Fault Management Daemon Programmer's Reference Manual

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: FMDPRM 1.2
August 2007

070830@18135

Contents

Preface

The *Fault Management Daemon Programmer's Reference Manual* (FMD PRM) is a description of the internal architecture of the Sun Fault Management Daemon, `fmd`(1M), as well as the programming interfaces exported by the daemon. A fault manager is a software component deployed in a product that acts as a multiplexor between error reports produced by system components, an affiliated set of components called *diagnosis engines* that can automatically diagnose problems using these error reports, and another set of components called *agents* that automatically respond to problem diagnoses. The fault manager also provides interfaces for system administrators and service personnel to observe the activity of the fault management system. The `fmd` daemon is the canonical Sun implementation of a fault manager as described in the Sun Fault Management Architecture (FMA). For more information, see "Sun Fault Management Architecture I/O Fault Services" in Chapter 13, "Hardening Solaris Drivers" in *Writing Device Drivers*.

Who Should Use This Book

This document is intended for systems software engineers who are developing and debugging `fmd`, who are writing new clients for `fmd`, or who are developing FMA error event producers and want to understand more about the subsequent processing of these events. This document also serves as an in-depth reference for service and support personnel who must interact with Sun's fault management components. This document is designed for readers with a high level of familiarity with C programming and with POSIX multi-threaded programming interfaces.

Before You Read This Book

Before you read this document, read "Sun Fault Management Architecture I/O Fault Services" in Chapter 13, "Hardening Solaris Drivers" in "Designing Device Drivers for the Solaris Platform" in *Writing Device Drivers*. For more information, see also the Fault Management community site (`http://www.opensolaris.org/os/community/fm/`) on the OpenSolaris web site (`http://www.opensolaris.org/os/`). To ask questions or make comments, use the `fm:discuss` forum on the OpenSolaris web site (`http://www.opensolaris.org/jive/forum.jspa?forumID=49`).

How This Book Is Organized

This document is organized into the following chapters:

Chapter 1, "Introduction" provides an overview of the fault manager and its services. Read this chapter first.

Chapter 2, "Module API" discusses all the services provided by the fault manager to its client modules, including module entry points, configuration files, and module programming interfaces. This chapter is intended for module developers.

Chapter 3, "Events" describes how events are managed inside of the fault manager and how the fault manager associates time with events. This chapter is intended as advanced background material for module developers, fault manager implementors, and other users who want to understand more about the low-level design issues.

Chapter 4, "Event Transports" describes the implementation of event transports and how events are provided to the fault manager. This chapter also provides a description of advanced programming APIs that can be used by developers to implement event transport modules.

Chapter 5, "Log Files" describes the implementation details of the fault manager log files. This chapter is intended for fault manager implementors.

Chapter 6, "Resource Cache" describes the implementation details of the resource cache. This chapter is intended for fault manager implementors.

Chapter 7, "Checkpoints" describes the implementation details of the module checkpoint mechanism. This chapter is intended for fault manager implementors.

Chapter 8, "Daemon Configuration" describes the configuration options for the fault manager daemon for use by module developers and deployers.

Chapter 9, "Topology" describes a topology snapshot, enumerator modules, and topology map files and how to use them.

Chapter 10, "`fminject` Utility" describes the `fminject` error event injector. The injector can be used for testing, simulation, and debugging the fault manager and its clients.

Chapter 11, "`fmsim` Utility" describes the `fmsim` utility. The fault manager simulation tool can be used for running automated test scenarios and replaying actual fault scenarios.

Chapter 12, "`fmtopo` Utility" describes the `fmtopo` utility, which displays the contents of a topology and enables developers to debug new functionality without interrupting the default fault manager that is active on the system.

Chapter 13, "Debugging" discusses tips and techniques for debugging fault manager modules. This chapter is intended for module developers, fault manager implementors, and service personnel.

Chapter 14, "`syslog-msgs` Agent" describes the interfaces presented by the `syslog-msgs` module, which implements the Sun messaging standard. This chapter is intended for module developers, service personnel, and fault manager implementors.

Chapter 15, "`snmp-trapgen` Agent" describes the Fault Management SNMP MIB and the `snmp-trapgen` module, which can convert fault diagnosis events into SNMP traps and send them through a standard NetSNMP stack.

Appendix A, "ARC Interface Tables" provides tables of interfaces exported and imported by the fault manager, references to ARC materials, and a brief description of how each interface is versioned. This chapter is intended for ARC reviewers and Sun developers who wish to know the stability levels associated with various pieces of the fault manager interfaces and components.

Glossary is a list of words and phrases found in this book and their definitions.

About This Book

The Fault Manager was written and designed by Mike Shapiro with Tim Haley, Cindi McGuire, Andy Rudoff, and Matt Simmons. The *Fault Manager Programmer's Reference Manual* was written by Mike Shapiro. Keith Wesolowski designed the SNMP support and contributed its documentation. To submit review comments on this book, use the `fm:discuss` forum on the OpenSolaris web site (`http://www.opensolaris.org/jive/forum.jspa?forumID=49`).

Accessing Sun Documentation Online

The docs.sun.com℠ Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is http://docs.sun.com. Additional documentation for the Fault Management Architecture is on the fault management community site at `http://www.opensolaris.org/os/community/fm/`.

Typographic Conventions

The following table describes the typographic changes that are used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new terms, or terms to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, Korn shell, and debugger.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`
mdb(1) debugger prompt	`>`

◆ ◆ ◆ CHAPTER 1

1 Introduction

This chapter provides a brief overview of all the major components of the fault manager and a block diagram view of its internal architecture. After reading this chapter, read chapters covering the module programming interfaces or chapters discussing low-level details of the internal daemon architecture.

1.1 What is a Fault Manager?

A *fault manager* is a software component deployed in a product that acts as a multiplexor between error telemetry produced by system components and companion software that is designed to diagnose and respond to that telemetry to facilitate self-healing and improve availability. The fault manager's clients are a set of components called *diagnosis engines* that can automatically diagnose problems using the error telemetry, and another set of components called *agents* that automatically respond to the problem diagnoses by performing actions such as disabling faulty components, issuing messages to alert human administrators, and providing information to higher-level management software and remote services. The fault manager also provides interfaces for system administrators and service personnel to observe the activity of the fault management system.

The `fmd` daemon is the canonical Sun implementation of a fault manager as described in the Sun Fault Management Architecture (FMA). The *Fault Management Daemon Programmer's Reference Manual* (FMD PRM) discusses both the internal architecture of the fault manager and the programming interfaces exported by the daemon. This PRM can be used by developers of diagnosis engines and agents, developers or maintainers of the fault manager, or anyone else who is interested in learning about FMA.

Fault managers are intended to be deployed at a variety of locations throughout Sun's product line inside of appropriate *fault regions* where enough information is available to make an informed, automated diagnosis and effect useful responses. The first fault managers are deployed in the Solaris OS, on service processors, and in network drivers. Fault managers are useful because complex diagnosis engines and agents can be implemented once and shared by a

large collection of components. In addition, providing a common programming model and deployment environment that abstracts away many of the details allows rapid, parallel development of large numbers of automated fault management components. Finally, the fault manager is part of the last line of defense against failure that the system provides. Therefore, the fault manager must be written very carefully and make strong guarantees about its own reliability. The reference implementation is a fairly generic POSIX C application to facilitate the portability and reuse of the design and implementation.

1.2 Events and Modules

Figure 1–1 shows a block diagram of the fault manager's event dispatch mechanism. The central part of the fault manager is the connection of an inbound telemetry event transport, shown at the top of the diagram, to a dispatcher that routes the events to one or more *modules* that have subscribed to the events according to their FMA Event Protocol *class*.

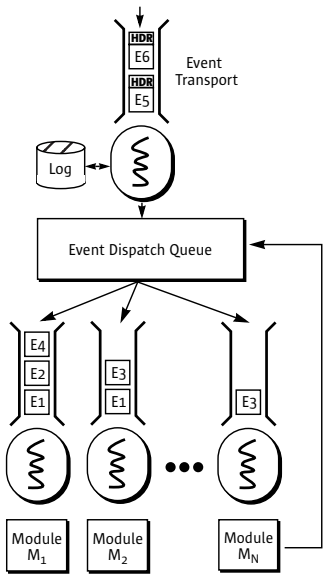


FIGURE 1–1 Event Dispatch Architecture

The bottom of Figure 1–1 illustrates that each module has its own queue onto which incoming events can be placed. Each module also has a processing thread that awakens when events are delivered and runs appropriate callbacks provided by the module. Modules that act as diagnosis engines might produce other kinds of events, such as `list.suspect` diagnosis results. These events then might be routed to other modules that act as response agents. Inbound events are also logged persistently, in case the fault manager or the fault region containing it crashes before the event is fully processed by its subscribers. Event delivery and event management is discussed in detail in Chapter 3, "Events," and log files are discussed in Chapter 5, "Log Files."

1.3 Programming Model

In addition to handling event delivery, the fault manager provides a programming model for implementing various kinds of modules, including diagnosis engines. Diagnosis algorithms can be implemented in higher-level languages such as Eversholt, so in some sense the fault manager's programming interfaces are intended more for the developer of "diagnosis middleware" than for a person who is writing diagnosis rules for a particular new product. The fault manager defines some common abstractions that we expect all diagnosis engines to require. The set of abstractions also facilitates a common checkpointing and observability model for these modules.

The programming model includes support for:

Cases — Diagnosis engines are expected to group telemetry events and other information related to the diagnosis of a particular problem on the system into a set of *cases*. The diagnosis engine may have any number of cases open at a given point in time and may create or destroy them as needed. Some cases may eventually be *solved* by associating a suspect list of one or more problems with the case, at which point the fault manager will publish a corresponding `list.suspect` event for the case. Other fault management agents may then act on the case by *convicting* one or more of the suspects and then attempting to act by disabling the corresponding component, lighting an LED to guide a repair, or notifying higher-level management software. Every case is named using a Universal Unique Identifier (UUID) that is associated with the resulting `list.suspect` event and corresponding message, as we'll see later.

Buffers — The fault manager ensures that its clients are highly available by checkpointing their state on a regular basis in case the module, the fault manager, or the system running the fault manager crash or are restarted. The fault manager checkpoints the state of most resources automatically, but also offers its clients the ability to create a set of named *buffers*, that can be optionally associated with cases and used to serialize and checkpoint module-specific data.

SERD Engines — Sun recommends the use of the Soft Error Rate Discrimination (SERD) algorithm for diagnosis of certain phenomena such as soft upsets in computer memories. The fault manager provides a set of simple interfaces for creating and managing SERD engines and inserting events into them.

Timers — Some diagnosis engines may wish to create timers to wait for the receipt of pairs of related events or other purposes. The fault manager provides interfaces for creating one-shot timers based on the current time or the time associated with a particular event.

Properties	Modules may wish to provide tunable behavior for use by developers, Sun Service, or by Operations and Manufacturing. The fault manager provides a facility for declaring typed properties and parsing a configuration file associated with each module. Configuration parameters can be used for tuning behavior such as SERD engine thresholds.
Statistics	The fault manager permits all modules to publish Private named statistics that can be used by developers and Sun Service to understand module-specific behavior, tune performance, and track down problems. The `fmstat`(1M) utility can be used to view an on-line report of each module's statistics, as well as those kept by `fmd`.
Transports	The fault manager allows modules themselves to implement event transports that can send and receive FMA Protocol events from the operating system or from another fault manager. Event transports and their associated programming APIs are discussed further in Chapter 4, "Event Transports."

In addition to these major abstractions, utility routines for allocating memory and event subscription and other miscellany are provided; the complete programming API for modules is discussed in Chapter 2, "Module API."

1.4 Fault Messaging

Sun has defined a messaging standard for fault messages associated with `list.suspect` events. Fault messages are produced when the diagnosis of a problem requires a human administrator to do something or to be aware of a problem that may impact system availability or service levels. The fault manager expects that at least one of its modules implements a messaging service for `list.suspect` events. In the Solaris reference implementation this module is the `syslog-msgs` agent. The `syslog-msgs` agent produces an FMA standard message to the system console and system log file using the `syslogd`(1M) service. The message also appears in the output of the `fmadm`(1M) `faulty` command. Following is an example message associated with `fmd`.

```
SUNW-MSG-ID: FMD-8000-0W, TYPE: Defect, VER: 1, SEVERITY: Minor
EVENT-TIME: Fri Jan 23 18:33:31 PST 2004
PLATFORM: SUNW,Sun-Fire-V440, CSN: -, HOSTNAME: mix
SOURCE: fmd-self-diagnosis, REV: 1.0
EVENT-ID: e9390b15-bcb8-4a3d-c10c-fe1cb4a67998
DESC: The Solaris Fault Manager received an event from a component to which no
automated diagnosis software is currently subscribed. Refer to
http://sun.com/msg/FMD-8000-0W for more information.
AUTO-RESPONSE: Error reports from the component will be logged for examination by Sun.
IMPACT: Automated diagnosis and response for these events will not occur.
```

```
REC-ACTION: Run pkgchk -n SUNWfmd to ensure that fault management software is
installed properly.  Contact Sun for support.
```

Notice that the message includes a UUID for the diagnosis (the field labeled "EVENT-ID") and a static message identifier (the field labeled "SUNW-MSG-ID"). The UUID is the UUID of the case that the diagnosis engine used to gather information for this diagnosis. The message identifier is a code computed from the class strings of the individual fault events that are part of the suspect list; it is generated by `libdiagcode.so.1` (see PSARC 2003/323. The UUID can be used as an argument to the `fmadm faulty` command or to `fmdump`(1M) to retrieve all of the telemetry information associated with the diagnosis or the details of the individual suspected faults. The message identifier can be used as an argument to a CGI script on Sun's web site to retrieve a knowledge article explaining more about the problem and appropriate responses. The FMA team is working with Enterprise Services to build this web site and populate it automatically from a company-wide registry of all FMA events.

1.5 Resource Cache

In addition to managing events and modules, the fault manager also maintains a cache of resources that have been referenced by a diagnosis. The resource cache is lazily populated and *only* stores the state of the resource with respect to fault management activities. The resource cache is composed of a collection of log files, each associated with a particular Automated System Recovery Unit (ASRU) that has been diagnosed as faulty. The diagnosis results for problems in the resource are stored in this log, allowing this information to persist across system reboots and fault manager restarts. The details of the resource cache are discussed further in Chapter 6, "Resource Cache."

1.6 Service Tools

The Solaris reference implementation of the fault manager includes several bundled observability tools for use by administrators and service personnel. These tools are:

`fmadm`(1M) — The `fmadm` utility can be used to view the modules registered with the fault manager, load and unload modules, and view and update the resource cache. The `fmadm` utility provides service personnel and administrators with an easy way to display every resource on the system that the fault manager believes to be faulty. The `fmadm` utility can display a description of fault, the impact on the system, and the recommended action.

`fmdump`(1M) — The `fmdump` utility can be used to view any of the persistent log files associated with `fmd`, including the error log, fault log, and resource cache log files. `fmdump` can be used by administrators to retrieve the specific details of a particular

diagnosis, and it can be used by service personnel and developers to review all of the telemetry that led to the diagnosis.

`fmstat`(1M) The `fmstat` utility can be used by service personnel and developers to understand the performance characteristics of the fault management system, and to view the private statistics published by any of the fault manager's modules. `fmstat` also provides the ability to view the status of every SERD engine active in a given module.

1.7 Developer Tools

The Solaris reference implementation of the fault manager also includes Private tools for use by developers. These tools include:

fmtopo The `fmtopo` utility is used to view aspects of a FMRI topology including ASRU, FRU, and label properties for all resources in a given scheme-based topology. See Chapter 9, "Topology," for information about topology.

fminject The `fminject` utility can be used to write a high-level description of one or more FMA Protocol events and their payloads, compile these descriptions into a name-value pair list representation, and inject these events into the local SysEvent transport for use in development. `fminject` can also replay events stored in a fault manager log file. See Chapter 10, "`fminject` Utility" for more information.

fmsim The `fmsim` utility can be used to start a fault manager for debugging, testing, or simulation purposes, load a set of customizations, and run one or more injection scenarios. See Chapter 11, "`fmsim` Utility" for more information.

fmd.so The fault manager also provides a `fmd.so` debugging module for the Solaris `mdb`(1) debugger that provides observability for internal `fmd` data structures. These features are described further in Chapter 13, "Debugging."

1.8 Where To Go Next

Readers who wish to learn more about how to develop and deploy a fault manager client module should proceed to Chapter 2, "Module API." All of the material after Chapter 2, "Module API" is intended for readers who wish to learn more about the internals of the fault manager. This material may be helpful for understanding more about the inner workings of some of the programming APIs, but it is not required for writing client modules. Service personnel may also wish to read the man pages for `fmadm`(1M), `fmdump`(1M), and `fmstat`(1M) to learn more about the use of these tools.

