuit name, Annotate node name, sequence number, and expiry time. The prototype is:

```
int32 ANNO_getRequestInfo(
            const ANNO_ConnectionId annoId,
            const ANNO_RequestId requestId,
            char **circuitName,
            char **annotateNodeName,
            int32 *seqNum,
            int32 *expiryTime
        )
```

The returned data may be used to discriminate between requests, ascertain the order in which requests were sent, or to determine if the time for a response has expired. The parameters are :

- `circuitName` is the address of a pointer to a string containing the name of the circuit that issued the request. Storage for this string is allocated from the heap and must be freed by calling *free*(3c).
- `annotateNodeName` is the address of a pointer to the name of the Annotate node generating the annotation request. It is a string of the form: *`moduleN.compoundP…compoundT.annotateX`*. Storage for this string is allocated from the heap and must be freed by calling *free*(3c).
- `seqNum` is incremented by the circuit for each annotation request generated by that circuit. This number may be useful in preserving the order in which requests are processed.
- `expiryTime` is the time and date by which a response is needed, if the response is to be included in the Annotate node's output. This `time_t` type value is passed across from the originating Annotate node's `Time Limit` parameter. If the response arrives at the Annotate node after the `expiryTime`, it is discarded by the engine.

The engine makes no allowance for the round-trip network transit delays involved in delivering the request to the annotation server and returning the response back to the Annotate node. It is the responsibility of both the circuit designer and the annotation server developer to allow for any delays.

The annotation server *must* respond to *every* request, even if the `expiryTime` has passed. This allows the API to release resources that are allocated for each request. If a valid response cannot be provided before `expiryTime` has been reached, then a null response can be sent.

Constructing an Annotation Response

Constructing an annotation response requires that you assemble an EV_AttrValue structure containing the data to be returned. You then call ANNO_sendResponse() passing a pointer to this EV_AttrValue structure, the connection ID, and the request ID. The connection ID and the request ID (supplied as the cid and requestId parameters to your callback function) ensure that the response is returned to the Annotate node that issued the request, and that the response is identified with a particular request from that node.

You must not change any of the contents of the original request (annoRequest) and you must not pass the request pointer back as a response. Instead, you must construct a new response using the EV_parseFormattedValue() API function.

Alternatively, if you do not want to send a response, a null response can be sent as described below.

NOTE Every request received by the server must be responded to by calling ANNO_sendResponse(). Resources consumed by the request are freed when a response is sent.

EV_parseFormattedValue() takes an ECDL-like string representation of a list and assembles an EV_AttrValue structure. The simplest way to construct the string representation is to use *sprintf*(3S):

```
EV_AttrValue *responseList;
char responseString[ 1000 ];
char text[] = "Hourly rate";
real64 salary = 1.23;

sprintf(responseString, "[\"%s\", %f]", text, salary);
rc = EV_parseFormattedValue(responseString, &responseList);
if( rc == ECS_SUCCESS )
    ANNO_sendResponse(conId, reqId, responseList);
else
    ANNO_sendResponse(conId, reqId, 0);
EV_deleteValue(responseList);
```

For *sprintf*(3S) formats, refer to Table 5-1 on page 71, and details of the appropriate data type.

Note that `EV_parseFormattedValue()` takes the address of an uninitialized pointer to an `EV_AttrValue` structure. Storage is allocated to the structure by `EV_parseFormattedValue()` and must be released by calling `EV_deleteValue()` when it is no longer needed.

Sending a Null Response

If a meaningful result cannot be returned before `expiryTime` has been reached, or an error occurs (in the example above, if `rc` is not equal to `ECS_SUCCESS`) then a null response can be sent by calling `ANNO_sendResponse()` with a value of `0` (zero) for the `annoResponse` parameter. This allows the API to clean up the corresponding request, preventing memory leaks.

Testing a Circuit with an Annotate Node

A circuit using an Annotate node can be tested in the ECS Designer[1] in Simulate mode. You cannot run the ECS Designer with an annotation server to perform “live” annotation—you can only run “live” annotation with the ECS Engine. Therefore, you must simulate annotation requests and responses using event logs. The ECS Designer has a facility to capture annotation requests to a log, and to read annotation responses from a log.

To use the ECS Designer to simulate annotation you can take one of two approaches:

- Run the circuit in the ECS Engine connected to the real annotation server and collect the input log. This log is then used as both the input and annotation log. When used as an input log, only the events are read (the annotation responses are ignored), and when used as the annotation log, only the annotation responses are read (the events are ignored).
- Run the circuit in the ECS Designer's simulation mode using an input log designed for testing the circuit (this can be either hand crafted or an event log from a running ECS Engine). Capture the

1. Ignore all references of the ECS Designer. The ECS Designer is sold as a separate product and must be ordered separately.

annotation requests and manually alter them to be annotation responses. You can now simulate the annotation circuit in the ECS Designer.

The first approach is more appropriate when you are trying to diagnose a circuit design problem with an existing annotation server. The second approach is generally preferred because it is quicker and simpler, and does not require the ECS Engine or an annotation server. Both approaches are described in detail in the following sections.

Generating an Annotation Log using the ECS Engine

1. Using the ECS Designer, design and build a circuit that performs annotation. In this example, the circuit is called `anno.ecs`.
2. Select `Circuit:Compile` from the ECS Designer menu to convert the circuit to a form that can be loaded by the engine. The compiled circuit is called `anno.eco`.
3. Start an ECS Engine and load the circuit, naming it “simulate”. You *must* give it the name “simulate” if you want the resulting logs to be usable by the ECS Designer. For example:

```
ecsd
ecsmgr -circuit_load simulate anno.eco
```

4. Enable input logging on the circuit and enable the circuit. For example:

```
ecsmgr -log_events_in on
ecsmgr -enable simulate
```

5. Start your annotation server(s).
6. Fire events at the ECS Engine – either live events or an event log sent using the `ecsevgen` utility.
7. When your input events have finished, and you have received all of your annotation responses, shut down the ECS Engine and your input and annotation event log is in the `$OV_LOG/ecs/1/ecsin.evt0` log file just generated.
8. Rename the event log to `ecsin.evt` so you can read it in the ECS Designer.
9. Start the ECS Designer and load the circuit.
10. Switch to Simulate mode by selecting the `[Simulate]` button.

