Event Logging (POSIX 1003.25) Architecture Specification

An Intel Corporation White Paper

Version 1.0

David Brinkman,
Senior Software Engineer
Telco Server Development, Enterprise Server Group
Intel Corporation
October, 2001

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Copyright © Intel Corporation 2001
* Other names and brands may be claimed as the property of others.
Contents

CONTENTS................................................................................................................................. 2

INTRODUCTION ......................................................................................................................... 1

ARCHITECTURE COMPONENTS ................................................................................................. 1

REQUIREMENTS .......................................................................................................................... 3

SAMPLE CONTROL INTERACTIONS ......................................................................................... 3

FAULT MANAGER CLIENT TO HA DRIVER ................................................................................. 4
FAULT MANAGER CLIENT TO HA APPLICATION ........................................................................ 4
FAULT MANAGER CLIENT TO GENERAL SYSTEM EVENTS......................................................... 5

EXAMPLE API USAGE ............................................................................................................... 5

FAULT MANAGER CLIENT .......................................................................................................... 5
Sample API Code......................................................................................................................... 5

POSIX 1003.25 IMPLEMENTATIONS............................................................................................. 7

IBM IMPLEMENTATION ........................................................................................................... 8
IBM Features above and beyond POSIX 1003.25 ...................................................................... 9
INTEL IMPLEMENTATION ......................................................................................................... 12
API.......................................................................................................................................... 12
Daemon – API communication .................................................................................................. 13
Interaction with existing system logger ..................................................................................... 13
POSIX log writes and Query processing ................................................................................... 14

NEEDED IMPROVEMENTS TO THE IBM IMPLEMENTATION .................................................... 14
IBM GLIBC changes not possible .............................................................................................. 14
Remote Event Logging............................................................................................................... 15
Introduction

Event delivery with a persistent store of the events is a requirement for several of the projects for Telco Linux, and is a key enabling building block for other HA and Telco functionality. The POSIX 1003.25 standard contains a specification for an Event Logging facility that meets these requirements. The Event Logger, described in this document, uses the POSIX industry standard APIs for clients to register for events, receive and log event information from the kernel and user applications, and to deliver event notification to the clients that registered for them. Events are kept in a persistent log that can be viewed and managed. Tools and APIs are provided for displaying, registering for and taking actions when events are entered, and managing events in the event log.

Architecture Components

- **Client Applications** are user space processes that register with the Event Logger to receive events when they are entered in the event log. They register for events and receive event notification using the POSIX Event Log APIs. They may be fault managers who monitor kernel subsystems (i.e. SCSI or Ethernet) or user mode subsystems and applications. They may also be any other applications that require the ability to receive events about the system.

- **Subsystems** are kernel or user subsystems, i.e. SCSI in the kernel or CORBA as a middleware subsystem. This may also be an application that provides a service in the implementation. These subsystems will have a unique tag for identity in the event log records, and will use the kernel or user mode event APIs to log events. Note that this can include subsystems that do not use the event APIs but use the standard kernel printk() or libc syslog() functionality to log errors; in these cases syslog() has been enhanced to create more general event records that can still be used by client applications to receive events.

User mode subsystems use the POSIX Event Log specified APIs to create event records. The kernel event APIs currently are an extension of the standard that will be hopefully be included in a future version of the standard.

The following diagram shows the architecture of the POSIX Event Logging feature.
Event logging provides kernel interfaces for creating event log records that contain the content required by the POSIX specification. This requires additional kernel APIs to report event information, in addition to the existing kernel printk() and C library syslog() message reporting. Additional information (i.e. subsystem id and priority) is required and populates the event records using these APIs and should be used by drivers and other system components that will provide event information to client applications.
In addition to kernel events, events from user mode applications and middleware can be entered using the POSIX specified APIs. HA applications that report events to a service manager can enter events to the logger that the manager has registered for and receives. Additionally there is a path to enter events from other system utilities and applications that call the C library syslog() function to log errors. The Kernel Resource Monitoring and Kernel Message Monitoring projects use the event log to distribute operating system fault and resource-monitoring events to registered client applications.