◆ ◆ ◆ CHAPTER 2

Module API

A fault manager *module* is a binary object implementing one or more of an optional set of predefined routines called *entry points*, and an optional corresponding configuration file. The module code is written in C and can use any of a collection of fault manager programming interfaces to implement the entry points. This chapter discusses the classes of fault manager modules, methods of installing and deploying modules, and the C programming interfaces for implementing modules. The discussion of the programming APIs also helps to cover nearly all of the services provided by the fault manager, and therefore assists in motivating the material in later chapters. The interfaces described in this chapter are associated with the include file `/usr/include/fm/fmd_api.h`.

2.1 Module Classes

The fault manager distinguishes modules by their *class*, or method of deployment, as opposed to by their purpose (e.g. diagnosis engine, repair agent, messaging agent), of which it has no explicit knowledge. The fault manager assumes that each module is simply a software component that subscribes to one or more event classes and implements an optional set of entry points, described later in this chapter. The fault manager is designed to support three distinct classes of modules, all using the exact same programming model; two are supported in the first version of `fmd`. This design permits module source code to be trivially recompiled into modules of different classes by simply editing a single line in a Makefile, enabling deployers to easily change module classes based on deployment constraints, security considerations, and reliability requirements.

2.1.1 Built-in Modules

The fault manager permits modules to be directly compiled into the `fmd` application binary. In the reference implementation, the built-ins are listed in a table in the source file `fmd/common/fmd_builtin.c`. `fmd` uses a built-in module to perform self-diagnosis of its own error telemetry. Built-in modules are all initialized at `fmd` startup time, and must perform all

initialization activities in their `_fmd_init` routine; unlike other modules, built-ins may not have unbundled configuration files. Built-in modules should be used sparingly, either for modules that are required for the fault manager to operate or in embedded systems contexts where memory constraints or the host environment is insufficient to use one of the other supported module classes.

2.1.2 Plug-in Modules

The fault manager permits modules to be delivered as shared library plug-ins that are installed as separate binary objects that are loaded into the fault manager's address space using `dlopen()`. Plug-in modules are installed in one of the predefined plug-in module directories and may have an optional configuration file. For example, the `syslog-msgs` module consists of the files `syslog-msgs.so` and `syslog-msgs.conf` installed in the directory `/usr/lib/fm/fmd/plugins/`. When `fmd` initializes, it searches for plug-in modules to load in the following directories:

- `/usr/platform/`*platform*`/lib/fm/fmd/plugins` (where *platform* is `uname -i` by default)
- `/usr/platform/`*machine*`/lib/fm/fmd/plugins` (where *machine* is `uname -m` by default)
- `/usr/lib/fm/fmd/plugins`

Plug-in modules have slightly greater memory overhead than built-in modules, but permit parallel development and independent delivery and upgrade of the module. Plug-in modules have essentially the same reliability characteristics as built-in modules in that they execute inside the address space of the fault manager and therefore plug-in module defects can potentially cause failure of the fault manager or other modules.

2.1.3 External Modules

The fault manager permits modules to be delivered as independent applications that are installed as separate binary objects and launched by the fault manager on-demand. External modules are installed in one of the predefined external module directories and must have an associated configuration file. External modules are intended to be installed in the following directories:

- `/usr/platform/`*platform*`/lib/fm/fmd/agents` (where *platform* is `uname -i` by default)
- `/usr/platform/`*machine*`/lib/fm/fmd/agents` (where *machine* is `uname -m` by default)
- `/usr/lib/fm/fmd/agents`

The first version of `fmd` does not support external modules; support for this module class will be added in a future phase of the project. Like built-in modules, external modules permit parallel development, deployment, and upgrade of fault manager components, but with the additional advantage of greater reliability and security. External modules cannot corrupt one another or

the state of the fault manager, and are therefore better suited to future efforts to open the fault manager architecture to developers outside of Sun.

2.2 Threading Model

`fmd` deliberately exports a single-threaded programming model to client modules in order to simplify their design, coding, and testing. The fault manager guarantees that *only one thread* will execute in any of a given module's entry points at any given time, and that *only one module entry point* of a given module will be executing at any time. However, the fault manager is implemented as a multi-threaded daemon, so there are some important considerations for the development and compilation of `fmd` clients, depending on the module class selected for deployment.

First of all, code residing in separate modules can and will be executed simultaneously by `fmd` using different threads; a module can make no assumptions about what other modules are configured or executing at a given time, and no interfaces are providing for explicit discovery of or communication with other modules. Second, a module may not assume that any *particular* thread is associated with its execution. Although the fault manager will not execute multiple module entry points simultaneously, no guarantees are made that the thread which executes one entry point is the same thread that will next execute that entry point. As a result, module writers must not cache thread identifiers such as `pthread_self()` persistently or use thread-specific data as a mechanism for storing module state.

If a module is compiled as a built-in or plug-in module, it must additionally be written using re-entrant interfaces because multiple modules will be executing inside of `fmd` simultaneously. For example, modules must use `strtok_r()` rather than `strtok()` because `strtok()` uses static data inside of `libc.so.1` and therefore multiple modules calling `strtok()` simultaneously could corrupt one another. However, all modules may use static data in their own module source code because within the module `fmd` guarantees that only one thread will be executing in the module code at a time. Similarly, the use of mutexes to protect data within a module is not required; module writers can assume a single-threaded programming model based on the previously described rules.

If a module is deployed as an external module, then the re-entrant constraint is also relaxed because the module code executes outside of the address space of `fmd`, but programmers are encouraged to keep module code portable between the different classes and program to this constraint anyway. The Solaris compilation environment for `fmd` provides a set of common Makefiles that can be included to obtain the proper compilation flags for modules based upon their class, so that portable code can be trivially recompiled for a different class by simply changing a Makefile directive.

The fault manager does permit module developers to create multi-threaded modules for those situations where multi-threading is required. Typically, multi-threading is only required for modules that are implementing one or more event transports, as described in Chapter 4, "Event

Transports." The APIs for creating and manipulating threads within a module are described at the end of this chapter. Threads within a module should use the POSIX threads APIs to perform synchronization when necessary.

2.3 Failure Model

The fault manager provides a very rigid, simple failure handling model for its interfaces. Specifically, the majority of the programming interfaces are designed to detect programming errors internally and to abort execution of the module if an error is detected. As a result, most of the module interfaces return `void`, minimizing the opportunity for module code to inadvertently fail to check the result of an API call or fail to handle or test the handling of obscure error conditions. The precise conditions under which API calls will trigger an error are described below in the documentation for each interface.

If an error is triggered by an interface call, `fmd` will force the thread executing the module code to `longjmp()` (unwind the stack) to the point at which it entered the current module entry point, and then call the module's `_fmd_fini` routine (described below) if appropriate. If the `_fmd_init` or `_fmd_fini` routines trigger an error, no further module code is executed. After this error handling occurs, the module will be forcibly disabled by `fmd`. A failed module can later be reset using the `fmadm` command.

If an auxiliary module thread triggers a programming error, the auxiliary thread will be forced to exit as if by `pthread_exit()` and the fault manager will arrange for the main module thread to terminate the module at the earliest possible opportunity. The fault manager uses POSIX thread cancellations to forcibly terminate any remaining module threads following the termination of the main module thread.

The fault manager generates error reports for all of its errors and those generated by its client modules using the class family `ereport.fm.fmd.*`; a complete list of the errors is contained in the source file `fmd/common/fmd_error.h`. Any error reports generated by `fmd` are also placed in the system error log for subsequent analysis by Sun. The fault manager also provides additional capabilities for developers to aid in debugging modules, such as stopping `fmd` for examination by a debugger, forcing a core dump, or reporting errors to `stderr` or `syslogd`(1M); these behaviors can be configured by a developer and are described further in Chapter 13, "Debugging."

In the descriptions of programming interfaces below, the phrase "module abort" is used to describe situations under which module execution will be aborted due to a (likely) programming error or corruption of `fmd` internal state. Module writers may also implement assertions by explicitly calling `fmd_abort()` or `fmd_vabort()`, described below, which use the same mechanism. When a module aborts, any pending events on its queue are silently discarded. Similarly, any subsequent events received by the fault manager that match the failed module subscriptions will be silently discarded. After a module abort occurs, the `fmadm config` command will report the module as `failed`, as shown in the following example:

```
# fmadm config
MODULE                   VERSION STATUS  DESCRIPTION
buggy                    1.0     failed  buggy client module
cpumem-retire            1.0     active  CPU/Memory Retire Agent
eft                      1.12    active  eft diagnosis engine
fmd-self-diagnosis       1.0     active  Fault Manager Self-Diagnosis
io-retire                1.0     active  I/O Retire Agent
syslog-msgs              1.0     active  Syslog Messaging Agent
```

Similarly, the `fmadm faulty` command reports a faulty resource using the "`fmd`" scheme and indicating the name of the module:

```
# fmadm faulty
--------------- ------------------------------------  -------------- ---------
TIME            EVENT-ID                              MSG-ID         SEVERITY
--------------- ------------------------------------  -------------- ---------
Aug 16 04:24:22 cc3a6778-bcc0-4e46-8943-f9cf48b50b4b  FMD-8000-2K    Minor
Fault class : defect.sunos.fmd.module
Problem in  : fmd:///module/buggy

Description : A Solaris Fault Manager component has experienced an error that
              required the module to be disabled.  Refer to
              http://sun.com/msg/FMD-8000-2K for more information.

Response    : The module has been disabled.  Events destined for the module
              will be saved for manual diagnosis.

Impact      : Automated diagnosis and response for subsequent events associated
              with this module will not occur.

Action      : Use fmdump -v -u <EVENT-ID> to locate the module.  Use fmadm
              reset <module> to reset the module.
```

As part of module failure handling, the fault manager's `fmd-self-diagnosis` engine will publish a suspect list containing one or more `defect.fm.fmd.module` events indicating the likely cause of the problem and instructing the system administrator how to reset the module or repair its configuration file. The following example illustrates output from the fault log after a module failure has occurred:

```
# fmdump -v
TIME                 UUID                                 SUNW-MSG-ID
Jan 01 19:36:22.0955 331ebed4-cd86-e25b-b7ea-f606fb9883a8 FMD-8000-2K
  100%  defect.fm.fmd.module
         FRU: -
        rsrc: fmd:///module/buggy
```

The suspect list event will cross-reference events of class `ereport.fm.fmd.module` for each error that was previously detected in the module or its configuration file or reported using the

`fmd_hdl_abort()` or `fmd_hdl_error()` functions. The cross-referenced events can be retrieved by specifying the `-e` and `-u` options to `fmdump`, as shown in the following example:

```
# fmdump -V -e -u 331ebed4-cd86-e25b-b7ea-f606fb9883a8
TIME                           CLASS
Jan 01 2005 19:36:22.095527000 ereport.fm.fmd.module
nvlist version: 0
    version = 0x0
    class = ereport.fm.fmd.module
    detector = (embedded nvlist)
    nvlist version: 0
        version = 0x0
        scheme = fmd
        authority = (embedded nvlist)
        nvlist version: 0
            version = 0x0
            product-id = i86pc
            server-id = poptart
        (end authority)

        mod-name = buggy
        mod-version = 1.0
    (end detector)

    ena = 0x2c01
    msg = someone told me to abort
```

2.4 Persistence Model

The fault manager provides a set of services to ensure that module state is persistent across system reboot and restarts or failures of the fault manager. Module state is persisted by taking *checkpoints* of the module after any call to a module entry point is completed, and then restoring the checkpointed state when the module is subsequently loaded. The checkpoints contain some information about the module's metadata and serialized copies of module state such as SERD engines and buffers. Every module is responsible for restoring data structures corresponding to any persistent state as needed. From the module's perspective, each checkpoint is *atomic*; after an entry point completes, either all of state modified during the entry point is checkpointed or none of it is. The implementation details of checkpointing are discussed later in Chapter 7, “Checkpoints.” Checkpointing semantics associated with particular `fmd` services are discussed in the corresponding sections of this chapter.

2.5 Module Loading

At `fmd` startup time, modules are loaded in the order built-in modules, plug-in modules, external modules, according to the search paths configured for each class shown in "2.1.2 Plug-in Modules" on page 24 and "2.1.3 External Modules" on page 24. Plug-in and external modules are assigned the name corresponding to the basename of the module object with any trailing `.so` suffix removed. Built-in modules have their names declared in a table in the `fmd` source code. Module names are kept in a single, global namespace by `fmd`, and only one module of a given name is permitted at a time. If `fmd` encounters a module in a module search path whose name corresponds to an already loaded module, then the second module is silently ignored and is not loaded. This implies that built-in modules take precedence over plug-in modules, and plug-in modules take precedence over external modules. Furthermore, the search paths for each class are defined so that platform or machine-class-specific modules take precedence over common modules. This architecture permits easy deployment of a generic module and then architecture-specific modules that can overload generic behavior on appropriate platforms.

`fmd` guarantees that all built-in and plug-in modules will be loaded and fully initialized at startup prior to processing any events ready to received from the inbound event transport. `fmd` guarantees that external modules will have their configuration files (described below) processed for any subscriptions prior to processing any events, but reserves the right to delay startup and initialization of modules of this class until an event matching a subscription is received. Once `fmd` is initialized, no further scans of the various module paths are made for the remainder of its lifetime. However, modules may be loaded after `fmd` begins running using the `fmadm`(1M) `load` subcommand; this can be used by administrators or field personnel, or can be used as part of a script to install and load a new version of a fault manager module.

2.5.1 _fmd_init

`void _fmd_init(fmd_hdl_t *`*handle*`)`

The `_fmd_init()` function is called once when the module is initialized, and is required to be implemented by all modules. This function receives as a parameter a pointer to an opaque *handle*, which is associated with this particular instance of the loaded module. The handle initially is *unregistered* in that it has no configuration information associated with it. The `_fmd_init()` routine is responsible for performing any one-time initialization of the module and *registering* the handle with the `fmd` framework using `fmd_hdl_register()`, described below. If at the time the `_fmd_init()` function returns the handle has been registered, then module initialization is considered to be successful. If `_fmd_init()` returns without registering the handle (or after registering the handle and then subsequently unregistering it), the module initialization is considered to be unsuccessful and an appropriate error will be logged and the module will be unloaded as the result of failing to register.

The same handle is passed to all of the other module entry points and is also passed back to `fmd` as the first parameter to most API calls. Handles are associated with particular modules and any attempt to pass a handle which is not owned by the caller or is not valid will cause `fmd` or the calling module to be aborted. The handle values are not persistent across restarts, and modules should make no assumptions about the nature of the handle value nor should they attempt to dereference this value as a pointer.

If the module has any checkpointed state, including cases, SERD engines, and buffers, these will be restored prior to entry to `_fmd_init()`. Therefore, the module may use `_fmd_init()` as a place to reload any data structures that have been serialized in these buffers or re-initialize appropriate case-specific data structures. If any stale or invalid data is discovered in checkpointed state, the module should delete the offending data structures and make a best effort to continue. The `fmd_hdl_error()` function can be used to report any non-fatal errors.

2.5.2 _fmd_fini

`void _fmd_fini(fmd_hdl_t *`*handle*`)`

The `_fmd_fini()` function is an optional entry point that a module can implement in order to provide any one-time cleanup activities prior to unloading. The module will be checkpointed one more time, if necessary, after `_fmd_fini()` returns. It is not necessary to unregister the *handle* associated with the module in `_fmd_fini()`, although the module is free to do so. If the handle is not unregistered here, it will be unregistered automatically by `fmd` once the function returns. If `_fmd_init()` did not register a handle, or if an error occurred during `_fmd_init()`, `_fmd_fini()` will not be called. If an error occurs during the execution of `_fmd_fini()`, module execution will be aborted and no further calls to `_fmd_fini()` will occur.

2.6 Handle Registration

The `_fmd_init()` function receives an opaque pointer known as a *handle* associated with the module instance, and is required to *register* this handle with `fmd` in order to describe the module's metadata to the daemon and trigger processing of the module's configuration file, if one is present. The handle is passed as the first parameter to all module entry points, and is then used as a parameter to most application programming interface calls. The functions in this section can be used by the module to register and unregister handles, store module-specific data in a handle, and subscribe to events.

2.6.1 fmd_hdl_register

`int fmd_hdl_register(fmd_hdl_t *`*handle*`, int` *version*`, const fmd_hdl_info_t *`*info*`)`

Register the specified *handle* with `fmd` and complete module initialization by processing any configuration file that is present for this module. Unlike most other API functions,

`fmd_hdl_register()` returns an integer indicating whether it succeeded (zero) or failed (non-zero), permitting modules to deallocate any memory allocated in `_fmd_init()` prior to calling `fmd_hdl_register()`.