11. Load the `ecsin.evt` log as the input log by selecting `Simulate:Load Input Events`. Inspection should show that it contains only normal events, not the annotation responses.
12. Load the `ecsin.evt` log as your annotation log by selecting `Simulate:Load Annotation Events`. Inspection should show that it contains only annotation responses.
13. Start the simulation by selecting `[Run]`. The input events should enter the circuit and generate annotation requests. These requests will match the appropriate responses out of the annotation responses log, but they will not enter the circuit until the transit delay has expired.
14. If annotation responses arrive after the last event in the input log, select the Step-by `[Time]` button to see the responses enter the circuit. You may find it convenient to set a breakpoint on the Annotate node and set a large value for the time.

Generating an Annotation Response Log using the ECS Designer

It is frequently quicker and easier to use the ECS Designer to generate annotation requests, and to hand craft these into annotation responses.

The great advantage of this approach is that you can develop and simulate circuit containing Annotate nodes before building the annotation server. Once you know how the circuit and the annotation server *should* operate, you can construct the annotation server to provide the responses required.

1. Using the ECS Designer, design and build a circuit.
2. Switch to Simulate mode by selecting the `[Simulate]` button.
3. Load input events by selecting `Simulate:Load Input Events` from the menu.
4. Run a simulation using these input events by selecting the `[Run]` button. The circuit generates annotation requests, though there are no annotation responses to match them.
5. Save the output log by selecting `Simulate:Show Output Events-> Save`. The output log contains the annotation requests.
6. Using a text editor, change all occurrences of % anno:request: to % `anno:response:`. This will change the requests to matching responses. Now you can hand craft the annotation responses to

match what you expect your annotation server to reply. If you want to simulate transit delays between the request and response then add the number of seconds delay to the end of the line. For example, to simulate a 12 second transit delay before the response:

```
% anno:response:12
```

Save the response log.

7. Reset the simulation by selecting the [Reset] button, and load the response log by selecting Simulate:Load Annotation Events.
8. Start the simulation by selecting the [Run] button. The input events should enter the circuit and generate annotation requests. These requests will match the appropriate responses out of the annotation responses log, but the responses will not enter the circuit until the transit delay has expired.
9. If annotation responses arrive after the last event in the input log, select the Step-by [Time] button to see the responses enter the circuit. You may find it convenient to set a breakpoint on the Annotate node and set a large value for the time.

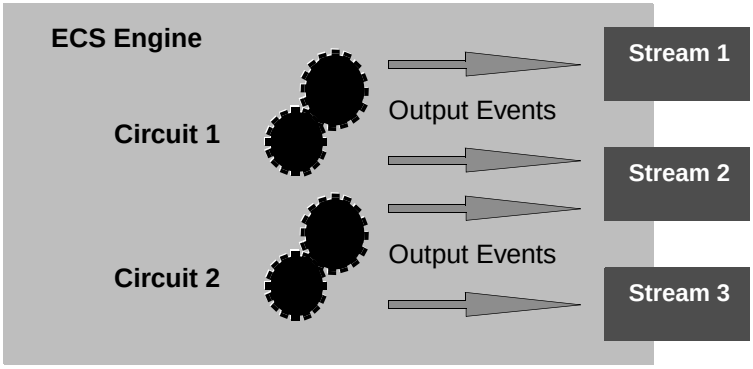
6 Drill Down

This chapter discusses the drill environment in HP OpenView Event Correlation Services Engine. The process is illustrated with two simple approaches that highlight the issues involved.

Drill Environment

In ECS every circuit is associated with a output stream. A circuit can be enabled on more than one stream at a time.

Figure 6-1 Drill Record



Every stream has associated with it two files, namely

- drill information log
- drill event log

The drill information for an event flowing through a circuit is specified by the circuit designer within the node's condition parameter. The engine stores this information in either an application specified Correlation log (drill information log) or the default Correlation log. The events which are refered from within the drill information are logged seperately in drill events log. Like Correlation log, drill event log can be application specified or default.

Drill logging can be enabled or disabled with the help of `ecsmgr` commands as listed below:

By default ECS uses the following files to log correlation info and drill event.

`$OV_LOG/ecs/<instance>/drill_info.log0` and
`$OV_LOG/ecs/<instance>/drill_event.log0.`

By default logging is disabled and can be enabled through `ecsmgr` interface.

The application can also register their own drill logs through `ecsmgr` interface. Drill logging for a stream by default is disabled.

Create a stream

To create a new stream :

`ecsmgr -create_stream stream_name`

where *`stream_name`* is the name of the stream created by user.

Enable default drill logging

`ecsmgr -log_drill_info on`

`ecsmgr -log_drill_event on`

Enable drill logging on new stream

`ecsmgr -log_drill_info stream stream_name on`

`ecsmgr -log_drill_event stream stream_name on`

Register drill log files on new stream

`ecsmgr -stream stream_name -drill_info_log <path>`

`ecsmgr -stream stream_name -drill_event log <path>`

Drill Record Format

Drilldown and drillup information is logged in drill information log in the following format

```
ueid : ueid -> relation -> relation.... : ueid -> relation -> relation .... :: ueid ->relation -> relation .... : ueid -> relation -> relation ....
```

Drill events are logged in the drill event log in the following format

```
ueid : textevent :: ueid : textevent :: .... ueid : textevent
```

The Correlation log will have the complete drill information of an event logged as a tree. Drill information is separated by the first ':'. A correlating event and its relationship forms a corrleation tuple. The correlated tuples are seperated by ':' and event and relationship are separated by "->". The drill-up and drill-down information are seperated by '::'. In the specified drill information format the drill-up information will always follow the drill-down information. For example

```
e1:e2->relation_string:........::e5->relation_string
```

where e1,e2 etc are the unique event-ids generated by the ECS engine during the event flow.

In the above example , e1 is the correlated output event whose drill information follows after the first ':'. The tuples e2, relation_string can repeat seperated by ':'. There may be no drill information for e1 which will mean that e1 is output without being correlated or the circuit designer did not choose to put any drill information for this event. the tuples after '::' specify drill-up information. For example , e1 is a correlating event of e5 and any other events that follow.