**Requirements**

- A publish/subscribe style interface for communicating events from platform management system to interested software
- Standards-based client registration and delivery of events
- A common format of event messages for all events
- Asynchronous generation of event messages when they arrive rather than requiring software to poll devices to learn of events
- Automatic storage of events in non-volatile memory so they are not lost if there is no current event listener
- System-wide synchronized time-stamp on events
- Common classification system for events to identify severity, urgency, etc., of event
- Inclusion of enough information with an event to permit immediate fault management action as well as later analysis for the purpose of root cause analysis on system faults.
- Adherence to the current POSIX 1003.25 Event Log specification (http://evlog.sourceforge.net/posix_evlog.html)

**Sample Control Interactions**

The following diagram presents the control flow for system component interactions with the Event Logger:
This presents a typical session for a fault manager application that manages a HA aware driver in the kernel, which uses the kernel Event APIs to report critical events it sees.

1. The fault manager client application registers with the Event Logger to receive events for the driver’s subsystem, using the POSIX 1003.25 Event Log APIs.
2. When an event occurs in the HA driver it uses the kernel Event APIs to report the event. The Event Logger logs the event information.
3. The Event Logger delivers the new event to all applications that have registered for it along with the POSIX event record that contains the event information from the driver.
4. The fault manager takes appropriate actions to try to resolve the fault or escalate the error and update the system model.

**Fault Manager Client to HA Application**

A similar event delivery path is available to HA aware applications that directly call the POSIX Event Log APIs to report events that they encounter.
Fault Manager Client to General System Events

Other kernel and user mode system errors that do not use the Event APIs are logged using the standard kernel error reporting (i.e. printk()) and/or the syslog() API. The syslog() functionality has been enhanced to additionally enter these events into the Event Log so that they can be delivered to clients that register for them.

Example API Usage

Fault Manager Client

In this example, the fault manager is responsible for the recovery or restart of an H323 IVR and an H323 Gatekeeper. These applications normally fail in one of two ways. A mutex deadlock in the H323 library causes the applications to log an error before they attempt to handle the problem themselves. More often than not, they cannot handle the condition and the process hangs. The next most common failure is a crash of the application before an error message can be logged.

To handle these known problems, the fault manager performs two registrations with the POSIX event log daemon (one for each application). Each registration provides a query string that shows the unique event log attributes that identify the error condition. In this case the message portion of the event log will contain “IVR mutex assertion” for the IVR application and “Gatekeeper mutex assertion” for the Gatekeeper application. The registrations will also include the name of the desired callback function within the fault manager (like: handle_ivr and handle_gatekeeper). After registrations are complete, the fault manager looks for the non-event-logging crash condition by polling the IVR and Gatekeeper processes every few seconds. If a process is not detected, the fault manager restarts the missing process. If an event matching one of the registered queries occurs while the fault manager is polling for the other error condition, the event log daemon will send a POSIX signal to the registered process/function with the event log record ID. If the event was an “IVR mutex assertion”, then the function handle_ivr() would be called. Logic within handle_ivr will attempt to resolve the condition or kill the process (in which case, a return to the normal fault manager processing with poll and restart the process.)

Sample API Code

The following API functions would be used to perform what is described in the example above:

Sample HA Application Code

In the mutex assertion function of the H323 library:

```c
Posix_log_printf(LOG_USER, H323_MUTEX_ASSERTION, LOG_CRIT,
    "%s mutex assertion", program_name);
```

/* Note: Since the event-type attribute (H323_MUTEX_ASSERTION) is very specific to this problem, the query could have looked at just this but that would not have identified the application name and the target fault manager must manage more than one application. A regular syslog() call could have also done the same job here. The real advantage of posix_log_printf() is when you are interested in attributes normally not contained in the message portion of a
log entry (things like severity, uid, gid, pid, event_type, and facility). */