The module is required to specify the *version* of the application programming interface that it compiled against (using the constant `FMD_API_VERSION` provided in the header file) and a pointer to a structure describing the module's entry points and metadata, defined as follows:

```
typedef struct fmd_hdl_info {
    const char *fmdi_desc;          /* client description string */
    const char *fmdi_vers;          /* client version string */
    const fmd_hdl_ops_t *fmdi_ops;  /* ops vector for client */
    const fmd_prop_t *fmdi_props;   /* array of configuration props */
} fmd_hdl_info_t;
```

The `fmdi_desc` member should point to an ASCII string briefly describing the module's purpose (for example, “Syslog Messaging Agent”). The `fmdi_vers` member should point to an ASCII string containing a version string in dotted-decimal format. Modules are expected to maintain their own version strings in accordance with the rules described in `attributes`(5). If the *version* is not supported by `fmd` or if the description or version string for the module are not specified, `fmd_hdl_register()` will return non-zero to indicate failure.

The `fmdi_ops` member must point to a valid `fmd_hdl_ops_t` structure, defined as follows:

```
typedef struct fmd_hdl_ops {
    void (*fmdo_recv)(fmd_hdl_t *, fmd_event_t *, nvlist_t *, const char *);
    void (*fmdo_timeout)(fmd_hdl_t *, void *);
    void (*fmdo_close)(fmd_hdl_t *, fmd_case_t *);
    void (*fmdo_stats)(fmd_hdl_t *);
    void (*fmdo_gc)(fmd_hdl_t *);
    int (*fmdo_send)(fmd_hdl_t *, fmd_xprt_t *, fmd_event_t *, nvlist_t *);
 void (*fmdo_topo)(fmd_hdl_t*, topo_hdl_t*);
} fmd_hdl_ops_t;
```

These members in turn should be initialized to the various functions in the module that implement the corresponding entry points. The semantics of each entry point are discussed in the next section. All of the entry points are optional; entry points which are not implemented may either be defined as an empty routine or a `NULL` pointer may be used for the corresponding member.

The `fmdi_props` member should be set to point to an array of `fmd_prop_t` structures, terminated by an element filled with zeroes, or it may be set to `NULL` indicating that the module has no properties. The properties can be retrieved later using the property interfaces described in this chapter. The `fmd_prop_t` structure is defined as follows:

```
typedef struct fmd_prop {
    const char *fmdp_name;  /* property name */
```

```
    uint_t fmdp_type;       /* property type */
    const char *fmdp_defv;  /* default value */
} fmd_prop_t;
```

The `fmdp_name` member of each property should indicate a unique string name for the property that will be used to retrieve its value in the module and specify its value in the module configuration file. Each module has its own property namespace, and different modules may reuse the same property names. Property names are restricted to be sequences of letters, digits, underscores, hyphens, and periods. `fmd` reserves for its own use all property names beginning with the prefixes "_", ".", "fmd_", "fmd.", "FMD_", and "FMD."; all other valid strings are free to be used by the module. If an invalid property name is specified in the table or if the same property name is specified more than once, an error will occur and `fmd_hdl_register()` will return non-zero.

The `fmdp_type` member of each property should be set to one of the constants listed in the table below, indicating the type of the property. If an invalid type is specified, an error will occur and `fmd_hdl_register()` will return non-zero.

TABLE 2–1 Property Types

fmdp_type Setting	Value Type	Description
`FMD_TYPE_BOOL`	`int`	Boolean value. The value will be set to the constant `FMD_B_FALSE` or `FMD_B_TRUE`.
`FMD_TYPE_INT32`	`int32_t`	Signed 32–bit integer.
`FMD_TYPE_UINT32`	`uint32_t`	Unsigned 32–bit integer.
`FMD_TYPE_INT64`	`int64_t`	Signed 64–bit integer.
`FMD_TYPE_UINT64`	`uint64_t`	Unsigned 64–bit integer.
`FMD_TYPE_STRING`	`const char *`	ASCII string.
`FMD_TYPE_TIME`	`uint64_t`	Time quantum expressed as an unsigned 64–bit number of nanoseconds.
`FMD_TYPE_SIZE`	`uint64_t`	Size expressed as an unsigned 64–bit number of bytes.

The `fmdp_defv` member can be optionally set to a default value for the property; this string should specify the value in the same syntax permitted in the module configuration file (for example, "123" would correspond to the integer value 123, and "7h" would correspond to the time quantity 7 hours expressed in nanoseconds). The syntax for values is described below; see "2.7.3 setprop" on page 35. If the `fmdp_defv` member is set to `NULL`, the property will have zero as its initial value if the property is an integer, or `NULL` as its initial value if the property is a string. If the `fmd_defv` string contains a syntax error or invalid value for the property according to the rules for the property type, an error will occur and `fmd_hdl_register()` will return non-zero.

2.6.2 fmd_hdl_unregister

```
void fmd_hdl_unregister(fmd_hdl_t *handle)
```

The `fmd_hdl_unregister()` function unregisters a *handle* previously registered with `fmd` using `fmd_hdl_register()`. Once a handle is unregistered, all of its subscriptions will be removed, no further entry points will be called, and the module may be unloaded. Modules do not typically need to call `fmd_hdl_unregister()` as it will be called automatically following the completion of `_fmd_fini()`; it is provided in case a module wishes to cause module load failure in `_fmd_init()` after `fmd_hdl_register()` succeeds, and for symmetry. If the specified *handle* is not registered, `fmd_hdl_unregister()` has no effect.

2.6.3 fmd_hdl_setspecific

```
void fmd_hdl_setspecific(fmd_hdl_t *handle, void *data)
```

The `fmd_hdl_setspecific()` function can be used to associate a *data* pointer with the specified *handle* for the duration of the module's lifetime. This pointer can be subsequently retrieved using `fmd_hdl_getspecific()`. If the pointer is used to refer to dynamically allocated memory, the module is responsible for freeing this memory in its `_fmd_fini()` entry point before the module is unloaded. The handle-specific data pointer is *not* saved by the module checkpointing mechanism; modules must restore any relevant value in `_fmd_init()`.

2.6.4 fmd_hdl_getspecific

```
void *fmd_hdl_getspecific(fmd_hdl_t *handle)
```

Return the handle-specific data pointer previously associated with *handle* using `fmd_hdl_setspecific()`. If the module has not ever called `fmd_hdl_setspecific()` on this handle, `fmd_hdl_getspecific()` returns `NULL`.

2.7 Configuration Files

A module configuration file can be optionally delivered for any module. The configuration file must be installed in the same directory as the module and should be named using the module basename and the suffix "`.conf`". For example, the `syslog-msgs` module is installed using the files:

```
/usr/lib/fm/fmd/plugins/syslog-msgs.so
/usr/lib/fm/fmd/plugins/syslog-msgs.conf
```

Module configuration files are intended to simplify the development and tuning of module behavior on the part of module developers, service personnel, and operations and manufacturing. Module configuration files are *not* intended to be stable, publicly documented

interfaces for administrators and users. If a module needs to offer interfaces that are intended to be publicly documented and tuned by customers, these should be provided through some higher-level management mechanism (for example, a graphical user interface, firmware command-line utility, or facility such as the Solaris Service Management Framework). Module configuration files are important, though, because service, operations, and manufacturing personnel all require the ability to tune fault management behaviors on a live system without having to recompile binary modules. For example, manufacturing may wish to lower the threshold on a SERD engine in order to implement stricter standards during test executions.

Module configuration files are processed automatically as part of a module's call to `fmd_hdl_register()`; the set of properties that can be used in the configuration file are defined by the structures passed to this call. An `fmd` configuration file is a text file consisting of one or more lines of directives in which blank lines and lines beginning with # (comments) are ignored. Whitespace is used to separate directives from arguments, and a newline is used to end a directive line. The following subsections describe the syntax for the supported configuration file directives. If a syntax error, invalid directive, or invalid directive argument are detected in a configuration file, an appropriate error is logged and `fmd_hdl_register()` returns non-zero to indicate failure.

2.7.1 subscribe

`subscribe` *class*

Subscribe the module to the specified event *class*. The class string is a sequence of one or more period (`.`) delimited strings which refer to FMA Event Protocol event classes. The class string may contain an asterisk (`*`) at one or more positions to indicate a wildcard at the corresponding location. For example, the configuration file directives:

```
subscribe soup.leek
subscribe soup.potato
subscribe fruit.*
```

would result in subscriptions to the event classes `soup.leek`, `soup.potato`, `fruit.citrus.orange`, `fruit.banana` and so on for any other events whose class string begins with the prefix "`fruit.`". The `subscribe` directive is equivalent to a call to the `fmd_hdl_subscribe()` function for the specified module.

If a module subscribes to a particular event class, it is also implicitly subscribed to any `list.suspect` event that *contains* an event of that particular class. This permits response agent modules to subscribe to a specific list of faults and receive them when a fault is replayed by the resource cache (see Chapter 6, "Resource Cache") or when a new `list.suspect` event is received containing one of the faults as a suspect. The module's `fmdo_recv` entry point must be written to be able to handle this behavior by checking the received event's *class* and unpacking the list as necessary.

2.7.2 dictionary

`dictionary` *name*

Associate suspect lists generated by this module with the specified `libdiagcode.so.1` event dictionary. The event dictionary is used to generate message codes for the suspect lists that can be used to retrieve knowledge articles from Sun's web site. Message codes are also used to look up human-readable localized summary messages corresponding to any `list.suspect` events received by the `syslog-msgs` agent.

The dictionary *name* can refer to an absolute path of a dictionary file, or a dictionary name found in the default system dictionary location (`/usr/lib/fm/dict/`). Dictionary names may include any of the pathname expansion tokens described in Table 8–1. Dictionary directives are only used with modules that are functioning as diagnosis engines (that is, those that call `fmd_case_solve()`) and are not required for other modules. If a dictionary is not named by a module that is a diagnosis engine, any `list.suspect` events produced by this module will use a dictionary code in the `fmd` self-diagnosis dictionary indicating that the module has no dictionary.

2.7.3 setprop

`setprop` *property value*

Set the specified module *property* to the specified *value*. The *property* should correspond to the string name of one of the module properties registered as part of the module's call to `fmd_hdl_register()`. The *value* should be set according to the following rules for the property's type. Integer constants may be represented in decimal, octal (indicated with a leading "`0`"), or hexadecimal (indicated with a leading "`0x`" or "`0X`").

TABLE 2–2 Value Syntax

Property Type	Property Value
`FMD_TYPE_BOOL`	The string `true` or the string `false` (case-insensitive).
`FMD_TYPE_INT32`	Integer constant in the range `[INT32_MIN, INT32_MAX]`.
`FMD_TYPE_UINT32`	Integer constant in the range `[0, UINT32_MAX]`.
`FMD_TYPE_INT64`	Integer constant in the range `[INT64_MIN, INT64_MAX]`.
`FMD_TYPE_UINT64`	Integer constant in the range `[0, UINT64_MAX]`.
`FMD_TYPE_STRING`	ASCII string. If the string is not surrounded by double quotes, it may not contain any whitespace or newlines. If the string is surrounded by double quotes, these will be removed and any ANSI C character escape sequences will be expanded inside of the string (for example, `\t` will become a tab).

TABLE 2–2 Value Syntax *(Continued)*

Property Type	Property Value
FMD_TYPE_TIME	Integer constant in the range [0, UINT64_MAX] representing a number of nanoseconds. The integer constant may have one of the valid time suffixes (described below) appended to it to perform a unit conversion.
FMD_TYPE_SIZE	Integer constant in the range [0, UINT64_MAX] representing a number of bytes. The integer constant may have one of the valid size suffixes (described below) appended to it to perform a unit conversion.

Properties of type FMD_TYPE_TIME may use any of the following suffixes:

ns or nsec	Number of nanoseconds
us or usec	Number of microseconds
ms or msec	Number of milliseconds
s or sec	Number of seconds
m or min	Number of minutes
h or hour	Number of hours
d or day	Number of days
hz	Number of times per second

Properties of type FMD_TYPE_SIZE may use any of the following suffixes:

k or K	Number of kilobytes
m or M	Number of megabytes
g or G	Number of gigabytes
t or T	Number of terabytes

2.8 Entry Points

Modules are expected to implement functions corresponding to one or more of the module entry points described in the fmd_hdl_ops_t structure, described earlier. In this section we describe the syntax and semantics of each entry point.

2.8.1 fmdo_recv

```
void module_recv(fmd_hdl_t *handle, fmd_event_t *event, nvlist_t *nvl, const char *class)
```

The `fmdo_recv` entry point is called for each event that is received by the fault manager that has a *class* that matches one of the module's subscriptions. An opaque handle for the *event* that can be used to associate the event with a case or SERD engine is included as a parameter, along with a name-value pair list representation of the actual event protocol data, and a pointer to the class string. The *event*, *nvl*, and *class* should not be used after this entry point returns. If the module requires longer-term access to this data, it can make a copy of the name-value pair list or class string.

The fault manager provides *at most once* semantics for `ereport.*` events, in that error events may be lost due to out-of-memory conditions or queue overflows, but a module is guaranteed that if it receives an error event and successfully checkpoints, the same event will not be seen more than once. The fault manager provides *at least once* semantics for `fault.*` and `list.*` events, in that these events may be presented to the module more than once if, in the middle of event processing, the fault manager or the module crashes, or the system containing the fault manager crashes or restarts. Once the `fmdo_recv` entry point returns, if the event has been added to a case or SERD engine, it will be marked in the fault manager log to indicate it should not be replayed. If the event has not been added to any case or SERD engine, it will be discarded and will not be presented to the module again. All of these changes in state are checkpointed as a single unit after the entry point returns; therefore a module is guaranteed that either the case or SERD engine will reference the event and it will not be replayed, or no references to the event will exist and it will be replayed.

2.8.2 fmdo_timeout

```
void module_timeout(fmd_hdl_t *handle, void *data)
```

The `fmdo_timeout` entry point is called once to indicate the expiry of a timer installed using `fmd_timer_install()`. This function receives as a parameter the *data* pointer associated with the timer installation; see "2.19.1 fmd_timer_install" on page 57 for details.

2.8.3 fmdo_close

```
void module_close(fmd_hdl_t *handle, fmd_case_t *case)
```

The `fmdo_close` entry point is called once to indicate that the module itself or another module has closed the specified *case*. Once this entry point returns, all data structures associated with the case will be automatically deallocated. Cases can be closed when the module no longer needs them or when a Solved case has had one or more of its convicted faults resolved by disabling the corresponding ASRUs. The module is also responsible for freeing any case-specific data structures (such as those referred to by the case-specific data pointer) as part of this call.

2.8.4 fmdo_stats

`void` *module*`_stats(fmd_hdl_t *`*handle*`)`

The `fmdo_stats` entry point is called to indicate that a user has requested a snapshot of the module's statistics using the `fmstat`(1M) utility. The entry point provides the module with an opportunity to perform any appropriate updates to the statistics (especially if these are costly and cannot be done continuously). After this entry point returns but before any other entry points are called, the fault manager will take a snapshot of the entire set of module statistics and return this snapshot to the requester. If many requests are received within a short span of time, the fault manager may omit the call to this entry point and reuse the same snapshot in order to limit the overhead of statistics requests on the fault manager or its clients.

2.8.5 fmdo_gc

`void` *module*`_gc(fmd_hdl_t *`*handle*`)`

The `fmdo_gc` entry point is called to indicate that the module should garbage collect any data structures that may be stale. Any SERD engines associated with the module are garbage collected automatically by the fault manager *before* calling this entry point. The garbage collection entry point is called once per day by default, and may also be called if the fault manager is running low on memory. Module programmers should limit activity in this entry point to examining existing data structures and cleaning them up, and not allocating any new ones. If a module requires periodic updates that may include *allocating* memory, these updates should be performed from a timer callback instead.

2.8.6 fmdo_send

`int` *module*`_send(fmd_hdl_t *`*handle*`, fmd_xprt_t *`*xp*`, fmd_event_t *`*event*`, nvlist_t *`*nvl*`)`

The `fmdo_send()` entry point is called for those modules that implement one or more event transports; other modules may omit this entry point and specify NULL instead. The `fmdo_send()` entry point is called when the transport *xp* is ready to transport the specified *event* to the transport peer. Transport module implementation is described in further detail in Chapter 4, "Event Transports."

2.8.7 fmdo_topo

`void` *module*`_topo(fmd_hdl_t *`*handle*`, topo_hdl_t *`*topo*`)`

The `fmdo_topo()` entry point is called whenever the resource topology changes. See "2.11 Resource Topology" on page 40 for more information. This entry point is optional. If you

specify NULL for the `fmdo_topo()` entry point, then resource topology changes are ignored. The topology handle is valid only for the duration of the entry point callback. To retrieve a persistent copy of the topology, call `fmd_hdl_topo_hold()`. See "2.11.1 fmd_hdl_topo_hold" on page 40 for more information.