An event in the drill event log will be logged exactly once. The relation_string can be any arbitrary alpha-numeric string. This string is provided to indicate the relationship of e1 with event e2.

The user can specify the relationship string which may best indicate the relationship. For example `e1:e2,"suppressed"`.

The engine will create and maintain a datastructure on a per event basis to record the drill information tree. The memory will be free once the event is either deleted or output.

Logging of Drill Information

The Drill information tree will be logged in the Correlation log and the events refered from the drill information will be logged in the drill event log. The engine will itself provide a default Correlation log and a default drill event log.

Correlation and drill event logs can be registered for a stream. Both these logs can be registered independently. An application may also choose to work with default logs.

The physical logging of both the logs will be done by the engine at the time of either outputting the event to the stream or deleting the event from the engine .

ECDL built in functions for capturing the drill information

The drill information for an event can be added to an event from within any node within the node's condition parameter.

append <event 1> <event 2> <relation string>

The above function is used to add to correlation information to an event. It takes the correlated event, correlating event and the relation string as parameters. In the above example event 1 is the correlated event and event 2 is the correlating event.

A drill information can be added more than once to an event. A correlation tuple can be appended to an event more than once by two nodes or two different circuits. For example if the engine puts

```
e1:e2->string 1->string 2:.....
```

e2 correlates to e1 with the relationship "string 1" in one circuit and in another circuit e2 correlates to e1 with another relationship specified as in "string 2".

There maybe conditions when the user is sure that the event is going to be stored within the circuit forever then it is required to flush this event to guarantee the singleness of the log. The general format of this API is

flush <event>

where event is the event that is to be logged.

Since the output streams cannot be determined in advance for an event being flushed, the drill information of the event will be logged to all the streams where the circuit is enabled.

NOTE flush can also be used on events if there cannot be any more appends happening with the event. This will force drill logging of the event. As flush forces drill logging for an event, using append with the event already flushed should be avoided. If used it will result in duplicate correlation records (in Correlation log files) for the event

Drill API

Drilling APIs are provided for easy, off-line reading of drill log files.These APIs call drill logging functions for reading, if they are available. Following are the APIs used for reading of the drill logs.

EDI_initDrilling

Function

int32 EDI_initDrilling()

It initializes drill log browsing. This should be the first function called by drilling API called by any application. This should be called only once, even for browsing multiple drill log files.

Input Parameters

None

Return Parameteres

`ECS_SUCCESS`, if successful

`EDI_INIT_ALREADY_DONE`, if alsready intialized

`EDI_MALLOC_FAILED`, if memory allocation failed

WARNING

The above function must be called before using any other drilling functions.

EDI_openDrillLog

Function

```
int32 EDI_openDrillLog
        ( char *drillInfoLog,
          char *drillEventLog
        )
```

It opens drill log files specified by the user. The log files should be in ECS engine's drill log format.

One drill log file can be opened by passing NULL for the other parameter. Also, multiple files can be opened by calling this many times in the same session.

If both drill log files are mentioned, they should be related ie, should have been created on the same stream and/or engine.

Input Parameters drillInfoLog - drill info log file name

drillEventLog - drill event log file name

Return Parameters id, if succeeful id >=0 ,else one of the following errors is returned

EDI_INITNOTDONE, if drilling is not intialized

EDI_INVALID_PARAMS, if both drillInfoLog and drillEventLog are null

EDI_MALLOC_FAILED, if memory allocation failed

EDI_CANNOT_OPEN_FILE, if log files could not be opened

EDI_PARSE_ERROR, if the log files have invalid format

EDI_INTERNAL_ERROR, in case of other errors

EDI_closeDrillLog

Function int32 EDI_closeDrillLog(int32 id)

Closes previously opened drill log files

Input Parameters id - drilling id (id returned by EDI_openDrillLog)

Return Parameters ECS_SUCCESS, if successful

EDI_INITNOTDONE, if drilling is not initialized

EDI_INVALID_PARAMS, if the id is invalid

EDI_resetDrilling

Function int32 EDI_resetDrilling()

Resets and cleans-up drill log browsing session. It closes all opened drill log files and sets the drilling session to the initial state.

Return Parmeters ECS_SUCCESS

EDI_getDrillInfo

Function

```
int32 EDI_getDrillInfo
        ( int32 id,
          EDI_UEID *ueid,
          char **drillInfo
        )
```

Reads the drill information from drill info log file. The memory for the drill info record in allocated in the drillInfo parameter and returned to the calling application. The application has to free the memory returned in drillInfo parameter.

Input Parameters

`id` - drilling id

`ueid` - ueid of the event whose record has to be read

`drillInfo` - pointer to hold the record

Return Parameters

size of drill record (>=0, if successful)

`EDI_INVALID_PARAMS`, if id is invalid or drillInfo is invalid

`EDI_INITNOTNODE`, if drilling is not intialized

`EDI_BAD_PARAMETER`, if ueid is not valid

`EDI_INTERNAL_ERROR`, if any other error

EDI_getNextDrillInfo

Function

```
int32 EDI_getNextDrillInfo
        ( int32 id,
          EDI_UEID *ueidBase,
          EDI_UEID *ueidNext,
          char **drillInfo
        )
```

Reads the drill information record from drill info log file. The memory for drill info record is allocated in the drillInfo parameter. The application has to free the memory returned in drillInfo parameter.

Unique event id of the event whose record is read is returned in ueidNext parameter. This can be used for sequential reading of drill info records.

Input Parameters

`id` - drilling id

`ueidBase` - ueid of the event whose next record has to be read

`ueidNext` - ueid of the event whose record is read

`drillInfo` - pointer to hold the record

Return Parmaters

size of drill record (>=0, if successful)

`EDI_INVALID_PARAMS`, if id is invalidor drillInfo is invalid

`EDI_INITNOTNODE`, if drilling is not intialized

`EDI_BAD_PARAMETER`, if ueid is invalid

`EDI_EOF`, if end of fifle is reached

`EDI_INTERNAL_ERROR`, if any other error

EDI_getDrillEvent

Function

```
int32 EDI_getDrillEvent
                ( int32 id,
                  EDI_UEID *ueid,
                  char **drillEvent
                )
```

Reads the event from drill event log file. The memory for drill event record is allocated in the drillEvent parameter. The application has to free the memory returned in drillEvent parameter.