Sample Fault Manager Code
/* Setup for the notification calls */
Once=0;
mynotification.sigev_notify=SIGRTMIN+1;
mynotification.sigev_notify_function=handle_ivr;
posix_log_notify_add("data contains IVR mutex assertion\", mynotification,
once|PXLOG_SEND_RECID, &myhandle);
mynotification.sigev_notify=SIGRTMIN+2;
mynotification.sigev_notify_function=handle_gatekeeper;
posix_log_notify_add("data contains Gatekeeper mutex assertion\",

In the main() portion of the fault manager:

/* Handle the case where no error is sent and the process dies */
while(1) {
  if(poll_ivr() == DEAD)
    restart_ivr();
  if(poll_gatekeeper() == DEAD)
    restart_gatekeeper();
}

In the functions section of the fault manager:

int handle_ivr(int signo, siginfo_t *info, void *ignored) {

/* The call to posix_log_notify_add registered a query string and this function as the callback. If the query evaluates to true for any incoming event, then this function will be called. Normal application execution continues after this function returns. */

/* Get the recid of the event that caused this function call. */
posix_log_siginfo_recid(info, ignored, &recid);
printf(" signal=%d from event: Record ID=%d\n",
    info->si_signo, recid);
    try_to_recover_ivr();
}

int handle_gatekeeper(int signo, siginfo_t *info, void *ignored) {

/* The call to posix_log_notify_add registered a query string and this function as the callback. If the query evaluates to true for any incoming event, then this function will be called. Normal application execution continues after this function returns. */
/* Get the recid of the event that caused this function call. */
posix_log_siginfo_recid(info, ignored, &recid);
printf(" signal=%d from event: Record ID=%d\n", 
    info->si_signo, recid);
/* We know what kind of problem led us here but, just for fun, 
let's get the full event record and also print the event type.
if(posix_log_open(&logdes, NULL)==0) {
    /* Go to end of log file since tis is where all new entries go */
    if(posix_log_seek(logdes,NULL,PXLOG_SEEK_END)==0) {
        /* Read backwards looking for the query that was originally used 
to register this function */
        qslen=PXLOG_ENTRY_MAXLEN;
        if(posix_log_notify_get(myhandle2, myoldnotification, &flag,
            qsbuf, qslen, &reqlen) == 0) {
            printf(" Query string: %s \n", qsbuf);
        } else {
            printf("nERROR: posix_log_notify_get failed. errno=%d\n", 
                errno);
        }

        query2=(posix_log_query_t *)&qsbuf;
        if(posix_log_seek(logdes, query2, PXLOG_SEEK_BACKWARD)
            == 0) {
            if((ret=posix_log_read(logdes, entry,
                log_buf,15))==0) {
                printf("The Event type is: %d\n", 
                    entry->log_event_type);
            }
        }
    }
}
try_to_recover_gatekeeper();
}

POSIX 1003.25 Implementations

IBM and Intel have developed separate POSIX 1003.25 Event log implementations. The Intel solution 
meets the basic POSIX requirements without requiring changes to the Operating System (no patches to 
kernel, glibc, linux syslog, or system utilities). The IBM implementation provides the POSIX 
requirements plus many additional features. The Intel solution was the first available and designed to 
meet the needs of early application development where a POSIX event log mechanism was needed. Intel 
and IBM are now working together to provide the richer IBM solution ASAP and make it the standard 
solution (push all kernel and glibc changes into Redhat).

The POSIX Standard does not require the new event logging mechanism to tie in with existing event log 
solutions. IBM and Intel, however, view this as a basic requirement even though existing mechanisms
provide only a subset of the event attributes called out by the POSIX standard. We can’t change all kernel drivers and user applications to use the new POSIX API. IBM and Intel will include the POSIX API in new applications and in HA hardening projects and include methods for intercepting existing syslog and printk log entries to the new event log mechanism. This ensures that all event logging will go to the new POSIX 1003.25 log event. IBM has, initially, provided kernel and GLIBC changes that allow existing syslog() messages to be passed to the POSIX log with most key POSIX attributes included. Printk messages are also captured with a couple of extra attributes. IBM has added new kernel log functions that provide the full POSIX attributes. The Intel implementation makes no attempt to include additional POSIX attributes to the syslog() or printk() messages nor does the Intel version provide new kernel functions that handle the new attributes.