2.9 Event Subscription

The fault manager provides the interfaces described in this section to modify event subscriptions associated with a module handle. Event subscriptions can also be registered with `fmd` using the `subscribe` directive in a module's configuration file; see "2.7.1 subscribe" on page 34. Event subscriptions are *not* preserved by the module checkpoint mechanism; modules are required to subscribe to events appropriately as part of their initialization or using their configuration file.

2.9.1 fmd_hdl_subscribe

`void fmd_hdl_subscribe(fmd_hdl_t *`*handle*`, const char *`*class*`)`

Add the specified *class* to the module's event subscription list. The class may use the "*" character as a wildcard, as described above for the `subscribe` configuration file directive (see "2.7.1 subscribe" on page 34). The wildcard character may only be used to match an entire period-delimited field or set of fields; it may not be used to match partial string values within a particular period delimited field. No validation of the *class* can be done, as `fmd` is not aware of the event publishers and these may change dynamically during the system lifetime. Module developers are responsible for constructing appropriately matching pairs of telemetry providers and module subscribers for deployment in a given system.

Note – If a module creates more than one identical or overlapping subscription to a given event class, an event that matches multiple subscriptions will be delivered exactly once. This behavior was not implemented properly in FMD 1.0, but has now been corrected for FMD 1.1 and may be relied upon for future versions of the fault manager.

2.9.2 fmd_hdl_unsubscribe

`void fmd_hdl_unsubscribe(fmd_hdl_t *`*handle*`, const char *`*class*`)`

Remove the specified *class* from the module's event subscription list. The class must exactly match an earlier call to `fmd_hdl_subscribe()` or a subscription found in the module's configuration file; if it does not, a module abort will be triggered. Once `fmd_hdl_unsubscribe()` returns, the module is guaranteed that no further events matching the

specified class will be received by the module's `fmdo_recv` entry point. If any matching events are pending on the module's event queue when `fmd_hdl_unsubscribe()` is called, they are deleted as part of this call.

2.10 Event Dictionaries

The fault manager manages a set of `libdiagcode.so.1` event dictionaries on behalf of each module. When a case is solved, the suspect list is automatically associated with an appropriate message identifier by looking up the suspect list keys in the event dictionaries associated with the module.

2.10.1 fmd_hdl_opendict

`void fmd_hdl_opendict(fmd_hdl_t *`*handle*`, const char *`*dictname*`)`

Open the event dictionary specified by *dictname* and add this dictionary to the list of dictionaries that are used to compute message identifiers for `list.suspect` events. Any number of calls to `fmd_hdl_opendict()` may be made if the client module uses multiple event dictionaries. The dictionary is opened using the same rules described for the `dictionary` configuration file directive (see "2.7.2 dictionary" on page 35). If the specified dictionary cannot be found, a module abort will be triggered.

2.11 Resource Topology

The fault manager reference implementation includes the library `libtopo.so` to provide a standardized set of interfaces between platform software that probe and enumerate the *topology* of a set of resources, and higher-level fault management software such as diagnosis engines and agents. The primary consumer of the topology information is the Eversholt diagnosis software provided with the reference implementation. See Chapter 9, "Topology" for information about topology and topology APIs.

2.11.1 fmd_hdl_topo_hold

`topo_hdl_t *fmd_hdl_topo_hold(fmd_hdl_t *`*handle*`, int `*version*`)`

Return a handle for accessing the topology library `libtopo.so`. The *version* indicates the library API version to be used, and should be specified as `TOPO_VERSION`. The handle represents the most recent snapshot delivered to the module. See "2.8.7 fmdo_topo" on page 38 for more information. The handle must be explicitly released by `fmd_hdl_topo_rele()`.

2.11.2 fmd_hdl_topo_rele

```
void fmd_hdl_topo_rele(fmd_hdl_t *handle, topo_hdl_t *topology)
```

Releases a topology handle returned from `fmd_hdl_topo_hold()`. Every call to `fmd_hdl_topo_hold()` must be accompanied by a matching `fmd_hdl_topo_rele()`. If a module abort occurs while a topology handle is open, the handle is automatically released.

2.12 Memory Allocation

The fault manager provides a set of functions for dynamic memory allocation and string duplication for the convenience of its clients. These functions are provided to ease programming, fault injection, and testing of `fmd` modules, and to permit `fmd` to implement accounting of dynamic memory allocation and enforce an upper bound on the amount of dynamic memory that a client module can allocate. This is by definition a limit that can only be enforced insofar as the client modules use these interfaces for memory allocation; this is recommended as it helps the fault manager guard against modules that leak memory. The Solaris reference implementation uses `libumem.so.1` to implement the memory allocation routines, offering developers additional debugging facilities; see Chapter 13, "Debugging" for more information.

2.12.1 fmd_hdl_alloc

```
void *fmd_hdl_alloc(fmd_hdl_t *handle, size_t size, int flag)
```

Allocate *size* bytes of memory and return the address of the start of this memory. The memory is aligned to permit storage of the largest C data structure, and no guarantees are made about its initial contents. If *flag* is specified as `FMD_NOSLEEP`, then failure to allocate the required amount of memory will cause `fmd_hdl_alloc()` to fail and return a `NULL` pointer. If *flag* is specified as `FMD_SLEEP`, then failure to allocate the required amount of memory will cause `fmd` to sleep repeatedly, initiating garbage-collection activities and then retrying the allocation. In this mode, `fmd_hdl_alloc()` is guaranteed to succeed; if `fmd` is unable to allocate the memory after a predefined maximum upper bound on sleep time, it will attempt to restart itself in order to free up memory. In either mode, if a *size* of zero is specified, `fmd_hdl_alloc` will succeed and return `NULL`. If the new allocation added to the current amount of memory allocated by the module exceeds a configurable threshold, a module abort will be triggered.

2.12.2 fmd_hdl_zalloc

```
void *fmd_hdl_zalloc(fmd_hdl_t *handle, size_t size, int flag)
```

Allocate *size* bytes of memory as if by `fmd_hdl_alloc()`, and then fill the contents of the allocation with zeroes if the result is not a `NULL` pointer.

2.12.3 fmd_hdl_free

`void fmd_hdl_free(fmd_hdl_t *`*handle*`, void *`*data*`, size_t` *size*`)`

Deallocate the *size* bytes referred to by the *data* pointer, which should have been obtained using a previous call to `fmd_hdl_alloc()` or `fmd_hdl_zalloc()`. The size must exactly match the size used to allocate the buffer; if it does not, a module abort or failure of `fmd` will be triggered. It is legal to free a `NULL` *data* pointer by specifying a *size* of zero. It is not legal to perform a partial or duplicate free; these errors will cause `fmd` to fail or will trigger a module abort.

2.12.4 fmd_hdl_strdup

`char *fmd_hdl_strdup(fmd_hdl_t *`*handle*`, const char *`*string*`, int` *flag*`)`

Duplicate the specified *string* by allocating memory as if by `fmd_hdl_alloc()` for the length of the string plus an additional byte for the trailing `\0`, and then copy the source string into this newly allocated memory. The address of the new string is returned. If *flag* is `FMD_NOSLEEP`, the function can fail and return `NULL`, similar to `fmd_hdl_alloc`. If *flag* is `FMD_SLEEP`, the function sleeps for memory similar to `fmd_hdl_alloc()` and cannot fail. If a `NULL` *string* is specified, `fmd_hdl_strdup()` always succeeds and returns `NULL`.

2.12.5 fmd_hdl_strfree

`void fmd_hdl_strfree(fmd_hdl_t *`*handle*`, char *`*string*`)`

Free the memory associated with *string*, where *string* must refer to the result of a previous call to `fmd_hdl_strdup()`. If *string* is `NULL`, this function always succeeds and has no effect.

Note – The Solaris reference implementation of the fault manager uses `libumem.so.1` to perform memory allocation, and therefore internally computes the size of the string to be freed by applying `strlen()` to it. Therefore, callers of `fmd_hdl_strfree()` should take care not to insert additional `\0` characters in the string or to remove the trailing `\0`.

2.13 Debugging Support

The fault manager provides several utility routines that can be called from module source code to assist developers in debugging modules during their development or post-mortem. More information about the use of these functions is found in Chapter 13, "Debugging."

2.13.1 fmd_hdl_abort

`void fmd_hdl_abort(fmd_hdl_t *`*handle*`, const char *`*format*`, ...)`

Trigger a module abort as if the module contained a programming error and record an error string using the specified *format*. The *format* and any additional arguments are formatted using `snprintf`(3C). If the format string does *not* contain a newline (`\n`) character, the message will have the text "`:` *reason*" appended to it, where *reason* will be the `strerror`(3C) string corresponding to the current value of `errno`. The module's `_fmd_fini()` entry point will be called (assuming the caller is not `_fmd_init()` or `_fmd_fini()`) and the module will be disabled and marked as failed. Programmers are encouraged to use `fmd_hdl_abort()` for assertions, but should carefully distinguish between assertions of correct program behavior and handling of input errors such as malformed events, where it is more appropriate to drop the invalid input and drive on.

2.13.2 fmd_hdl_vabort

`void fmd_hdl_vabort(fmd_hdl_t *`*handle*`, const char *`*format*`, va_list` *ap*`)`

Trigger a module abort as if by `fmd_hdl_abort()`, but specify the *format* arguments using a preconstructed argument list.

2.13.3 fmd_hdl_error

`void fmd_hdl_error(fmd_hdl_t *`*handle*`, const char *`*format*`, ...)`

Generate a module error event that records an error string using the specified *format*. This function is similar to `fmd_hdl_abort()`, but can be used in situations where the module has detected an error but can safely continue execution. The *format* and any additional arguments are formatted using `snprintf`(3C). If the format string does *not* contain a newline (`\n`) character, the message will have the text "`:` *reason*" appended to it, where *reason* will be the `strerror`(3C) string corresponding to the current value of `errno`.

2.13.4 fmd_hdl_verror

`void fmd_hdl_verror(fmd_hdl_t *`*handle*`, const char *`*format*`, va_list` *ap*`)`

Record a module error as if by `fmd_hdl_error()`, but specify the *format* arguments using a preconstructed argument list.

2.13.5 fmd_hdl_debug

`void fmd_hdl_debug(fmd_hdl_t *`*handle*`, const char *`*format*`, ...)`

Record a debug message for live or post-mortem analysis using the specified *format*. The *format* and any additional arguments are formatted using `snprintf`(3C). The message will either be printed to `stderr`, logged to `syslogd`(1M), or recorded in an in-memory tracing buffer depending on the current debug settings; see Chapter 13, "Debugging" for more information.

2.13.6 fmd_hdl_vdebug

`void fmd_hdl_vdebug(fmd_hdl_t *`*handle*`, const char *`*format*`, va_list` *ap*`)`

Record a debug message as if by `fmd_hdl_debug()`, but specify the *format* arguments using a preconstructed argument list.

2.14 Property Retrieval

Module properties are defined as part of the call to `fmd_hdl_register()`, described earlier. The properties initially assume any value specified as part of the `fmd_prop_t` definition, and then are subsequently modified by any corresponding `setprop` directives found in the module's configuration file. Module writers can then use the functions described in this section to retrieve the property values according to their types. Property values are *not* restored by the module checkpoint mechanism; on module restart, properties assume values based on the current version of the module and the current configuration file. If a module uses properties to size data structures that are serialized, the property values should be recorded in these serialized data structures. The module is free to define any appropriate policy as to whether it will respect the original or new property values if they have changed since the first checkpoint of the module.

In addition to the properties defined by the module, the fault manager exports several global properties whose names are prefixed with the reserved string "`fmd.`" into the property namespace of each module. These properties can also be retrieved using the programming interfaces described in this section, but they cannot be modified by `setprop` statements in a module configuration file. The global fault manager properties are:

TABLE 2–3 Global Properties

Name	Type	Description
`fmd.isaname`	`string`	Processor ISA name. By default, the value of this property is the same as the output of the `uname -p` command.

TABLE 2–3 Global Properties *(Continued)*

Name	Type	Description
`fmd.machine`	`string`	Machine class name. By default, the value of this property is the same as the output of the `uname -m` command.
`fmd.platform`	`string`	Platform name string. By default, the value of this property is the same as the output of the `uname -i` command.
`fmd.rootdir`	`string`	Root directory for pathname expansions. By default, this property is set to the empty string, causing pathname expansions to refer to the root "/".

2.14.1 fmd_prop_get_int32

`int32_t fmd_prop_get_int32(fmd_hdl_t *`*handle*`, const char *`*name*`)`

Return the value of the specified property *name*, where the property must be previously defined to be of type `FMD_TYPE_BOOL`, `FMD_TYPE_INT32`, or `FMD_TYPE_UINT32`. If the specified property is not defined by the module or is not one of these types, a module abort is triggered.

2.14.2 fmd_prop_get_int64

`int64_t fmd_prop_get_int64(fmd_hdl_t *`*handle*`, const char *`*name*`)`

Return the value of the specified property *name*, where the property must be previously defined to be of type `FMD_TYPE_INT64`, `FMD_TYPE_UINT64`, `FMD_TYPE_TIME`, or `FMD_TYPE_SIZE`. If the specified property is not defined by the module or is not one of these types, a module abort is triggered.

2.14.3 fmd_prop_get_string

`char *fmd_prop_get_string(fmd_hdl_t *`*handle*`, const char *`*name*`)`

Return the value of the specified property *name*, where the property must be previously defined to be of type `FMD_TYPE_STRING`. If the specified property is not defined by the module or is not of this type, a module abort is triggered. This function makes a copy of the string associated with the property and returns a pointer to the copy. The caller is responsible for freeing the result with `fmd_prop_free_string()`.

2.14.4 fmd_prop_free_string

`void fmd_prop_free_string(fmd_hdl_t *`*handle*`, char *`*string*`)`

Free the specified property *string*, which is required to be a pointer obtained using a previous call to `fmd_prop_get_string()`.

2.15 Statistics

Modules are permitted to publish a set of Private named statistics, similar to `kstats`, that can be used to aid in debugging and facilitate observability in the field or by operations or manufacturing. Each statistic is assigned a unique string name which must conform to the same naming rules as those described for property names. The current statistic values can be retrieved from a running fault manager using the `fmstat`(1M) utility. Statistics are manipulated by simply modifying the actual `fmd_stat_t` structure corresponding to the statistic, which is defined as follows:

```
typedef struct fmd_stat {
    char fmds_name[32];    /* statistic name */
    uint_t fmds_type;      /* statistic type */
    char fmds_desc[64];    /* statistic description */
    union {
        int bool;           /* FMD_TYPE_BOOL */
        int32_t i32;        /* FMD_TYPE_INT32 */
        uint32_t ui32;      /* FMD_TYPE_UINT32 */
        int64_t i64;        /* FMD_TYPE_INT64 */
        uint64_t ui64;      /* FMD_TYPE_UINT64, TIME, SIZE */
        char *str;          /* FMD_TYPE_STRING */
    } fmds_value;
} fmd_stat_t;
```

2.15.1 fmd_stat_create

`fmd_stat_t *fmd_stat_create(fmd_hdl_t *`*handle*`, uint_t `*flag*`, uint_t `*statc*`, fmd_stat_t *`*statv*`)`

Publish the statistics described by the *statv* array of *statc* `fmd_stat_t` structures. The statistic name, type, and description must be filled in prior to the call, and the initial value is also set according to the value in specified array. The name must conform to the same naming rules as property names, and duplicate names are not permitted within a given module. The statistic type should be set to one of the types shown earlier in Table 2–1. If a duplicate name or invalid type is detected, a module abort will be triggered.

If *flag* is set to `FMD_STAT_NOALLOC`, then the caller's *statv* array is used as the actual data storage for the statistics and this pointer value is returned. If *flag* is set to `FMD_STAT_ALLOC`, `fmd` allocates new memory for the statistics storage and initializes it using the descriptions in the *statv* array,

and a pointer to the new memory is returned. The caller is free to increment or otherwise modify statistic values at any time by modifying the result of `fmd_stat_create()`. The fault manager will automatically call `fmd_stat_destroy()` for the module on all statistics that are still published at the time `fmd_hdl_unregister()` is called or `_fmd_fini()` completes.

2.15.2 fmd_stat_destroy

`void fmd_stat_destroy(fmd_hdl_t *`*handle*`, uint_t `*statc*`, fmd_stat_t *`*statv*`)`

Unpublish the statistics described by the *statv* array of *statc* `fmd_stat_t` structures and free any associated data storage. The *statv* pointer must correspond to memory associated with a previous call to `fmd_stat_create()`. Note that it *is* legal to destroy statistics one-by-one or in an order different from their creation, so long as each statistic is only destroyed once. If `fmd_stat_destroy()` is applied to a statistic that is not currently published, a module abort is triggered. Note that only the statistic name is considered by this function; the description and values associated with the statistics are ignored by this call. Finally, the fault manager will automatically call `fmd_stat_destroy()` for the module on all statistics that are still published at the time `fmd_hdl_unregister()` is called or `_fmd_fini()` completes, so explicit use of this function may not be necessary in most modules.