Input Parameters

`id` - drilling id

`ueid` - ueid of the event whose record that has to be read

`drillEvent` - pointer to hold the record

Return Parameters

size of drill event record(>=0, if successful

`EDI_INVALID_PARAMS`, if id is invalid or drillEvent is invalid

`EDI_INITNOTDONE`, if drilling is not intialized

EDI_BAD_PARAMETER, if ueid is invalid

EDI_INTERNAL_ERROR, if any other error

EDI_getNextDrillEvent

Function

```
int32 EDI_getNextDrillEvent
                ( int32 id,
                  EDI_UEID *ueidBase,
                  EDI_UEID *ueidNext,
                  char **drillEvent
                )
```

Reads next event from drill event log file. The memory for drill event log is allocated int eh drillEvent parameter.The application has to free the memory returned in drillEvent parameter.

Unique event id of the event whose record is read is returned in ueidNext parameter. This can be used for sequential reading of drill info records.

Input Parameters

`id` - drilling id

`ueidBase` - ueid of the event whose next record has to be read

`ueidNext` - ueid of the event whose record is read

`drillEvent` - pointer to hold the record

Return Parameters

size of drill event record(>=0), if successful

EDI_INVALID_PARAMS, if id is invalid or drillEvent is invalid

EDI_INITNOTNODE, if drilling is not intialized

EDI_BAD_PARAMETER, if ueid is invalid

EDI_EOF, if end of file is reached

EDI_INTERNAL_ERROR, if any other error

Custom Logging framework

By default, the drill logging of events takes place through flat files. The user may create a mechanism to capture drill information according to his needs, for example to use a database.

The custom drill logging mechanism can be supplied to ECS using a shared library. If specified all drill logging will happen by the calling function from the specified framework. The library is placed under `$OV_CONF/ecs/drilllog directory and with libEDICL.sl name`. This library should contain user functions for initializing, resetting, storing drill info and storing drill event.

The custom logging framework is initialized by calling the function `EDI_INITIALIZE_FNT` from the shared library. This function should return the other three custom logging functions to the engine. These functions get called on a per event basis.

The custom logging functions are listed below along with their functionality.

EDI_INITIALIZE_FNT

Function

int32

EDI_INITIALIZE_FNT

(int32 instance,

EDI_Definition *defn

)

Initializes the custom logging framework. This is the first function called by engine in the custom logging framework. Typically all initialization required for custom logging should be done in this function. This is called only once by an engine.

User has to supply other logging function in defn parameter like,

defn->writeDrillInfo = mywriteDrillInfo;

defn->writeDrillEvent = mywriteDrillEvent;

defn->reset = myreset;

where mywriteDrillInfo, mywriteDrillEvent and myreset are user functions for custom logging.

Input Parameters engine instance - the instance number of the engine calling the function.

defn - custom logging definition structure. Contains pointers to custom logging function.

Return Parameters ECS_SUCCESS, if successful

ECS_ERROR, in case of error

reset

Function int32 reset (int instance)

resets the custom logging framework. This is the last function called by engine to reset custom logging. Typically, all reset operation including closing of files should happen here. This is called only once by an engine.

Input Parameters engine instance - the instance number of the engine calling the function.

Return Parameters ECS_SUCCESS, if success

ECS_ERROR, in case of error

writeDrillInfo

Function int32 writeDrillInfo

```
( int32 instance,
  int32 nstrm,
  char** streamName,
  EDI_UEID *ueid,
  int32 nDrillDown,
  EDI_CorrelInfo *drillDown,
  int32 nDrillUp,
  EDI_CorrelInfo *drillUp
)
```

This user supplied function is used for storing drill information of an event. Event's unique id is passed in ueid parameter and drill down and drill up informations are contained in drillDown and drillUp parameters. The streams which output the event are passed in streamName parameter.

Unlike drilling APIs, this function should not freeup memory of any parameter. If parameters are freedup then the behaviour is undefined.

NOTE StreamNames parameter will contain only the names of streams on which drill info logging is enabled or NULL. A stream name of NULL indicates engine's default drill info logging is enabled.

Input Parameters

instance - engine's instance number

nstrmNames - number of stream names contained in strmNames parameter

strmNames - array of stream names, where the event is output

ueid - events unique event id

nDrillDown - number of drilldown info contained in drillDown param

drillDown - array of drilldown info for the event

nDrillUp - number of drillup info contained in drillUp param

drillUp - array of drillup info for the event

Output Parameters

ECS_SUCCESS, if successful

ECS_ERROR

writeDrillEvent

Function

```
int32 writeDrillEvent
                ( int32 instance,
                  int32 nstrm,
                  char** streamName,
                  EDI_UEID *ueid,
                  char *textevent
```

)

This user supplied function is used for storing text format of an event. Event's unique id is passed in `ueid` parameter and text format of the event is contained in textevent parameter. Parameter `streamname` contains the list of streams where the event has to be stored. This list may include streams that did not output the event. This is required to preserve the completeness of drill logs. (If an event A has participated in the correlation of another event B which is output on a stream S, B's drill information is logged in stream S's log file. Both events are logged to drill event log, even though A is not output on S, as drill information has reference to both).

Unlike drilling APIs, this function should not freeup memory of any parameter. If parameters are freedup then the behaviour is undefined.

NOTE StreamNames parameter will contain only the names of streams on which drill event logging is enabled or NULL. A stream name of NULL indicates engine's default drill event logging is enabled.

Input Parameters

instance - engine's instance number

nstrmNames - number of stream names contained in strmNames parameter

strmNames - array of stream names, where the event is output

ueid - events unique event id

textevent - text format of the event.

Return Parameters

ECS_SUCCESS, if drill event is successfully logged

ECS_ERROR, in case of error.

7 ECDL Enhancments

This chapter discusses the ECDL enhancements in HP OpenView Communications Event Correlation Services Engine. The following features are described:

Circuit Serialization

The Engine performs specific correlation actions on an event stream, resulting in a stream of correlated events. Events emitted from the engine can be fed to other circuits. Events emitted from one circuit can be fed to the input node of other circuits. This feature of feeding of events emitted from the engine to othere circuits is called Circuit Serilaization.