**IBM Implementation**

IBM has been involved with the POSIX Event Logging Working Group for some time now and has been the main driving force behind the POSIX 1003.25 standard. Their implementation covers the POSIX 1003.25 and was made available to the POSIX Working Group participants in August, 2001. Additionally, IBM has added a rich set of features that go beyond the POSIX standard. The official public release of their POSIX features and a subset of the proposed additional features will be available to the open source community on September 20, 2001. The details behind the IBM design are located at: [http://evlog.sourceforge.net/linuxEvlog.html](http://evlog.sourceforge.net/linuxEvlog.html). A diagram of the basic IBM Event log architecture is shown below:
IBM Features above and beyond POSIX 1003.25

Many of the following features were seen by the POSIX working group as being “implementation details” which should not be included in the POSIX standard. Intel and IBM may continue to push some of these into the standard where it makes sense (as in the case of kernel messaging where standardization in Linux would be very valuable).

Extensions to BINARY message format

The POSIX standard provides for a log entry of binary type in the variable message section of the log entry but the POSIX API has no way to do special encoding or decoding of this binary message. IBM’s design goes to great lengths to include data format templates and special API functions to allow user and kernel code to write and read/decode binary log entries. As well as data format specification, a print format and operation can also be included in the messages template.

In user space, the following examples demonstrate the API that is available for binary messages:

```c
    evl_log_write( facility, event_type, severity,
                      "ushort", 0x1111,    /* type = unsigned short, value = 0x1111 */
                      "4*uchar", 5, 10, 15, 20,  /* 4 consecutive unsigned chars */
                      "int[]", 10, int_array,   /* array of 10 ints */
                      "string", "This is an example",
                      "endofdata");

    evl_format_evrec_fixed(const struct posix_log_entry *entry,
```
In kernel space, the following examples demonstrate the API that is available for binary messages:
evl_writek(facility, event_type, severity, "ushort", 0x1111, "4*uchar", 5, 10, 15, 20, "int[]", 10, int_array, "string", "This is an example", "endofdata");

Kernel Messaging
IBM has included changes to printk for writing to the new event log kernel buffer. This buffer is larger than the standard Linux log buffer (128KBs compared to 16KB). To take advantage of the full POSIX event log entry attributes, the following three kernel functions have been added.

```
evl_printk(posix_log_facility_t facility, int event_type, posix_log_severity_t severity, const char *format, ...);
evl_writek(posix_log_facility_t facility, int event_type, posix_log_severity_t severity, ...);
posix_log_write(posix_log_facility_t facility, int event_type, posix_log_severity_t severity, const void *buf, size_t len, int format, int flags);
```

Facility Register
The POSIX Event log specification defines a fixed set of facility codes that match the basic set defined in sys/syslog.h. The IBM event logging allows new facility types to be added/registered through a special command line utility (evlfac). Filters can also be applied to a registered facility to accept only certain types of log entries from this new facility. For example, the filter: (uid = ‘root’ & & severity != DEBUG) will filter-out all log entries that were not created by root or that were of severity DEBUG. Filters cannot be used to filter-out kernel messages or messages generated by root.

Private Log
The IBM event logging supports a private log that is similar to the syslog private log where messages of facility = AUTHPRIV can be directed to separate log file which may have limited read privileges. IBM’s Facility Registry also allows users/applications to register their own facility as a private log.

Handling of Duplicate Log Entries
The IBM event logging supports the handling/elimination of duplicate log entries. A command line utility (evlconfig) can be used to configure the duplicate count to be discarded and the interval of time between duplicates.