2.15.3 fmd_stat_setstr

`void fmd_stat_setstr(fmd_hdl_t *`*handle*`, fmd_stat_t *`*stat*`, const char *`*string*`)`

Assign the specified *string* as the new value of the specified `FMD_TYPE_STRING` statistic. The previous value of the statistic is deallocated as if by `fmd_hdl_strfree()` and the new value is allocated as if by `fmd_hdl_strdup`(*handle, string*, `FMD_SLEEP`). Any memory allocated by `fmd_stat_setstr()` will be automatically deallocated when the module handle is unregistered or when the statistic is destroyed using `fmd_stat_destroy()`.

2.16 Case Management

Modules that implement diagnosis algorithms are expected to associate incoming telemetry data and eventual diagnosis results with one or more *cases*, which act as a kind of metaphorical folder used to organize information relevant to a particular problem. Every case is named by a Universal Unique Identifier (UUID) which is recorded in any `list.suspect`, `list.isolated`, and `list.repaired` events associated with this case. Modules are free to create as many cases as needed and manage them in any way appropriate to the diagnosis algorithm. The fault manager considers each case to be in one of the following states at any given time:

Unsolved — The case has no `list.suspect` event associated with it and `fmd_case_solve()` has not yet been called.

Solved	The case has been *solved* as a result of the module adding one or more suspect faults to it and then calling `fmd_case_solve()`. When a case is solved, a conviction policy is applied to the suspects and a `list.suspect` event for the case is published through the event dispatch mechanism.
Close_Wait	The case is transitioning to either the Closed or Repaired state, and the diagnosis engine owning the case is still processing it. The fault manager places the case in the Close_Wait state to indicate that it has scheduled the execution of the `fmdo_close` entry point for the case. When `fmdo_close` returns, the case will transition to either the Closed state or the Repaired state, or it will be discarded if the case was never solved.
Closed	The case has been closed because it was solved and an agent has acted on the resulting suspect list by disabling the affected system ASRU(s). When a case reaches the Closed state, the fault manager dispatches a `list.isolated` event indicating that all ASRUs have been disabled.
Repaired	The problem associated with this case has been repaired or removed from the system. A case will transition to the Repaired state when `fmadm repair` is used to manually indicate a repair has taken place, or when the fault manager observes that the associated resources are no longer present in the system. When a case reaches the Repaired state, the fault manager dispatches a `list.repaired` event for the case and then frees the data structures associated with the case.

The state transitions are irreversible: once a case is Solved, it cannot be Unsolved. Once a case is Repaired, all of its state is deallocated inside of the fault manager as it is no longer needed (although the results persist in the fault manager log files). Further details of case states are discussed in the documentation for the various case manipulation functions. In addition to suspect list information, a case also maintains a reference to any telemetry event that has been *recorded* in the case. Events that are recorded will no longer be replayed by `fmd` on restart, and references to these telemetry events will be placed into the fault log with the eventual `list.suspect` event to aid in debugging the diagnosis algorithm. Further details of event processing are discussed in Chapter 3, “Events.”

Similar to module handles, case handles are opaque to module programmers and should not be dereferenced. Cases can be manipulated using their handles or by UUID (functions that are prefixed with “`fmd_case_uu`”). The fault manager will not permit any module other than the one that created the case to manipulate the case using its handle; other modules such as response agents may only manipulate cases using their UUID. If a module attempts an operation on a case handle which is not valid or which it does not own, a module abort will be triggered.

2.16.1 fmd_case_open

```
fmd_case_t *fmd_case_open(fmd_hdl_t *handle, void *data)
```

Open a new case and return an opaque handle for the case. The case is automatically assigned a new UUID as its name; the UUID can be retrieved using `fmd_case_uuid()`. The case is initially empty of suspects and events and begins life in the Unsolved state. Every case also has a module-specific data pointer which can be used to associate a private data structure with case; this pointer is initially assigned the value of *data*. The `fmd_case_t` data structure is opaque to clients and cannot be dereferenced. The case handle returned by `fmd_case_open()` is valid until the `fmdo_close` entry point is called for this case, but is not valid across restarts of the module. Although cases are checkpointed automatically by `fmd`, the case handles will be different on restart and the case-specific data pointer is reset to `NULL`. Modules can rediscover existing cases by iterating over them using `fmd_case_next()` from their `_fmd_init()` entry point.

2.16.2 fmd_case_reset

```
void fmd_case_reset(fmd_hdl_t *handle, fmd_case_t *case)
```

Reset an Unsolved case by clearing any suspects and events that have been added to it since it was opened. The case-specific data pointer is not affected by this call. If the case is Solved or Closed, a module abort will be triggered.

2.16.3 fmd_case_solve

```
void fmd_case_solve(fmd_hdl_t *handle, fmd_case_t *case)
```

Solve a case, apply a conviction policy to the suspect list, generate a `list.suspect` event for the case, and publish this event to any subscribing modules. The `list.suspect` event will contain the suspect list added to the case using `fmd_case_add_suspect()`, and will contain an FMRI for the diagnosis engine module generated automatically using the information from `fmd_hdl_register()`. A `libdiagcode` message identifier will also be automatically generated for the suspect list and added to the event based upon the suspect faults that have been previously associated with the case. If no event dictionary has been named by this module or the event dictionary does not contain a matching entry for this suspect list, a generic message identifier indicating that the module is broken or misconfigured is added to the suspect list.

A conviction policy is applied by the fault manager to the suspect list prior to publishing the `list.suspect` event to determine which suspects are marked as faulty in the event and in the resource cache. At present, the conviction policy convicts all suspects established by the diagnosis engine. Future versions of the fault manager may provide the ability for configurable

and/or dynamic conviction policies. Diagnosis engines should not encode any assumptions as to the nature or behavior of the conviction policy.

2.16.4 fmd_case_close

```
void fmd_case_close(fmd_hdl_t *handle, fmd_case_t *case)
```

Indicate that the specified *case* is to be closed and schedule a call to the `fmdo_close` entry point. If the case is Unsolved, this call indicates that the module wishes to discard the case. If the case is Solved, this call indicates that any ASRUs associated with convicted suspects have been disabled, and therefore no more error reports associated with the case are expected. The case transitions to the Close_Wait state immediately. If the case has been solved, it will then transition to the Closed state following the completion of the `fmdo_close` entry point.

2.16.5 fmd_case_uuid

```
const char *fmd_case_uuid(fmd_hdl_t *handle, fmd_case_t *case)
```

Return the string form of the UUID that forms the unique name for the specified *case*.

2.16.6 fmd_case_uulookup

```
fmd_case_t *fmd_case_uulookup(fmd_hdl_t *handle, const char *uuid)
```

Return the opaque handle for the case named by the specified *uuid*. If the case does not exist or is not owned by the calling module, a `NULL` handle is returned.

2.16.7 fmd_case_uuclose

```
void fmd_case_uuclose(fmd_hdl_t *handle, const char *uuid)
```

Close the specified case, named by its *uuid*. This function is identical to `fmd_case_close()`, but permits modules other than the case's owner (that is, response agents) to close cases. If the case does not exist, the function call has no effect (that is, we cannot distinguish between an already closed and no longer persisted valid *uuid* and one which is not valid at all).

2.16.8 fmd_case_uuclosed

```
int fmd_case_uuclosed(fmd_hdl_t *handle, const char *uuid)
```

Return a boolean value indicating if the specified case, named by its *uuid*, is currently in the Closed state.

Note – This function also returns non-zero if the case is not known, because the fault manager cannot distinguish between a legitimate *uuid* that has been closed and freed and one that is simply unknown or invalid. This behavior should not be relied upon, and may change in a future version of the fault manager.

2.16.9 fmd_case_solved

`int fmd_case_solved(fmd_hdl_t *`*handle*`, fmd_case_t *`*case*`)`

Return a boolean value indicating if the specified *case* is currently in the Solved state.

2.16.10 fmd_case_closed

`int fmd_case_closed(fmd_hdl_t *`*handle*`, fmd_case_t *`*case*`)`

Return a boolean value indicating if the specified *case* is currently in the Closed state. Closed cases are not actually deallocated until the `fmdo_close` entry point runs to completion.

2.16.11 fmd_case_add_ereport

`void fmd_case_add_ereport(fmd_hdl_t *`*handle*`, fmd_case_t *`*case*`, fmd_event_t *`*event*`)`

Add the specified *event* to the specified *case*, indicating that it no longer needs to be replayed by the fault manager on restart, and that any diagnosis resulting from this case should reference the specified event. The *event* handle should be obtained from the corresponding parameter of the `fmdo_event` entry point. Note that it is legal to continue adding events to a case once it is Solved (as we expect to continue to receive error telemetry from a faulty component until it is disabled and the case is Closed), but no further cross-references for such events will be added to `/var/fm/fmd/fltlog` once the case is Solved.

2.16.12 fmd_case_add_serd

`void fmd_case_add_serd(fmd_hdl_t *`*handle*`, fmd_case_t *`*case*`, const char *`*serd*`)`

Add all of the events referenced by the specified *serd* engine to the specified *case*, as if `fmd_case_add_ereport()` had been called on each one. If the specified engine does not exist, a module abort is triggered.

2.16.13 fmd_case_add_suspect

`void fmd_case_add_suspect(fmd_hdl_t *`*handle*`, fmd_case_t *`*case*`, nvlist_t *`*fault*`)`

Add the specified *fault* to the specified *case* as a likely suspect for causing the problem associated with this case. The *fault* should be a well-formed fault event according to the FMA Event Protocol; if required members are missing, a module abort will be triggered. If the specified *case* is not currently Unsolved, a module abort will be triggered. A module is free to add any number of suspect faults to case, but it should obviously attempt to name as few suspects as possible.

2.16.14 fmd_case_setspecific

`void fmd_case_setspecific(fmd_hdl_t *`*handle*`, fmd_case_t *`*case*`, void *`*data*`)`

Set the case-specific data pointer to be the specified *data* pointer, replacing any previous value. The case-specific data pointer is not maintained across checkpoints. If the previous value pointed to dynamically allocated memory, the caller is responsible for deallocating this memory first if appropriate. The caller is also responsible for deallocating any such memory stored here in the `_fmd_fini` entry point.

2.16.15 fmd_case_getspecific

`void *fmd_case_getspecific(fmd_hdl_t *`*handle*`, fmd_case_t *`*case*`)`

Return the current value of the case-specific data pointer, which is initially assigned using `fmd_case_open()` and can be later modified using `fmd_case_setspecific()`.

2.16.16 fmd_case_setprincipal

`void fmd_case_setprincipal(fmd_hdl_t *`*handle*`, fmd_case_t *`*case*`, fmd_event_t *`*event*`)`

Set the *principal event* for the specified *case* to be the specified *event*. Each case may have at most one principal event whose event handle is saved in the case checkpoint and can be retrieved with `fmd_case_getprincipal()`. If the specified *case* already had a principal event defined, the previous principal event handle is replaced by the new event handle. The value of *event* is opaque to the client module and a different value may be returned from `fmd_case_getprincipal()` than the one that was passed to `fmd_case_setprincipal()`. The principal event handle can be used to re-install relative event timers after a module is restored from its checkpoint.

2.16.17 fmd_case_getprincipal

```
fmd_event_t *fmd_case_getprincipal(fmd_hdl_t *handle, fmd_case_t *case)
```

Return an opaque handle to the principal event associated with the specified *case*. If no previous event handle had been installed for the case with `fmd_case_setprincipal()`, this function returns `NULL`. The value returned by `fmd_case_getprincipal()` is an opaque handle that should not be saved in any buffers or assumed to remain constant across checkpoint/restore cycles of the calling module.

2.16.18 fmd_case_next

```
fmd_case_t *fmd_case_next(fmd_hdl_t *handle, fmd_case_t *case)
```

Iterate over the set of cases associated with the calling module by returning the case created after the specified *case*. If *case* is `NULL`, the first case is returned. If there are no cases or the last case is specified, `NULL` is returned. The caller should typically perform a complete iteration within a module entry point and not attempt to cache case handles across calls to different entry points as the list may change.

2.16.19 fmd_case_prev

```
fmd_case_t *fmd_case_prev(fmd_hdl_t *handle, fmd_case_t *case)
```

Iterate over the set of cases associated with the calling module by returning the case created before the specified *case*. If *case* is `NULL`, the last case is returned. If there are no cases or the first case is specified, `NULL` is returned. The caller should typically perform a complete iteration within a module entry point and not attempt to cache case handles across calls to different entry points as the list may change.

2.17 Buffer Management

The fault manager provides a set of interfaces for clients to create named *buffers* for serializing data structures to be stored in checkpoints. The module is entirely responsible for the format of any data stored in a buffer, and for providing any useful `mdb`(1) debugging support necessary to debug problems in any complex data structures. As the primary purpose of the buffers is for storing persistent information across module restarts, elements that can change across restarts such as module handles, case handles, event handles, and pointers to any module data structures should *not* be stored in buffers. Modules are also responsible for *versioning* any data in buffers appropriately, if it is desirable that a new version of a module be able to process the state associated with an older version of the module. Versioning can take the form of storing

integer or string version numbers in the serialized buffer data structure, and should be explained in any accompanying project documentation for the module. Buffers are named using strings that must conform to the same naming rules used for property names, described earlier.

2.17.1 fmd_buf_create

```
void fmd_buf_create(fmd_hdl_t *handle, fmd_case_t *case, const char *name, size_t size)
```

Create a new buffer with the specified *name* and *size* in bytes. The buffer is initially filled with zeroes. If a *case* handle is specified, the buffer is added to a namespace associated with the case. If a `NULL` *case* is specified, the buffer is added to a namespace associated with the module. If the buffer *name* already exists in the corresponding namespace, a module abort is triggered. If a zero *size* is specified, a module abort is triggered. If the total amount of buffer space exceeds a configurable threshold, a module abort is triggered.

2.17.2 fmd_buf_destroy

```
void fmd_buf_destroy(fmd_hdl_t *handle, fmd_case_t *case, const char *name)
```

Destroy the buffer with the specified *name*. If the buffer does not exist in the corresponding namespace, this function has no effect.

2.17.3 fmd_buf_read

```
void fmd_buf_read(fmd_hdl_t *handle, fmd_case_t *case,
    const char *name, void *buf, size_t size)
```

Copy *size* bytes of data from the buffer specified by *name* in the namespace specified by *case* into the specified *buf*. If the specified *size* exceeds the size of the buffer, the remainder of the caller's *buf* is filled with zeroes. If the specified buffer does not exist, a module abort is triggered. All buffer reads begin from offset zero of the buffer, but a *size* smaller than the buffer size can be specified.

2.17.4 fmd_buf_write

```
void fmd_buf_write(fmd_hdl_t *handle, fmd_case_t *case,
    const char *name, const void *buf, size_t size)
```

Copy *size* bytes of data from the buffer specified by *buf* to the buffer specified by *name* in the namespace specified by *case*. If the specified *size* exceeds the size of the buffer, a module abort is

triggered. If the specified buffer does not exist, a module abort is triggered. All buffer writes occur at offset zero of the buffer, but a *size* smaller than the buffer size can be specified, in which case the remainder of the buffer is left unmodified.

2.17.5 fmd_buf_size

`size_t fmd_buf_size(fmd_hdl_t *`*handle*`, fmd_case_t *`*case*`, const char *`*name*`)`

Return the size in bytes of the buffer specified by *name* in the namespace specified by *case*. If the specified buffer does not exist in the namespace, a size of zero is returned.

2.18 SERD Engines

The fault manager provides interfaces for diagnosis engines to create and update Soft Error Rate Discrimination (SERD) engines that are automatically checkpointed and restored on behalf of the module. SERD engines are in effect ring buffers of events that can be used to determine whether more than N events in some time T have been seen, indicating an anomaly or discriminating expected soft upsets from faults. SERD engines are named by the module using arbitrary ASCII strings and kept in a separate namespace associated with each module. The `fmadm`(1M) and `fmstat`(1M) utilities provide facilities for viewing and manipulating the state of a module's SERD engines. The fault manager will also periodically garbage collect out-of-date entries in a module's SERD engines prior to invoking the module's `fmdo_gc` entry point, or when the fault manager is running low on memory. Module writers should use properties, described earlier, rather than hardcoded constants, to initialize SERD engines so that manufacturing and operations can modify the default settings during hardware or software testing.

At any given time, a SERD engine is said to be *pending* or *fired*, indicating whether or not sufficient events have been inserted to trigger the >N in time T threshold. Once an engine fires, its state freezes and no further events will be recorded by the engine (so that those triggering the engine firing can be viewed post-mortem). The engine can be reset to the pending state using `fmd_serd_reset()`.