The events that are to be fed back into the engine are done using the `feed()` API. The `feed()` API is provided to identify the events that will be sent to other circuits. `feed()` is provided to be used in the configuration of any node. The feed() API would have representation as below:

```
feed [true/false]
```

where <event> is a valid event

The second argument to the feed() API is optional. The default value is "`false`". In this case the event is fed back to all other circuits except the source circuit.

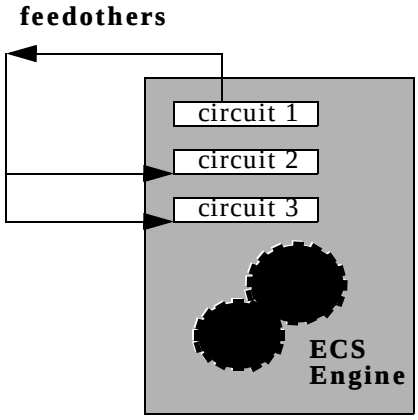
If the argument is"`true`" then the event is fed back to all circuits including the source circuit.

To ease the use of the `feed()` API, two new APIs have been added

- `feedall <event>`
- `feedothers <event>`

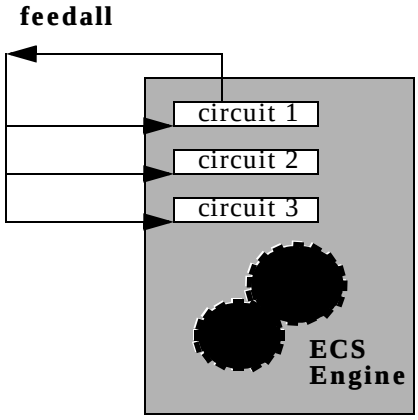
The `feedothers ()` API feeds back events to all circuits in the same engine excluding the source circuit which sends the event.

Figure 7-1 Circuit Serialization using `feedothers` API



The `feedall ()` API feeds back events to all the circuits in the same engine including the source circuit which sends the event.

Figure 7-2 Circuit Serialization using `feedall` API



When the `flagToSrcCircuit` is set to `"true"`, proper conditions must be given to avoid the continous feedback of events into the source circuit. This could lead to an infinite loop being created.

The designer can load only one circuit at a time and hence the circuits with circuit serialization cannot be simulated in the designer.

Multiple Event Creation

ECS supports the creation of multiple events from a single create node.The list of events is passed as a tuple list to the create node. The create_events API is of the form

create_events [(encoding type,event syntax), (encoding type,event syntax), ...]

Example 7-1 create_event ECDL function

```
create_events
[("ber",1.3.6.1.4.1.11.2.2.6.6),("ber",Trap_PDU)]
```

The create_events returns a list that is modified and then sent to the engine.This API is used in the create spec of the create events. The create spec of the create node to create multiple events may look like the following:

```
let
  --define a list if tuples to create events using create_events
  val evtype = [("ber",1.3.6.1.4.11.2.2.6.6), ("ber","Trap-PDU"),
("mdl","SimpleEvent")]
  -- Create events by passing evtype as parameter
  -- the returned value is assigned to events, which will     contain [cmip
event,snmp event, mdl event]
     val events = create_events evtype
  -- modify the cmip event, for simplicity alter spec is left out
  val _ = (nth 1 events) alter (......)
  -- modify the snmp event, for simplicity alter spec is left out
  val _ = (nth 2 events) alter (......)
  -- modify the mdl event, for simplicity alter spec is left out
  val _ = (nth 3 events) alter (......)
  -- return the created events to engine
in
  events
```

```
end
```

The statement *events* in the above example denotes that the engine list is stored as a whole and is returned to the engine.

Modification of Event List

The event list has to be modified before it is fed into the engine. There are a set of ECDL functions that can be used to perform the modification.

- *foldl* that has the iteration from left to right
- *foldr* that has the iteration from right to left

The events that have been created in the previous section can be modified as shown below:

```
let
  -- define a list of tuples to create events using create_events
  val evtype =
[(“ber”,1.3.6.1.4.11.2.2.6.6),(“ber”,”Trap-PDU”),(“mdl”,”SimpleEvent”)]
  --Create events by passing evtype as parameter
  -- the returned valuse will is assigned to events, which will contain [cmip
event, snmp event,mdl event]
   val events = create_events evtype
  -- function for iterating over elements in the event list
   fun loopThru element el_list =
    let
       val syntax = element “event_syntax”
    in
      choose syntax of
      1.3.6.1.4.1.11.2.2.6.6 => element alter (....)
      -- alter spec for cmip events
      |”Trap-PDU” => element alter (....)
      -- alter spec for snmp events
      | “SimpleEvent” => element alter (....)
      -- alter spec for mdl events
```

```
    end
  end
  -- modify new enents
val_ = foldl loopThru 0 events
  --return the created events to engine
in
  events
end
```

Engine Flow

Once the events have been modified they have to be sent to the engine. The ECS engine will further enhance it, and then outputs it. The order in which the events are output is in the same order in which it appears in the list.

If the create spec returns an event list [ev1,ev2,ev3,ev4]

- ev1 will be sent out of create node first. This event travels through list of nodes connected to output port of the create node till it reaches a strorage (in unless node, table node etc) or it reaches an output node.
- ev2 is taken for processing and it is sent out of create node.
- this flow continues till end of list is reached in this case ev4.

Tracing and Logging

The ECS Engine can log events as they arrive at the engine, as they leave a specific stream or a correlation circuit. Events that fail to enter a circuit within a stream can also be logged. Each Event log contains the ASCII representation of the logged events in the form that can be edited and displayed using text editing tools. Error messages can be logged from the ECS Engine to the engine log file, and trace the internal operations of the ECS Engine to the engine trace file.

The ECS Engine maintains two logs to

- Trace Log
- Engine Log

The Trace log captures the following kind of information

- Automatic logging of circuit functionality such as event flow, event and node creation and deletion.
- String output

The engine log captures the following types of information

- Automatic logging of Errors and Warnings for the engine operations. In case of errors from within a node the error messages with the decoded events are also logged.
- Audit logging as specified by the Circuit Designer. Audit log can be specified in any node using the ECDL built-in function. Refer to the *HP OV ECS Designer's Reference Guide* for more information.