Log Management
IBM provides an API and command line utility for configuring how the log file is managed. A daemon (evlogmgrd) is provided to carry out periodic management functions. The API includes functions for
returning the size of the log file (evl_log_getsize), a function to delete log entries (evl_delete_events), and
a function to compact the log file (evl_compact_log). The command line utility (evlogmgr) can be used to
list or modify the current log management configuration. Modification parameters include: frequency of
management actions, a delete query filter, and a compact operation flag. Operations can also be set for the
private log file.

**Special Event-Types**
Event type codes are, automatically, set for to identify messages that come from printk() and syslog().
These are EVL_PRINTK_MESSAGE and EVL_SYSLOG_MESSAGE. Since printk and syslog are not
capable of providing the full POSIX event log attributes, applications may want to check for these event
type settings when building event log query operations.
IBM also sets the event type of a message if the event log buffer is overrun or if duplicate entries are
discarded. These event types are EVL_BUFFER_OVERRUN, EVL_DUPS_DISCARDED.

Command line utilities to generate, read Event Log entries and Notification registration.
The POSIX specification only defines a C based API. IBM also provides command line utilities for these
important functions.

**Event Generation Command**
The evlsend utility can be used to log a POSIX type event with facility, event-type, severity, and event
message. Support is also available for a binary message by specifying a binary option with attribute-types
and attribute-values.

**Event Log Reading/Query Command**
The evlview utility allows you to view events in real time or query events already in the log file.
Options allow you to specify the output format and a destination file. The log file can be scanned
in a forward or backward direction.

**Event Log Notification Registration**
The evlnotify utility can be used to register a command execution when a specified query match occurs
on new events. The execution can be registered as a “once” only execution or a persistent registration.
Registered events can also be listed and removed with this utility.

**Intel Implementation**
The base design for the Intel Implementation is from the March 20, 2001 POSIX 1003.25 specification

**API**
Intel’s implementation covers all the POSIX 1003.25 API functions (identical arguments, argument types,
return codes, errno codes, header definitions, data structures, event attributes, and defined attribute
values.) By closely following the defined API, you should be able to easily switch (compile and link) to
the IBM Event Log solution.
Daemon – API communication

The callback functionality is based on the proposed POSIX real-time signals (sigqueue). Registration and other communications to the daemon (pxlogd) are by sockets. The POSIX specification had no specific recommendation for this. The socket method was chosen to leverage/use the existing open source code of the Linux syslogd and its remote logging capabilities. The POSIX design does not support remote logging but this is easily implemented by adding a nodeid member to the posix_log_entry and query language. Remote logging can then be activated by starting syslogd with the –r option and controlled by adding entries to /etc/syslog.conf for each remote machine.

Interaction with existing system logger

This implementation writes (posix_log_write and posix_log_printf) to the existing system log file (/var/log/messages) through syslog(). At pxlogd startup, the entire system log is scanned and translated to a new POSIX log file (/var/log/pxlog). Pxlogd will then look for new messages (read non-blocking) in the system log file and process all registered queries as each new message comes in. In addition to processing the log entries, pxlogd also handles all registration requests through a socket.

The separate log file is maintained for two reasons. First, the log entries must be numbered in the order they are posted. The current system log does not maintain log entry numbers as required by POSIX. Also, system log rotations could occur at any time and would cause expected entries to disappear. The POSIX log file entries are maintained over system log file rotations and only replaced at startup when there are no registered clients. Log rotations of syslogd will also trigger log file limit checks on the POSIX log file and configuration settings will determine how long and how many events will be maintained in the POSIX log file. The POSIX specification does not require the POSIX implementation to work with the existing system logger but it does state that there should be a single system logging mechanism. We have chosen to work through the current Linux syslog so that other basic OS and kernel modules would not have to be modified to include POSIX log writes. We don’t really think of this as an extension or enhancement to the POSIX specification. It is more thought of as a basic requirement.

The current system log format remains the same with additional semicolon separated fields added to the variable length message. The POSIX log entry length and current syslog max log entry size are both 1024 chars. Since all logs are done through syslogd, the POSIX log mechanism should never see an entry that exceeds the syslog maximum except in the POSIX write functions (posix_log_write and posix_log_printf). These functions should return an error if POSIX PXLOG_TRUNCATE is not set or allow syslog to do the truncation if POSIX PXLOG_TRUNCATE is set. This implementation assumes no truncation so entries longer than the maximum will cause posix_log_write and posix_log_printf to return an error.