2.18.1 fmd_serd_create

`void fmd_serd_create(fmd_hdl_t *`*handle*`, const char *`*name*`, uint_t` *N*`, hrtime_t` *T*`)`

Create a new SERD engine using the specified *name* which will fire after *greater than N* events in *T* nanoseconds have been inserted into the engine. If a SERD engine with the specified *name* already exists, a module abort is triggered.

2.18.2 fmd_serd_destroy

```
void fmd_serd_destroy(fmd_hdl_t *handle, const char *name)
```

Destroy the SERD engine specified by its *name*. Any events associated with the engine are in effect discarded by the module (see Chapter 3, "Events" for more details on event management). If no engine by that *name* exists, this function has no effect.

2.18.3 fmd_serd_reset

```
void fmd_serd_reset(fmd_hdl_t *handle, const char *name)
```

Remove all events from the specified SERD engine and reset the engine's state to Pending, permitting additional events to be added. If the specified engine does not exist, a module abort is triggered.

2.18.4 fmd_serd_record

```
int fmd_serd_record(fmd_hdl_t *handle, const char *name, fmd_event_t *event)
```

Record the specified *event* in the SERD engine and return a boolean value indicating whether or not the engine fired as the result of inserting this event. Once an event is inserted into an engine, it will not be replayed on restart by the fault manager. If the specified engine has already fired, the event is discarded and zero is returned. If the engine contains out-of-date events, they are discarded as a side-effect of this function. If the specified engine does not exist, a module abort is triggered. Note that the determination of T is performed by comparing the times associated with the events inserted into the engine, and not by the current adjustable time-of-day; for more information on event times, see Chapter 3, "Events."

2.18.5 fmd_serd_fired

```
int fmd_serd_fired(fmd_hdl_t *handle, const char *name)
```

Return a boolean value indicating whether or not the specified SERD engine is currently in the Fired state. If the specified engine does not exist, a module abort is triggered.

2.18.6 fmd_serd_empty

```
int fmd_serd_empty(fmd_hdl_t *handle, const char *name)
```

Return a boolean value indicating whether or not the specified SERD engine is currently empty (that is, has no events in it). If the specified engine does not exist, a module abort is triggered.

2.18.7 fmd_serd_exists

`int fmd_serd_exists(fmd_hdl_t *`*handle*`, const char *`*name*`)`

Return a boolean value indicating whether or not the specified SERD engine exists in the module's collection of SERD engines.

2.19 Timers

The fault manager provides a simple set of routines for modules to install one-shot timers that fire after a specified number of nanoseconds. Timers can be installed to run either in a specified number of nanoseconds, or a specified number of nanoseconds after a particular event occurred. Timers are *not* checkpointed by the fault manager, unlike most other module services.

2.19.1 fmd_timer_install

`id_t fmd_timer_install(fmd_hdl_t *`*handle*`, void *`*data*`, fmd_event_t *`*event*`, hrtime_t` *delta*`)`

Install a new timer which will fire *at least delta* nanoseconds after the specified *event* occurred. If *event* is `NULL`, the timer will fire at least *delta* nanoseconds from the current time. If the timer expiry refers to a time that has already passed, the timer will fire as soon as possible. If the timer expiry time precedes any of the events currently waiting in the module's event queue, the timer will be processed prior to any of these events. To indicate that the timer has expired, the fault manager schedules a call to the module's `fmdo_timeout` entry point and passes the specified *data* to this function. If the *data* pointer refers to dynamically allocated memory, the caller is responsible for freeing this memory. This function returns an opaque integer identifier for the timer, which can be used to cancel it; the caller should not attempt to interpret the value of this identifier.

2.19.2 fmd_timer_remove

`void fmd_timer_remove(fmd_hdl_t` *handle*`,id_t` *id*`)`

Cancel the timer specified by *id*, which should correspond to the result of an earlier call to `fmd_timer_install()`. If the *id* does not refer to a valid timer owned by this module or refers to

a timer that has already expired, a module abort is triggered. The fault manager guarantees that the module will not receive a call to `fmdo_timeout` for the specified timer once this function returns.

2.20 Name-Value Pair Lists

The fault manager provides a set of utility functions to assist in the manipulation of name-value pair lists for events and FMRIs. These functions are designed to supplement the basic functions provided by `libnvpair.so.1`.

2.20.1 fmd_nvl_create_fault

```
nvlist_t *fmd_nvl_create_fault(fmd_hdl_t *handle, const char *class, uint8_t percent,
    nvlist_t *asru, nvlist_t *fru, nvlist_t *resource)
```

Create a new name-value pair list structure representing a `fault.*` event with the specified attributes and return a pointer to it. The new event is assigned the specified *class* and *percent* likelihood, and is associated with the specified FMRI values for *asru*, *fru*, and *resource*, any of which may optionally be set to `NULL`. The three FMRIs are copied into the new name-value pair list representing the fault event: the caller is responsible for deallocating these parameters if they are no longer needed.

2.20.2 fmd_nvl_class_match

```
int fmd_nvl_class_match(fmd_hdl_t *handle, nvlist_t *event, const char *class)
```

Return a boolean value indicating if the specified *event* is an FMA Protocol Event whose event class string matches the specified *class*. The *class* may use the "*" character as a wildcard, as described above for the `subscribe` configuration file directive (see "2.7.1 subscribe" on page 34).

2.20.3 fmd_nvl_fmri_expand

```
int fmd_nvl_fmri_expand(fmd_hdl_t *handle, nvlist_t *fmri)
```

Expand the specified *fmri* name-value pair list by calling the corresponding resource scheme's `fmd_fmri_expand()` routine, if one is defined, to add appropriate members to the FMRI that were not known when it was originally created. For example, a kernel error event producer may not know the complete serial number data for a DIMM and therefore this information can be

added by expanding the FMRI for the DIMM once it is received by the corresponding diagnosis engine. If the expansion succeeds, this function returns zero for success; otherwise a non-zero value is returned to indicate an error.

2.20.4 fmd_nvl_fmri_present

`int fmd_nvl_fmri_present(fmd_hdl_t *`*handle*`, nvlist_t *`*fmri*`)`

Return a boolean value indicating if the resource named by the specified *fmri* is present on the system. This function calls the corresponding resource scheme's `fmd_fmri_present()` routine to determine the result.

2.20.5 fmd_nvl_fmri_unusable

`int fmd_nvl_fmri_unusable(fmd_hdl_t *`*handle*`, nvlist_t *`*fmri*`)`

Return a boolean value indicating if the resource named by the specified *fmri* is unusable (that is, disabled either by the fault manager or by some administrative mechanism). This function calls the corresponding resource scheme's `fmd_fmri_unusable()` routine to determine the result.

2.20.6 fmd_nvl_fmri_faulty

`int fmd_nvl_fmri_faulty(fmd_hdl_t *`*handle*`, nvlist_t *`*fmri*`)`

Return a boolean value indicating if the resource named by the specified *fmri* is marked as faulty in the fault manager's resource cache. If no entry is present in the resource cache for the specified *fmri*, this function returns zero.

2.20.7 fmd_nvl_fmri_contains

`int fmd_nvl_fmri_contains(fmd_hdl_t *`*handle*`, nvlist_t *`*fmri1*`, nvlist_t *`*fmri2*`)`

Return a boolean value indicating if the resource named by the specified *fmri1* contains the resource named by the specified *fmri2* in the current system topology. See Chapter 9, "Topology" for information about topology. This function calls the corresponding resource scheme's `fmd_fmri_contains()` routine, if one is defined, to determine the result.

2.20.8 fmd_nvl_fmri_translate

`nvlist_t *fmd_nvl_fmri_translate(fmd_hdl_t *`*handle*`, nvlist_t *`*fmri*`, nvlist_t *`*authority*`)`

Translate the specified *fmri* into a form suitable for use in the fault region specified by the given FMRI *authority*. If the translation succeeds, a new name-value pair list is allocated for the translated FMRI and this value is returned; otherwise NULL is returned to indicate an error. The specified *fmri* and *auth* are not modified or deallocated by this function. This function calls the corresponding resource scheme's `fmd_fmri_translate()` routine, if one is defined, to determine the result.

2.21 Auxiliary Threads

The fault manager permits modules to create one or more *auxiliary threads* in addition to the thread(s) that execute the various module entry points. Auxiliary threads should be used with great caution and only when absolutely necessary, such as for the development of multi-threaded transport modules. Auxiliary threads should use the POSIX threads APIs to perform appropriate synchronization. Auxiliary threads must be created using `fmd_thr_create()`; threads created using the underlying system APIs such as `pthread_create()` will not be able to call the fault manager programming interfaces. The use of POSIX thread cancellations on auxiliary threads is discouraged; thread cancellations are disabled when auxiliary threads execute code in the fault manager.

2.21.1 fmd_thr_create

`pthread_t fmd_thr_create(fmd_hdl_t *`*handle*`, void (*`*function*`)(void *), void *`*arg*`)`

Create a new auxiliary thread for the module corresponding to the specified *handle*, arrange for it to be begin execution at the specified *function*, and return the POSIX thread identifier for the new thread. The specified *arg* is passed to this function as an argument. Auxiliary threads are created with all signals masked in their per-thread signal mask, except for `SIGABRT`, which is used for `assert()`. If a new thread cannot be created, or if the configurable limit on per-module auxiliary threads has been exceeded, a module abort is triggered.

2.21.2 fmd_thr_destroy

`void fmd_thr_destroy(fmd_hdl_t *`*handle*`, pthread_t `*id*`)`

To destroy an auxiliary thread, the module should first signal the thread using `fmd_thr_signal()` or some other mechanism such as a condition variable to request that the thread exit by calling `pthread_exit()`. The module can then call `fmd_thr_destroy()` to wait

for the thread to exit, join with it as if by `pthread_join()`, and free internal data structures associated with the auxiliary thread. If the thread has not yet exited, `fmd_thr_destroy()` will block waiting for the thread to exit before returning. If the specified thread *id* is invalid or refers to an auxiliary thread that does not belong to the calling module, a module abort will be triggered.

If `fmd_hdl_unregister()` is called or the `fmdo_fini` entry point completes and auxiliary threads associated with the module have not yet been destroyed by `fmd_thr_destroy()`, the fault manager will attempt to terminate any remaining auxiliary threads using cancellations and will then apply `fmd_thr_destroy()` to each remaining thread. Modules that employ auxiliary threads must take great care to write an `fmdo_fini` routine that cleans up all auxiliary threads and does not create a situation in which the module or fault manager will deadlock.

2.21.3 fmd_thr_signal

`void fmd_thr_signal(fmd_hdl_t *`*handle*`, pthread_t` *id*`)`

Send a software signal to the specified thread *id* as if by `pthread_kill()`. When a module creates auxiliary threads, these threads may be used to execute system calls that sleep interruptibly in the kernel. Examples of such interruptible sleep include executing an `ioctl()` to wait for a message from a Service Processor mailbox or executing a `read()` on a socket that has no data queued. In these circumstances, it is desirable to have some means to cause such system calls to be interrupted and return `EINTR`, allowing the thread to gracefully clean up and exit. The `fmd_thr_signal()` function signals the specified thread using some appropriate system signal that causes interruptible system calls to return `EINTR`; the module should not make assumptions about which signal is used. If the specified thread *id* is not a valid thread or does not belong to the calling module, a module abort will be triggered.

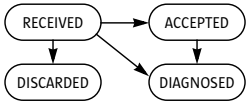
◆ ◆ ◆ CHAPTER 3

Events

The fault manager's primary responsibility is managing inbound telemetry events and dispatching these events to its clients, the collection of modules that are implementing diagnosis engines and response and messaging agents. Although the basics of how a module interacts with events were covered in Chapter 2, "Module API," we provide additional design and implementation details in this chapter. In particular, we discuss the state machine used for events and issues related to computing event times.

3.1 Event States

The fault manager uses the following state machine to represent the current state of every event it receives from the event transport or that is generated by any of its clients:



Events begin life in the Received state, which indicates that the event must be replayed on restart of the fault manager. If the event transitions to any other state, the replay group tag is reset as described in Chapter 5, "Log Files." Events are reference counted; as different events are placed on subscribing modules' queues, the reference count is incremented rather than copying the event, since events are read-only. If the reference count drops to zero and the current state is Received, the event automatically transitions to the Discarded state and is then deallocated. The event states are defined as follows:

Discarded — The event has been processed by all subscribers and no subscribers have elected to add this event to a SERD engine or case. As a result, the event can be safely deallocated by the fault manager.

Received — The event has been received by the event transport and is in the process of being dispatched to and received by subscribing modules. No subscribers have yet transitioned the event to the Accepted state.

Accepted — The event has been received by at least one subscriber who has added the event to a SERD engine or case. Once this transition occurs, the event will no longer be replayed.

Diagnosed — The event has been received by at least one subscriber who has associated the event with a case that has now been solved.

The state transitions are irreversible, and one state is maintained regardless of the number of subscribers. As a result, if one subscriber is able to transition to event to the Accepted or Diagnosed states and checkpoint, no other subscribers will have the event replayed even if they have not yet processed the event. Typically this is not a problem as in the system design, only one diagnosis engine is intended to diagnose a particular class of telemetry event.

If an event is received that has no subscribers, the fault manager automatically enqueues this event for its self-diagnosis engine. The self-diagnosis engine will associate a case with each event class that has no subscribers, and generate a `list.suspect` event with a message identifier indicating that there is a likely software defect or mismatch between the telemetry producers and fault management consumers (if, for example, a software package was mistakenly removed).

3.2 Event Times

The fault manager must take special care when it assigns times to events. Figure 3–1 shows a timeline of a fault and a resulting error. In addition to the absolute notion of time, the time of the steps A, B, C, D, and E can be read using either or both of two clocks, a non-adjustable clock corresponding to a processor cycle count, or an adjustable clock representing a time of day.

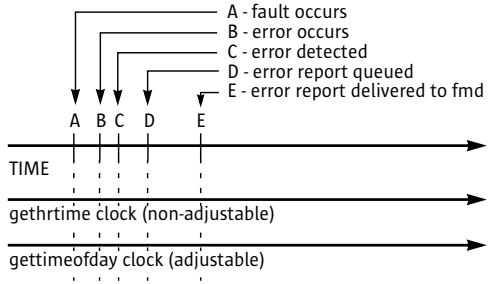


FIGURE 3–1 Event Time Diagram

As described in the *FMA Event Protocol* document, each error report is assigned an Error Numeric Association (ENA) value which uniquely identifies it during its trip through the producer code and transport until it reaches the fault manager. The ENA contains a (relatively) small number of timestamp bits that can identify the time at which an error was detected (labeled `C` in the event time diagram). As the diagram shows, this time is in fact an upper bound

on the time the error occurred and was then detected. The error (`B`) occurred some unknown time after the actual underlying fault that produced this symptom (labeled `A` in the event time diagram).

The fault manager expects its event transport to label each event with a time value when the event is queued with the transport (`D`) so that it can compute appropriate values of both high-resolution non-adjustable time and adjustable time-of-day for the actual error report. The design requires that the ENA timestamp contain sufficient timestamp bits to cover the time between `C` and `D` so that the fault manager can compute values of `C` for both of the clocks shown in the diagram by subtracting appropriate numbers of nanoseconds from times sampled at `E`.

The fault manager uses both types of clocks shown in the diagram for different purposes: the `gethrtime` non-adjustable clock is used for measuring relative time deltas, such as those required for interval timers and SERD engines. The `gethrtime` clock may only be used while events are active in the fault manager, because this clock only returns monotonically increasing relative values and resets to zero whenever the system reboots. When an event is written to a log file, the fault manager must write out the `gettimeofday` adjustable time value as this is the only clock that has persistent meaning when the fault manager is not running. When the fault manager replays events, it computes a new `gethrtime` clock value for the event by adjusting the saved time-of-day value according to a new, updated sample of the two clocks.

The design implies that both types of clocks need to be available for any fault manager implementation, and that the upper bound on SERD engines and interval timers must be capped at the maximum number of nanoseconds that can be represented in a 64–bit integer value (about 585 years). If the fault manager attempts to restore an event to a case or SERD engine that has aged beyond this point, the event is automatically *euthanized* and discarded, as it is extremely unlikely such an event is relevant to current activities (or to the original admin staff, who have been deceased for about 500 years).

◆ ◆ ◆ CHAPTER 4

Event Transports

A fault manager *transport* is the endpoint of a connection to some source of FMA Protocol events, along with a set of module routines that know how to send and receive events through this endpoint. As shown in Figure 1–1, the fault manager takes protocol events received by a transport and dispatches them to one or more modules based upon their subscriptions. Although only one transport is shown in this introductory diagram, the fault manager permits any number of transports to be active at a time, and transports themselves are also implemented by fault manager modules. This chapter provides advanced developers with information on fault manager transports, and describes additional programming APIs beyond those in Chapter 2, "Module API" that can be used to implement transport modules. The interfaces described in this chapter are also defined in `/usr/include/fm/fmd_api.h`.