In a real time scenario, there could be a situation where in the user feels the need to check if the circuit is working suitably. The user can then debug the circuit. The contents of the node can be extracted using the function `System.node_dump`.

For Example a node dump within a Table node will dump the table persistent data available at the time of a trigger. Similarly the Combine node would generate all the events queued up waiting for combine.

The System.node_dump does not take any parameters. It performs a node level dump and generates node specific data into the Trace log.

The System.node_dump could look like

```
let
        val _ = System.trace("BEGIN_LOG_&_TRACE")
        val _ = System.node_dump()
        val _ = System.trace("END_LOG_&_TRACE")
in
        false
end
```

Glossary

Abstract Syntax Notation 1 (ASN.1) An OSI standard related to the Presentation Layer where the abstract representation of the data is independent of its physical encoding. It is specified in ISO/IEC 8824, X.208.

agent A program or process running on a remote device or computer system that responds to management requests, performs management operations, and/or sends event notifications.

annotation API A set of application program interface functions and data structures that supports the transfer of data between an external annotaton server and one or more Annotate nodes in an ECS circuit.

annotation server A user supplied server that receives a request from an Annotation node within a correlation circuit, performs some action, and returns a response to the Annotate node. The action performed by the annotation server may involve information extracted from events in the circuit, and the information returned is typically obtained external to the ECS Engine and the annotation server.

arrival time The time an event arrives at the ECS engine in Universal Coordinated Time (UTC).

ASCII American Standard Code for Information Interchange. A standard used by computers for interpreting binary numbers as characters.

ASN.1 Abstract Syntax Notation 1.

attribute An object characteristic or property that describes the current state of the object and which has a unique identifier by which it is accessed. In ECS, for example, the “eventTime” attribute of a CMIP event, or the “Rate” attribute of a Rate node. See event attribute; identifier; correlation node attribute.

attribute-value pair The combination of an attribute identifier and the value of that attribute for a specific object. In ECS, attribute-value pairs are represented as key-value pairs in an ECDL dictionary. See also key-value pair; dictionary.

Basic Encoding Rules (BER) Defines how ASN.1 data types are encoded for transport on the network.

breakpoint A point in a program at which execution is halted so that the program’s status, contents of variables and other factors can be examined. In the ECS Designer, in simulation mode, breakpoints are locations in a correlation circuit where event processing is halted to allow for manual intervention.

canvas The working area of the ECS Designer screen. This is where you place, connect, and configure correlation nodes to create your correlation circuit.

CCITT The International Telegraph and Telephone Consultative Committee, an international organization concerned with proposing recommendations for international communications. Replaced by the International Telecommunications

Union, Telecommunications (ITU-T) in 1992. See International Telecommunications Union, Telecommunications (ITU-T).

circuit *See See correlation circuit.*

CMIP *See See Common Management Information Protocol (CMIP).*

Common Management Information Protocol (CMIP) A protocol for exchanging network management information in an OSI environment (ISO/ITU-T X.710). CMIP communicates management information between a manager and an agent. CMIP allows a manager to retrieve (get) management information from, or to alter (set) management information on an agent. CMIP also allows the manager to create and delete instances of an object managed by the agent, or perform an action on an object. An agent can also emit unsolicited messages, called notifications, to alert managers of noteworthy local conditions.

component event An event that is combined with other events to create a new event. In ECS, a composite event is composed of two or more component events. See composite event.

composite event In ECS, a composite event consists of a structured aggregation of addressible component events each of which may be a primitive event, a temporary event, or a composite event. A composite event may only exist within a correlation circuit. See also component event; primitive event; temporary event.

compound node A graphical element that represents a container of lower level components. The lower level components will be displayed when the user opens the compound node. In ECS, a correlation circuit fragment may be encapsulated in a compound node, hence creating a new user-defined correlation node. Compound nodes may be added to libraries and re-used by reference or by copy. Compare with primitive node.

condition (parameter) In ECS, a condition is an ECDL expression specified for a correlation node parameter, usually involving attribute from an event, that returns a value used to modify the behavior of the correlation node.

correlation A procedure for evaluating the relationship between sets of data or objects to determine the degree to which changes in one are accompanied by changes in the other. In ECS, correlation is a process of analyzing a stream of events by filtering and detecting patterns and replacing groups of events with single events that have (possibly) higher information content.

correlation circuit In ECS, a collection of interconnected primitive nodes and compound nodes, configured to perform a filtering or correlation activity. Each correlation node is configured appropriately to the correlation requirement. The configuration includes the specification of the event types, and the allowed transit delays for those events, to be accepted from the external event stream. A correlation circuit can be loaded into an ECS Engine.

correlation circuit port The logical connections between a correlation circuit and the containing infrastructure where events enter and leave the circuit. These ports may be configured to select a subset of events in the input event stream, based upon event encoding type and event syntax. A single port may be connected to multiple Source/Sink nodes, and a single Source/Sink node may be connected to multiple circuit ports.

correlation engine The ECS runtime component that reads an input event stream, decodes the input events, performs the event correlation, encodes the output events and returns the output events to the event stream. The event correlation is as specified by the one or more correlation circuits loaded into the correlation engine.

correlation node A processing element in a correlation circuit. See also compound node; primitive node.

correlation node attribute A property of a correlation node that can be read from another correlation node. The Count, Rate, and Table nodes have attributes (which may be exported by a containing compound node as attributes of the compound node). Attributes are addressed using a dot notation: “node_name.attribute_name”.

correlation node parameter In the ECS Designer, a correlation node parameter is an ECDL expression used to configure a correlation node.

correlation node port One of possibly many connection points of a correlation node used to interconnect correlation nodes. Events enter a correlation node through a port and leave a correlation node through a port. Port types include input, output, control, reset, and error ports. In the ECS Designer, ports visually indicate the sense of the associated event flow. Optional ports are not displayed by default.

creation time The time an event was created. Inside the ECS Engine creation time is represented in Universal Coordinated Time (UTC).

daemon A process that “serves” clients. Sometimes referred to as a server.

data store In ECS, a component of the ECS Engine which holds user-specified named data items of an ECDL data type. The entries in the data store may be referenced from the ECDL expressions configured into the correlation nodes. A correlation circuit may be associated with one of the possibly many data stores loaded into the correlation engine.

data type A particular kind of data; for example integer, alphanumeric, boolean, date. In ECS, data types are ECDL data types which define the type and range of values to which an identifier may be assigned. Every value in ECDL has a data type, but the type need not be explicitly stated. The types range from simple types such as integers, to compound types such as dictionaries and lists, and special types such as functions and events.

dictionary (data type) In ECS, a dictionary is an ECDL data type comprised of an unordered list of key-value pairs. Any value is accessed via reference to the key.