Syslogd should be modified to include facility code, and severity in a POSIX semicolon separated section. The current syslogd already receives/handles these values but does not pass them on to the system log file. This change would allow clients to query by severity and facility instead of being limited to just pattern matches in the variable length portion of the log entry. Regular POSIX entries have these basic fields and others that can be used in more complex queries. Since the syslog facility uses a string that includes the name of the logging program, the POSIX logging mechanism would need to have these defined in a configuration file. Note: This implementation, currently, does not provide these syslogd changes so all queries for non-POSIX log entries must focus on just the message (data contains).
Since current Linux event logging does not provide the level of information called out by the POSIX 1003.25, query returns on these entries will have a special value (UNKNOWN) for many parameters. The posix_log_entry member log_format will be set to -1 to identify non-POSIX entries.

**POSIX log writes and Query processing**

Applications wishing to take full advantage of the POSIX event logging would register with the daemon through the API. This registration will include a query string that tells the POSIX daemon what kind of events should trigger a notification to this application. Applications may register more than once to specify different queries and different signal handling functions. The applications may also do direct event logging using the API and may do direct queries at any time.

Initial query string handling (expression handling and interpretation) will be handled by system calls to the Perl interpreter. This implementation may be replaced at the backend of project development if performance is an issue. Initial prototyping indicates that performance will not be an issue here (3 to 4 entries/query-checks can be performed per second for several registered clients).

The message argument to syslog (from the POSIX write and printf functions) will include a semicolon-separated list of all POSIX log parameters. These will be log_recid, log_size, log_format, log_event_type, log_facility, log_severity, log_uid, log_gid, log_pid, log_pgrp, log_time, log_flags, log_thread, log_cpu, log_nodeid.

**Packaging and Install/Configuration of POSIX Event Logger**

This Event Logging Implementation will be provided in an RPM. Special logic will add an entry to the logrotate configuration of syslogd so that rotations of the system log will be reported to pxlogd by a USR2 signal. This is necessary since pxlogd is doing a continuous non-blocking read of the system log and it must be informed when the active file is changed. The RPM install must also add a pxlog entry to the /etc/systems file and send a SIGHUP to inetd (xinetd in Redhat 7.1). The package will also have to allow upgrades by restarting the event log daemon at install completion. Startup scripts (rc.d scripts) will be similar to the syslog startup scripts and will be executed just after syslog startup and just before syslog shutdown.

**Needed Improvements to the IBM Implementation**

The Intel Implementation provides two features that will not be available in the initial IBM Event Logging code release. IBM, currently, is not able to consider these for the initial release so Intel will need to handle possible additions with a re-packaged version of the IBM software. A re-packaging is already necessary to support the initial Redhat 7.1 2.4.2 kernel (IBM’s initial release is built for a 2.4.4 kernel).

**IBM GLIBC changes not possible.**

Changes to the basic Linux libraries (GLIBC) have been rejected as being too risky by the Linux Telco Build Team (TSP). This means that log entries through the current Linux user-mode syslog() will not go to the new IBM event log buffer or log file. To work around this, Intel is adding a daemon (evlforward)
which will monitor all new syslog() entries to the standard Linux log file and forward them to the IBM log through posix_log_printf(). Intel will add this daemon and automatic daemon startup logic to the system through the addition of the re-packaged IBM event log. Intel will not support the IBM GLIBC changes until they are available in a standard Redhat Release.

**Remote Event Logging**

The POSIX working group rejected this as an addition to the standard because POSIX standards only apply to a single system image. If/When the Intel Resource Monitoring product adds remote monitoring support, it will be necessary to enable POSIX Event logging over a network/cluster of servers. This is a future project as Resource Monitoring is limited to a single system for the Telco Release 1.0.