4.1 Transport Semantics

A transport module provides routines that know how to transfer FMA Protocol events using some underlying mechanism such as a TCP/IP socket, door, or shared-memory buffer. The transport module must implement a mechanism to marshal and unmarshal a name-value pair list representing an FMA event and send the marshalled data to another transport module, representing a *peer*. The transport module is also responsible for providing any mechanisms needed to determine which peers to connect to and any directory or credential services that are required to establish the connection of the underlying mechanism. The fault manager implements a *common transport layer* on top of these base capabilities, including common services such as statistics and observability, connections to the fault manager log files, and queue management. Transports come in two flavors: read-only and read-write. Read-only transports are used when the event source represents only a collection of error handlers rather than another fault manager. Read-write transports are used when connecting to another fault manager, and provide bidirectional flow of arbitrary FMA events and control events that manage the transports.

4.1.1 Programming Model

The programming model for transport modules is relatively simple; the API details are found later in this chapter. The transport module is responsible for creating a new transport handle using `fmd_xprt_open()` to represent each virtual connection to an event source or remote peer. The transport handle does not need to be created until the underlying transport connection is established; modules are free to create and destroy transport handles at any time. Once the transport handle is active, the fault manager will call the module's `fmdo_send` entry point whenever the transport module should transport an event to the remote peer, and it expects the transport module to call `fmd_xprt_post()` whenever it receives an event from the remote peer. If the underlying connection experiences a transient failure, the transport module can call `fmd_xprt_suspend()` to request that the fault manager temporarily pause activity on the specified transport, and then `fmd_xprt_resume()` to resume it.

Unlike most fault manager modules, transport modules are inherently multi-threaded: the fault manager provides multiple threads to execute `fmdo_send` for simultaneously active transports, and the transport APIs themselves may be called in a multi-threaded fashion from any transport module auxiliary threads. The transport module is free to provide any appropriate threading scheme for the receive mechanism. For example, a transport module could provide one dedicated thread per transport to receive events, a fixed-size pool of threads multiplexing over the active transports, a single thread polling all transports, or could even use a `fmdo_timeout` routine from the main module thread to poll the transports.

A complete example of a read-write transport sending an event from one fault manager to another is shown in Figure 4–1:

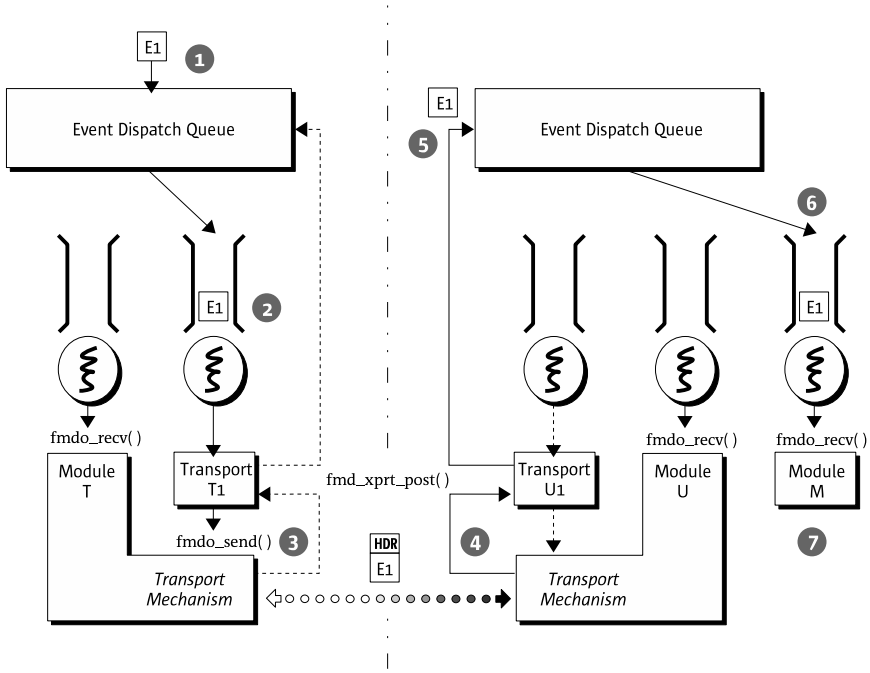


FIGURE 4–1 Fault Manager Transport Example

The annotations in the diagram illustrate the event flow for a sample event, E1, between two fault managers connected by a transport module. Solid lines are used to illustrate the path taken by E1. Dashed lines are used to indicate the transport direction that is not used in this example. The annotations in the diagram correspond to the following fault manager activities:

1. The event E1 is posted to the dispatch queue of the left-hand fault manager. This event could represent an event created within this fault manager, such as a `list.suspect` event, or an event received by another transport module, such as the `sysevent-transport`.
2. Transport module T has an open transport T1 that is proxying a subscription on behalf of its peer, transport U1, that matches the event class for event E1. As a result, a reference to event E1 is inserted into the outbound event queue for T1.
3. When E1 reaches the front of T1's outbound queue, module T's `fmdo_send` entry point is called for event E1. The underlying transport mechanism implemented by the module marshals the event name-value pair data and adds any necessary packet headers, and transmits the data to module U.
4. The underlying transport mechanism in module U receives the data for E1, unmarshals it into a name-value pair list, and calls `fmd_xprt_post()` on the event data.
5. The common transport layer verifies that the data for E1 represents a valid FMA Protocol event that is appropriate for the state of transport U1, decrements the time-to-live value, and posts the event to the dispatch queue of the right-hand fault manager.

6. Module M, running in the right-hand fault manager, has a subscription that matches the event class for event E1. As a result, a reference to event E1 is inserted into the inbound event queue for module M.
7. When E1 reaches the front of M's inbound queue, module M's `fmdo_recv` entry point is called as usual and module M processes the event just as it would for any locally received event.

4.1.2 Design Considerations

Transport module implementations must address the following design considerations:

Atomicity — The underlying transport mechanism for all fault manager transports must provide event atomicity in the form of *at-most-once* semantics. That is, if event E1 is passed to `fmdo_send`, then event E1 should be received by the remote peer transport module *at most* one time and `fmd_xprt_post()` should be called *at most once* for E1. However, the transport is free to drop events and not deliver them if a transient error condition is encountered. The fault manager keeps appropriate statistics on such conditions automatically.

Ordering — The underlying transport mechanism for all fault manager transports is required to guarantee *ordering* of all messages. That is, if event E1 is passed to `fmdo_send` before another event E2, then the receiver must call `fmd_xprt_post()` for E1 prior to calling `fmd_xprt_post()` for E2. For example, a naive transport module implemented on top of a UDP socket would *not* be a valid implementation because UDP datagrams may be delivered in any order in the event of any routing delays, etc.

Connectivity — The underlying transport mechanism for all fault manager transports must define an appropriate policy and design for detecting connection loss and determine how to effect connection recovery. If a connection is lost and recovered, the underlying transport mechanism is solely responsible for the replay of any in-transit events, but it may choose not to do so. If an event replay mechanism is provided, it must adhere to the *atomicity* and *ordering* constraints described above. If an event replay mechanism is not provided, all in-transit error events that were not yet recorded in the receiver's error log will be lost, and any in-transit fault and list events will be replayed by the common transport layer as described in "4.2 Transport Entry Point" on page 74.

Reliability — The underlying transport mechanism for all fault manager transports should make some reasonable provision for the reliability of message data. When appropriate, data should be checksummed, and data that was corrupted during transfer should not result in the remote peer crashing. The underlying transport mechanism is permitted to implement a capability to retry or replay

corrupted events, so long as this mechanism does not conflict with the *atomicity* and *ordering* constraints described above.

Security — If a transport module provides the ability to communicate with another fault region that has a different security context (for example, a Solaris Zone, Service Processor, or another operating system instance), it must make appropriate provisions for security and authentication. Specifically, if malicious users can spoof a fault manager connection, they may be able to inject fake events that cause the fault manager to disable services on the system, thereby constituting a denial-of-service attack. An appropriate description of the security attributes of any transport module must be provided as part of its design documentation.

Versioning — The underlying transport mechanism should make some arrangements for versioning of its marshalling format, and describe this versioning mechanism in accompanying documentation. The transport layer implemented by the fault manager also provides complete versioning of the FMA Protocol, and will check and validate the version of the protocol as part of establishing the virtual transport connection.

4.1.3 Protocol and Event Subscriptions

Once a transport module has established a virtual connection to a remote peer, the fault manager will automatically transmit a series of control events across the transport representing the active event subscriptions. On the receiving side, the transport peer will post these events, which will cause the peer's fault manager to proxy those subscriptions in the peer fault manager. Therefore, if a local module subscribes to `ereport.foo.bar`, the fault manager will arrange for the transport module to request that its peer subscribe to `ereport.foo.bar`, and any events of this class seen by the peer will be sent across the transport, dispatched, and then received by the local module as usual. While the fault manager is running, if modules are loaded or unloaded or modify their active subscriptions, these changes will be reflected through subscription control events passed to all active transport modules. The complete event protocol is shown in Figure 4–2.

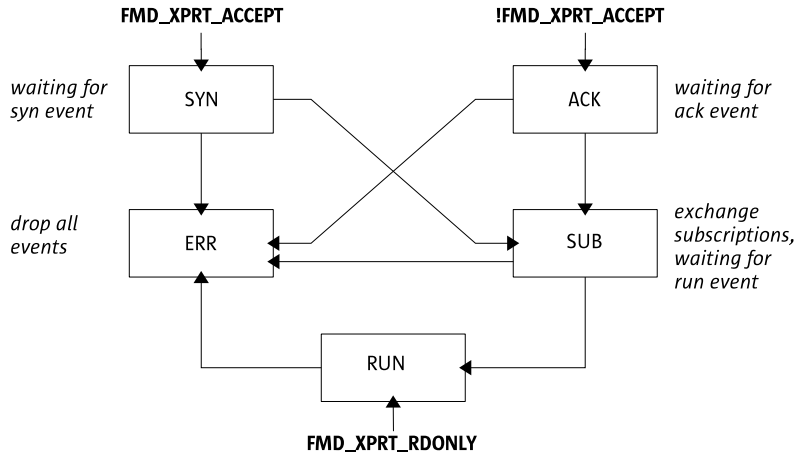


FIGURE 4–2 Fault Manager Transport Protocol

The initial protocol state is determined by whether the transport is read-only or read-write, and whether the FMD_XPRT_ACCEPT flag is specified as part of the call to `fmd_xprt_open()`. After the initial connection is established and version exchange occurs, the transports exchange subscriptions in the SUB state, and then send a final RUN control event to complete the transport handshake.

Once the RUN state is reached, the fault manager will send `list.suspect` events for any cases in the Solved state, and `fault.*` events for any cases in the Closed state where resource cache entries indicate the corresponding ASRU is in the Degraded state. Following this initial transmission of case-related events, subsequent events will be transmitted as they are dispatched locally. In all cases, the remote fault manager will only receive events for which it has active subscriptions.

4.1.4 Event Time-To-Live

The fault manager's common transport layer implements a simple time-to-live (TTL) mechanism for all events that prevents circular transport configurations from routing the same event back and forth indefinitely. Every event that is active in the fault manager is assigned a continuously decreasing TTL unsigned integer value. When a new event is created, it is assigned a TTL value derived from the `xprt.ttl` property (default is one). The TTL value is examined prior to executing `fmdo_send`; if it is zero, `fmdo_send` will not be called for the specified event. The TTL value is examined again when an event is received by a transport module and `fmd_xprt_post()` is called. If TTL is already zero, the fault manager assumes the event has timed out and automatically drops it. Otherwise the transport layer decrements the TTL by one prior to publishing the event to the global dispatch queue.

4.1.5 Case Proxying

The fault manager's common transport layer also provides a mechanism for case UUIDs to be proxied by a remote peer. When a `list.suspect` event passes from one transport module to another, the receiving fault manager will add the corresponding case UUID to its local dictionary of cases, noting that the receiving transport is acting as a proxy for this case. If any other modules in the receiving fault manager apply `fmd_case_uuclose()` to this UUID, the fault manager will execute the `fmdo_close` entry point of the transport module, send a control event across the transport to the peer fault manager that owns the case, and then execute a `fmd_case_uuclose()` operation in the fault manager that owns the case.

4.1.6 Time Conversion

Fault manager events are associated with both adjustable time-of-day and non-adjustable high-resolution time, as described in Chapter 3, "Events." When an FMA Protocol event is transferred from one fault manager to another through a transport, the fault manager arranges to preserve the event time so that the event retains its original time signature on the receiving side. Specifically, the fault manager encodes the adjustable time-of-day inside of the event payload that is passed to `fmdo_send` and uses this adjustable time-of-day to recompute an appropriate non-adjustable high-resolution time for the event in the receiving fault manager. Therefore, an `ereport.*` event associated with time T in the error log will also reflect time T in the error log of a remote fault manager that receives the event. However, if the receiver wishes to draw conclusions as to the relative ordering of local and remote events, some higher-level mechanism for synchronizing the adjustable time-of-day clocks for the two fault regions (such as NTP) must be provided. Developers must consider this issue carefully as part of any fault management deployment.

4.1.7 Observability

The fault manager and its observability tools provide features for observing the state of active transports. The `fmstat`(1M) utility can be used to view transport statistics and authority information. To view transport statistics for all active transports, use the `-t` option:

```
# fmstat -t
 id state ev_send ev_recv ev_drop ev_lost wait  svc_t  %w  %b module
  1   RUN       0       0       0       0  0.0    0.0   0   0 sysevent-transport
  2   RUN      32      31       0       0  2.1    5.1   8   8 ip-transport
```

To view transport authority information for all active transports, use the `-T` option:

```
# fmstat -T
 id state module              authority
```

```
1   RUN sysevent-transport
2   RUN ip-transport        server-id=127.0.0.1:45241
```

Developers can also enable a debugging log for transports by setting the `client.xprtlog` property, as described in Chapter 8, "Daemon Configuration." If the debugging log is enabled, the fault manager will store a copy of each event received by the transport into the file `/var/fm/fmd/xprt/`*id*`.log` where *id* is the integer transport identifier shown by `fmstat`. Debugging support is also provided for transport data structures; see Chapter 13, "Debugging" for more information about debugging facilities provided for the fault manager.

4.2 Transport Entry Point

Transport modules that implement read-write event transports to a remote fault manager are responsible for implementing an additional module entry point, `fmdo_send`, with the following signature:

```
int module_send(fmd_hdl_t *handle, fmd_xprt_t *xp, fmd_event_t *event, nvlist_t *nvl)
```

The `fmdo_send` entry point is given references to the corresponding module *handle*, transport *xp*, and *event* to send. The event payload, represented as a name-value pair list, is given by *nvl*. The fault manager will free *nvl* after `fmdo_send` returns; the transport module must make a copy of the payload or serialize it as part of `fmdo_send` as necessary. The entry point should return an integer status corresponding to one of the following values:

`FMD_SEND_SUCCESS`	The event was sent successfully and can now be deallocated and removed from the transport event queue.
`FMD_SEND_FAILED`	The event could not be sent because the transport detected an unrecoverable error associated with the transmission of this event. The event is deallocated and removed from the transport event queue.
`FMD_SEND_RETRY`	The event could not be sent because the transport detected a transient error condition. The event remains in memory, is restored to the transport event queue, and `fmdo_send` is called again for the specified event.

As events are queued up for the specified transport, the fault manager will call `fmdo_send` as needed. Some of the events are control events enqueued by the fault manager to drive the transport protocol that links the two fault managers together. Unlike other module entry points, the `fmdo_send` entry point will be called in a multi-threaded fashion. That is, multiple threads may be active in `fmdo_send` simultaneously when multiple transports are opened by a given module and are ready to send. Therefore the implementation of `fmdo_send` must use mutual exclusion when accessing shared data structures that are associated with the module.

The fault manager guarantees that only one thread will be active in `fmdo_send` *for a given transport* at a time, so mutual exclusion is not require when accessing per-transport data structures, such as those associated with `fmd_xprt_setspecific()`. However, the fault manager does not guarantee that the same thread will be used for each invocation, so transport modules should not associate transport-specific data with thread identifiers or thread-specific data.

4.3 Transport Interfaces

This section describes the programming APIs used to create and manipulate event transports. Transport modules may also make use of all the usual APIs described in Chapter 2, "Module API." None of the state manipulated by the transport APIs is checkpointed if the transport module produces a checkpoint file. Transport modules are responsible for establishing and re-establishing any connections upon module load.