Within ECS, an event is treated as a dictionary with attribute names being the dictionary keys which provide access to the attribute values.

Distributed Management Platform (DM) HP OpenView Communications Distributed Management Platform, the platform which provides the infrastructure for implementing OSI-based management solutions.

DM *See See Distributed Management Platform (DM)*

duration data type In ECS, a duration is an ECDL data type used to represent relative or elapsed time values. Compare with time data type.

dynamic parameter A parameter whose value is determined during program execution. In ECS, an ECDL expression configured for a correlation node parameter which is evaluated each time an event enters the correlation node. Typically, the value returned by a dynamic parameter changes for each event processed.

ECDL *See See Event Correlation Description Language (ECDL).*

ECS *See See Event Correlation Services (ECS).*

ECS circuit *See See correlation circuit.*

ECS Designer The ECS Designer is the ECS component which you use to create and test correlation circuits. The ECS Designer works in two modes: build mode where you create correlation circuits, and simulate mode where you test the circuits.

ECS Engine *See See correlation engine.*

ecsmgr The command line program used to administer a running ECS Engine.

endecode In ECS, a term used to refer to a combined encoding or decoding function or capability. An endecode module is an architectural entity which provides encoding and decoding for a specific type of event.

evaluation license A license granted for a specific period of time for the purpose of evaluating ECS.

event An event is an unsolicited notification such as an SNMP trap, a CMIP notification, or a TL1 event, generated by an agent process in a managed object or by a user action. Events usually indicate a change in the state of a managed object or cause an action to occur. In ECS, an event is encoded as a primitive, compound, or temporary event. ECS events contain header attributes added to the input events to assist the processing of the events while they are in the ECS correlation circuit. The header attributes are stripped before the events are transmitted from the ECS circuit.

event attribute A characteristic property of an event. In ECS, event attributes are either part of the internally created event header common to all event types, or part of the event body that contains the input event.

Event Correlation Description Language (ECDL) The language used to specify correlation circuits (node relationships, parameter expressions, data and fact store values) for the ECS Engine.

Event Correlation Services (ECS) The HP OpenView Communications Event Correlation Services product.

event encoding type The first and highest level in the three-tiered ECS event classification system. An event's encoding type determines the endecode module that will be used to translate the event to and from its native format. For example, CMIP notifications and SNMP traps both use the BER encoding type. ASCII events use the MDL encoding type, and OVO messages use the OVO encoding type. See also event syntax; event type

event flow An ECS circuit represented graphically as a circuit schematic consisting of correlation nodes interconnected by lines (connections). See also correlation circuit.

event body The body of an event depends on the event class. The body of a primitive event is the original message, trap or event; the body of a temporary event may be empty; and the body of a composite event consists of other events.

event header Inside ECS and event is augmented with additional information such as the event encoding type, event syntax, event type, and event class. This information is carried in a header that is attached to the event body. See also event body.

event I/O API A set of application program interface functions and data structures that supports the input and output of events to and from the ECS Engine.

event syntax The rules governing the structure and content of an event. In ECS, the event syntax is the second level in the three-tiered ECS event classification system. An event's syntax determines how the event's attributes are read and written. For example, SNMP traps have an event syntax of Trap-PDU and CMIP notifications have an event syntax that evaluates to an OID identifying the GDMO notification. ASCII events have a syntax determined by the MDL definition used to read and write them. See also event encoding type; event type.

event type A classification of an event into a particular category that further defines the nature of the event. In ECS, the event type is the third and lowest level in the three-tiered event classification system. The event type is represented by the ECS header attribute "event_type". For SNMP traps the event type is the generic trap number (1-6). The CMIP event type is the OID of the notification. ASCII events have an event type determined by the MDL definition used to read and write them. See also event encoding type; event syntax.

expiry time Annotation requests are valid for a limited time, determined by the Annotate node's Time Limit parameter. The expiry time is the time at which the annotation request was generated plus the Time Limit. In other words, it is the time at which the request expires.

expression In general, a set of reserved words, symbols, variables, and functions that is evaluated to provide a result. In ECS, an expression is any collection of valid ECDL statements. Note that ECDL is a functional language that has no concept of variables.

fact store A component of the ECS Engine which stores relationships between objects. Any two objects which may be any ECDL data type, may be related using any user-defined relationship. The facts may be accessed at runtime by the ECDL expressions configured into the correlation node parameters.

FLEXlm A Licensing technology used in stand-alone and DM-integrated ECS products.

floating license A license where there is a single license server for all licensing clients on the network. Any licensing client on the network can access the license server to check out a license.

function A general term for a portion of a program that performs a specific task. In ECS, an ECDL function is one of the built-in functions or operators, or a user defined function. ECDL functions can be named or anonymous, but must return an ECDL value.

GDMO See Guidelines for the Definition of Managed Objects (GDMO).

Greenwich Mean Time Standard time used throughout the world based on the mean solar time of the meridian of Greenwich. See Universal Coordinated Time (UTC).

Guidelines for the Definition of Managed Objects (GDMO) Describes a formal method for describing the important characteristics and operations of an object class. Specified in ISO 10165-4, X.722.

HP OpenView A family of network and system management products, and an architecture for those products. HP OpenView includes development environments and a wide variety of management applications.

identifier A name that within a given scope uniquely identifies the object with which it is associated.

IEC International Electrotechnical Commission.

IEEE Institute of Electronic and Electrical Engineers.

International Telecommunications Union, Telecommunications (ITU-T) The ITU is a world-wide organization within which governments and industry coordinate the establishment and operation of telecommunications networks and services. It is responsible for the regulation, standardization, coordination and development of international telecommunications as well as the harmonization of national policies. The ITU is an agency of the United Nations. In 1992 it took over the functions of the CCITT.

ISO International Standards Organization.

ITU-T International Telecommunications Union, Telecommunications.

key-value pair A data storage item consisting of a search key paired with a value. In ECDL, a key-value pair is written as “key => value”. See also dictionary.

library In ECS, a repository for compound nodes. Compound nodes in the library may be referenced from a circuit, or copied from the library and modified.

license The legal right to use a feature in a software program.

license server The server processes that manage access to ECS features by licensed users.

list data type a variable-length ordered set of values all of the same data type. In ECDL, a list data type may contain a set of values of any other ECDL data type including complex types such as lists and tuples.

Management Information Base (MIB) A logical collection of configuration and status values that can be accessed via a network management protocol.

MDL *See See Message Description Language.*

message description Detailed information about an event or message. In ECS, a description of the attributes and formatting of a text-based event (message), that allows the MDL endecode module to decode and encode events consistent with that syntax. Message descriptions which are written in Message Description Language (MDL) are translated into metadata before being used by the ECS engine endecode module. See metadata.

Message Description Language A language used to describe a text event’s attributes and formatting. Each text event syntax has its own message definition written in MDL. See also message definition; event syntax.

metadata Data about data. In ECS, message descriptions are translated into metadata which is a form which maximizes access performance by the MDL endecode module. See message description. CMIP and SNMP metadata is derived from MIBs.

MIB Management Information Base (MIB).

Network Node Manager (NNM)

Definition to come from OVSD.

NNM *See See Network Node Manager (NNM).*

node 1. A computer system or device (e.g., a printer, router, bridge) in a network. 2. A graphical element in a drawing that acts as a junction or connection point for other graphical elements. 3. In ECS, see correlation node.

nodelock license A license where the license server and license clients must be on the same machine, meaning that the licensed application is “locked” to running on the node that is the license server.

object identifier (OID) A unique sequence of numbers or string of characters used for specifying the identity of an object, that is obtained from an authorized registration authority or an algorithm designed to generate universally unique values.

OID *See See object identifier (IOD).*

oid data type In ECS, an oid is an ECDL data type which contains an Object Identifier in dot-separated notation (e.g., 1.2.3.4.5). Where the data item is dynamically interpreted, at least three elements (2 dots) are required to avoid interpretation as a real data type.

Open Systems Interconnection (OSI) A standardization model in which a manager process is responsible for executing specific management functions requested by the user through interactions with an agent process. The agent process represents the management services offered by the managed objects.

OSI *See See Open Systems Interconnection (OSI).*

OVO HP OpenView Operations, a distributed client/server software solution that helps system administrators detect, solve, and prevent problems occurring in networks, systems, and applications.

parameter *See In ECS, see correlation node parameter.*

pmd HP OpenView postmaster daemon.

port 1. A location for passing information into and out of a network device. 2. In ECS, a location for passing events into and out of a correlation node or a correlation circuit. See correlation node port; correlation circuit port.

primitive event An ECS internal event which encapsulates an input event. Several header attributes are added as a header for correlation and control purposes, which are stripped before the primitive event leaves the ECS engine. See also event; temporary event; composite event.

reserved word Words that have special meaning in ECS and cannot be used for any other identifier.

Simple Network Management Protocol (SNMP) The ARPA network management protocol running above TCP/IP used to communicate network management information between a manager and an agent. SNMPv2 has extended functionality over the original protocol.

simulate *See See simulation.*

simulation In general, the imitation by a program of a process or set of conditions affecting one or more objects such that the results of the program reflect the impact of the process or changes in conditions. In ECS, a simulation is the process of feeding events from an event log file through the correlation circuit to observe the behavior of the correlation circuit using aids such as breakpoints, tracing, and stepping.

SNMP *See See Simple Network Management Protocol (SNMP).*

SNMP trap An unconfirmed event, generated by an SNMP agent in response to some internal state change or fault condition, which conforms to the protocol specified in RFC-1155. See event.

socket stack An interface that supports interprocess communication based on the use of file handles. In ECS a socket stack is used to communicate with the ECS Engine for command, i/o and annotation purposes.

Software Distributor (SD) HP OpenView multi-platform software installation product.

static parameters In general, parameters whose values are determined prior to program execution. In ECS, a statically evaluated parameter is a correlation node parameter where the value is defined when the correlation circuit is loaded. The value does not change when an event enters the associated node/port. See dynamic parameters.

syntax In general, the rules governing the structure and content of a language or the description of an object. In ECS, see event syntax.

Telecommunications Management Network (TMN) The term used to identify a homogeneous approach to the management of heterogeneous networks. It is defined in the international standards referred to as ITU-TSS M3100. TMN recommendations incorporate OSI NM concepts, principles, protocols and application services.

temporary event In ECS, an event that is created transparently by particular correlation nodes, and which may exist only within a correlation circuit. Temporary events may consist only of header attributes created by the correlation engine, or they may additionally contain user data. Temporary events cannot be transmitted outside the correlation engine. See also event; primitive event; composite event.

time data type An ECDL data type that includes time and date.

TL1 Transaction Language One was developed by Bellcore and is a management system protocol that uses structured text messages to pass information about networks and network element states.

TMN See Telecommunications Management Network (TMN).

transit delay The difference between an event's arrival time and its creation time. Transit delays can be caused by external network delays or by deliberately introduced delays in an ECS circuit.

trap *See See SNMP trap; event.*

tuple data type An ECDL data type. A data structure consisting of a fixed collection of elements, where each element is a simple ECDL type or a complex ECDL data type.

Universal Coordinated Time (UTC)

Standard time used throughout the world based on the mean solar time of the meridian of Greenwich. Formerly known as Greenwich Mean Time (GMT).

universal pathname A set of environment variables that describe standard pathnames. Universal pathnames hide variations between pathnames on different versions of Unix.

UTC *See See Universal Coordinated Time (UTC).*

X/Open Management Protocol (XMP) An API specified by the X/Open standards body that provides a common access mechanism to both CMIS and SNMP management protocol services.

XMP *See See X/Open Management Protocol (XMP).*

Zulu *See See Universal Coordinated Time (UTC).*