4.3.1 fmd_xprt_open

```
fmd_xprt_t *fmd_xprt_open(fmd_hdl_t *handle, uint_t flags,
    nvlist_t *authority, void *data)
```

Open a new event transport and return an opaque transport handle corresponding to the new transport. The *flags* parameter should include the bitwise OR of one or more of the following values:

`FMD_XPRT_RDONLY`
: Indicate that the new transport is read-only and is only used to receive events. Read-only event transports may call `fmd_xprt_post()` to post received events, but `fmdo_send` will never be called for these transports. Read-only transports are used to represent an event source that is not implemented by another fault manager.

`FMD_XPRT_RDWR`
: Indicate that the new transport is read-write and can be used to send and receive events from a remote peer fault manager.

`FMD_XPRT_ACCEPT`
: Indicate that this transport is being used to accept a virtual connection from a remote peer fault manager that has an initial event pending. The fault manager expects that the transport module will call `fmd_xprt_post()` to initiate the transport session. If the `FMD_XPRT_ACCEPT` flag is not specified, the fault manager assumes that the transport is being used to initiate a virtual connection to a remote peer and it will enqueue an initial control event for `fmdo_send`.

`FMD_XPRT_SUSPENDED`
Create the transport in a suspended state as if by `fmd_xprt_suspend()`. No calls to `fmdo_send` will be made for this transport until `fmd_xprt_resume()` is called.

The module must specify exactly one of the `FMD_XPRT_RDONLY` and `FMD_XPRT_RDWR` flags; otherwise a module abort is triggered. The transport peer's FMA Protocol *authority* is specified as a name-value pair list. If the *authority* is NULL, the transport will use the authority information available from the transport peer (that is, the authority used by the remote peer to describe its own fault region). Finally, the specified client *data* value is stored in the transport handle and can be retrieved with `fmd_xprt_getspecific()`.

If `fmd_xprt_open()` is called from the transport module's `_fmd_init()` routine, no invocations of `fmdo_send` for the specified transport will occur until `_fmd_init()` completes successfully.

4.3.2 fmd_xprt_close

`void fmd_xprt_close(fmd_hdl_t *`*handle*`, fmd_xprt_t *`*xp*`)`

Close the specified event transport *xp*. All resources associated with the transport are deallocated, and any queued events that have not been processed by `fmdo_send` are discarded. The transport module is responsible for deallocating any memory associated with `fmd_xprt_setspecific()` if that is appropriate. If a transport module thread is executing `fmdo_send`, the `fmd_xprt_close()` function will block until the active call to `fmdo_send` has completed before proceeding. If a transport module's `_fmd_fini()` routine exits and one or more transports are still open, the fault manager will apply `fmd_xprt_close()` to each remaining transport.

Note – The fault manager will not permit a module to call `fmd_xprt_close()` on a transport handle from inside of the `fmdo_send` entry point when processing an event for the same transport. If such a call occurs, this condition will trigger a module abort.

4.3.3 fmd_xprt_post

`void fmd_xprt_post(fmd_hdl_t *`*handle*`, fmd_xprt_t *`*xp*`, nvlist_t *`*nvl*`, hrtime_t` *hrt*`)`

Post a newly received event for transport *xp* whose payload is specified by the name-value pair list *nvl* to the fault manager dispatch queue. The fault manager assumes responsibility for deallocating *nvl* when the event has completed the dispatch operation. If the event is a control event associated with the transport, the fault manager performs the corresponding control operation on the transport. If the event is invalid, the fault manager will automatically discard the event and update the transport statistics appropriately. If the event is an FMA Protocol `ereport.*` event, it will be written into the fault manager error log before it is dispatched.

If the transport has some mechanism to determine a high-resolution time at which the event was originally queued for the transport, this time can be specified as the *hrt* parameter; otherwise zero should be specified. This high-resolution time must correspond to the fault manager's native `gethrtime()` non-adjustable time source, and corresponds to time D in Figure 3–1. The *hrt* parameter is intended for use with read-only event transports: the fault managers participating in a read-write transport session will encode and decode the underlying event time automatically.

Note – The fault manager will not permit a module to call `fmd_xprt_post()` on a transport handle from inside of the module's `_fmd_init` routine. If such a call occurs, this condition will trigger a module abort.

4.3.4 fmd_xprt_error

```
int fmd_xprt_error(fmd_hdl_t *handle, fmd_xprt_t *xp)
```

Return a boolean value indicating whether the specified transport *xp* has detected an unrecoverable protocol error and is no longer making progress. If the transport has detected a protocol error, subsequent calls to `fmd_xprt_post()` will discard the specified event until the transport is closed.

4.3.5 fmd_xprt_suspend

```
void fmd_xprt_suspend(fmd_hdl_t *handle, fmd_xprt_t *xp)
```

Suspend the specified transport *xp*. Once a transport is suspended, no events will be dequeued (that is, no calls to `fmdo_send` will occur) until the transport is resumed using a call to `fmd_xprt_resume()`. Calls to other module entry points, including calls to `fmdo_send` associated with other active transports, will still occur. The `fmd_xprt_suspend()` call returns immediately without blocking, but the transport is not actually suspended until the end of any pending execution of the `fmdo_send` entry point. Therefore, if `fmd_xprt_suspend()` is called from `fmdo_send` and the entry point returns `FMD_SEND_RETRY`, the current event will be pushed back into the transport's event queue and the transport will be suspended with that event still enqueued. The transport must then be resumed by another thread or other module entry point.

Similar to the main module event queue, each transport event queue has a fixed queue limit determined by a configurable property (see `client.xprtqlimit` in Chapter 8, "Daemon Configuration"). When suspended, the transport's queue limit policy is still in effect: if events are queued for the transport and the queue limit is reached, additional events will be discarded until such time as the module is resumed and queued events are processed to reduce the queue length below the tunable limit.

The `fmd_xprt_suspend()` call only affects the transport send routine, controlled by the fault manager. The transport receive code is entirely under the control of the transport module, in that it determines when to call `fmd_xprt_post()`. The module is free to call `fmd_xprt_post()` on a suspended transport to pass along any newly received events, but the transport may not be able to complete a protocol handshake and make forward progress until it is resumed.

4.3.6 fmd_xprt_resume

`void fmd_xprt_resume(fmd_hdl_t *`*handle*`, fmd_xprt_t *`*xp*`)`

Resume the specified transport *xp*. Once a transport is resumed, calls to the `fmdo_send` entry point will be made as needed for events that should be sent to the transport peer. If the transport is not suspended, this function has no effect.

4.3.7 fmd_xprt_translate

`nvlist_t *fmd_xprt_translate(fmd_hdl_t *`*handle*`, fmd_xprt_t *`*xp*`, nvlist_t *`*nvl*`)`

Translate the specified FMA Protocol event name-value pair list *nvl* into a form suitable for the FMA authority associated with the peer of the specified transport *xp* and return a new event name-value pair list. The `fmd_xprt_translate()` function operates by recursively examining all members of *nvl*, locating those that represent FMRIs, and applying the corresponding FMRI scheme `fmd_fmri_translate()` function to each one using the authority information specified as a parameter to `fmd_xprt_open()`. If any of the scheme translation calls fail, `fmd_xprt_translate()` returns NULL to indicate an error. Otherwise the new name-value pair list is returned. The caller is responsible for deallocating this new name-value pair list using `nvlist_free()` when appropriate. See the `nvlist_alloc`(3NVPAIR) man page for more information about `nvlist_free()`.

4.3.8 fmd_xprt_getspecific

`void *fmd_xprt_getspecific(fmd_hdl_t *`*handle*`, fmd_xprt_t *`*xp*`)`

Return the client data pointer associated with the specified transport *xp*. The value returned is initially specified by the final parameter to `fmd_xprt_open()`, and can be subsequently modified by `fmd_xprt_setspecific()`.

4.3.9 fmd_xprt_setspecific

`void fmd_xprt_setspecific(fmd_hdl_t *`*handle*`, fmd_xprt_t *`*xp*`, void *`*data*`)`

Set the client data pointer associated with the transport *xp* to the specified *data*, replacing any previous value.

4.4 Event Interfaces

This section describes programming APIs that can be applied to opaque fault manager event handles, such as those provided as arguments to the `fmdo_send` and `fmdo_recv` entry points.

4.4.1 fmd_event_local

`int fmd_event_local(fmd_hdl_t *`*handle*`, fmd_event_t *`*event*`)`

Return a boolean indicating if the specified *event* was received from a local event transport (non-zero) or a transport associated with a remote fault manager (zero).

4.4.2 fmd_event_ena_create

`uint64_t fmd_event_ena_create(fmd_hdl_t *`*handle*`)`

Create a unique ENA for use when generating ereports. See "3.2 Event Times" on page 64 for more information. This ENA uses the current timestamp, and this ENA can be used by transport modules to generate synthetic ereports from a non-FMA source.

4.5 SysEvent Transport

The `sysevent-transport` module is provided as a built-in module with the Solaris reference implementation of the fault manager. The SysEvent facility provides a kernel-to-userland event queue with two-phase commit semantics using Solaris doors as the underlying transport mechanism. The `sysevent-transport` is a read-only transport that receives events published to a known SysEvent channel by the Solaris kernel and other software components.

4.5.1 Design Overview

Atomicity	The `sysevent-transport` module leverages the two-phase commit semantics of SysEvent in order to guarantee event atomicity.
Ordering	The `sysevent-transport` module provides event ordering because the SysEvent mechanism is an in-memory FIFO queue on the same system as the fault manager.
Connectivity	The `sysevent-transport` module is always connected to the local system.
Reliability	The `sysevent-transport` module leverages the local system's memory data integrity mechanism, as well as the data validation that is built into the `libnvpair` encoding for name-value pair lists. Invalid name-value pairs will be cleanly detected and result in a dropped event.

Security	Local process credentials are used to ensure that a valid fault manager is using `sysevent-transport`. SysEvent currently requires that the super-user credential open its queues.
Versioning	The SysEvent APIs themselves are versioned and the name-value pair encoding used by `libnvpair` assigns a Stable integer identifier to each encoding mechanism.

4.5.2 Properties

The following table describes the properties for the `sysevent-transport` module:

TABLE 4–1 `sysevent-transport` properties

Name	Type	Default	Description
`channel`	`string`	`FM_ERROR_CHAN`	Name of the SysEvent channel to use for inbound event telemetry.
`class`	`string`	`EC_ALL`	Subscriber class filter to use with SysEvent channel.
`device`	`string`	`-`	If set, attempt to replay saved events from the specified dump device. If not set (default), query the system configuration to determine the current dump device.
`sid`	`string`	`fmd`	SysEvent transport channel subscriber identifier string.

4.6 IP Transport

The `ip-transport` module is provided as a reference implementation of a read-write transport module and development and debugging tool for `fmsim` that uses TCP/IP sockets as the underlying transport mechanism. This module is provided as part of the Solaris reference implementation of the fault manager, but it is not used by the Solaris product at this time. The transport operates by marshaling name-value pair lists using an XDR encoding provided by `libnvpair`. The `ip-transport` can function in one of two modes, either as a server or client, depending upon the configuration of its properties. In client mode, the transport module attempts to make a connection to the specified server module at a known TCP/IP address and port number. In server mode, the transport module binds a socket to `INADDR_ANY` at the specified port and waits for incoming connections.

4.6.1 Design Overview

Atomicity — The `ip-transport` module leverages the TCP protocol's sliding window mechanism and and sequence numbers in order to guarantee event atomicity.

Ordering — The `ip-transport` module leverages the TCP protocol's ordering guarantees in order to provide fault manager event ordering.

Connectivity — The `ip-transport` module leverages the standard TCP/IP socket mechanisms for detecting broken connections at the time of each send or receive operation. If a connection is broken, the module currently makes no provisions to reconnect; this issue may need to be addressed by a system design which intends to deploy `ip-transport` in production.

Reliability — The `ip-transport` module leverages the standard TCP/IP checksums for data reliability, as well as the data validation that is built into the `libnvpair` XDR encoding for name-value pair lists. Invalid XDR will be cleanly detected and result in a dropped event.

Security — At this time, the `ip-transport` module makes no provisions for authentication or security: these issues must be addressed by any design which intends to deploy `ip-transport` in a production system.

Versioning — Each event transported by `ip-transport` is prefixed with a header that is checked by the receiver to implement a simple versioning scheme.

4.6.2 Properties

The following table describes the properties for the `ip-transport` module:

TABLE 4–2 `ip-transport` properties

Name	Type	Default	Description
`ip_authority`	`string`	`-`	If set, parse the specified string into a name-value pair list representing an authority by separating it based upon comma delimiters, and install this authority as the representation of the peer as part of `fmd_xprt_open()`. If this option is not set (default), an authority is constructed using the remote peer's Internet IP address and remote port number.

TABLE 4–2 `ip-transport` properties *(Continued)*

Name	Type	Default	Description
`ip_bufsize`	`size`	`4k`	Specify the maximum buffer size to use for marshaling and unmarshaling FMA Protocol events. The transport will reject any events whose size exceeds this tunable.
`ip_burp`	`time`	`0`	If the `ip_mtbf` property is enabled and `ip_burp` is non-zero, the transport will induce extra delay into the simulated failure by suspending the transport for `ip_burp` nanoseconds and then resuming it again.
`ip_enable`	`boolean`	`false`	If set to true, enable the transport module to load. By default, the transport module is disabled because it is not yet used in production Solaris.
`ip_mtbf`	`int32`	`0`	If set to a non-zero value, induce pseudo-random simulated failures into the transport by returning `FMD_SEND_RETRY` from `fmdo_send` when the result of `gethrtime() % ip_mtbf` is zero.
`ip_port`	`string`	`664`	Specifies the remote port number or Internet service to connect to (client mode) or bind to (server mode).
`ip_qlen`	`int32`	`32`	Specifies the maximum length of the TCP/IP pending connection queue when the transport module is operating in server mode.
`ip_retry`	`uint32`	`50`	Specifies the number of times the transport module should retry its attempt to establish the appropriate socket connection (client mode) or binding (server mode) before giving up and calling `fmd_hdl_abort()`. The module will delay for `ip_sleep` nanoseconds between each attempt.
`ip_server`	`string`	`-`	Specify the Internet IPv4 or IPv6 hostname or address to connect to if the module is operating in client mode. If the property is not set (default), the module operates in server mode instead.

TABLE 4–2 `ip-transport` properties *(Continued)*

Name	Type	Default	Description
`ip_sleep`	`time`	`10s`	Specify the retry interval for the `ip_retry` property.
`ip_translate`	`boolean`	`false`	If set to true, this property indicates that the transport module should apply `fmd_xprt_translate()` to each event that is passed to `fmdo_send`.

◆ ◆ ◆ CHAPTER 5

Log Files

The fault manager maintains two persistent log files of FMA events, the *error log* and the *fault log*. The error log is used to persistently record inbound error event telemetry information. The fault log is used to persistently record diagnosis results (`list.suspect` events containing lists of faults, defects, and upsets). This chapter discusses the structure of the log files and algorithms used by the fault manager to maintain them. The `fmdump`(1M) utility is provided to permit examination of all fault manager logs.

5.1 Log Structure

The fault manager log files use the Extended Accounting format associated with `libexacct.so.1`. Extended Accounting log files consist of a header following by a series of structured groups, each containing one or more groups or items. Each group and item is labeled by a 32–bit catalog tag; the system catalog is stored in the header file `/usr/include/sys/exacct_catalog.h`. The following BNF grammar describes the structure of the fault manager logs, where each left-hand element indicates a record group:

```
log := hdr toc event*
hdr := EXD_FMA_LABEL EXD_FMA_VERSION
       EXD_FMA_OSREL EXD_FMA_OSVER EXD_FMA_PLAT EXD_FMA_UUID
toc := EXD_FMA_OFFSET
event := EXD_FMA_TODSEC EXD_FMA_TODNSEC EXD_FMA_NVLIST evref*
evref := EXD_FMA_UUID EXD_FMA_OFFSET
```

The log header elements can be displayed using the `fmdump -H` option, as shown in the following example. At present, this option is not publicly documented because it exposes part of the log file implementation.

```
$ fmdump -H /var/fm/fmd/fltlog
EXD_CREATOR = fmd
EXD_HOSTNAME = poptart
EXD_FMA_LABEL = fault
```

```
EXD_FMA_VERSION = 1.1
EXD_FMA_OSREL = 5.11
EXD_FMA_OSVER = snv_30
EXD_FMA_PLAT = i86pc
EXD_FMA_UUID = d2e6d6ad-2efd-6e37-cf76-c837173aa658
```

The following table describes the set of extended accounting tags used by the fault manager and their meanings:

TABLE 5–1 Extended Accounting Catalog Tags

Tag Name	Data Type	Description
EXD_GROUP_RFMA	EXT_GROUP	Group tag for a fault manager record group that needs to be replayed when the fault manager restarts after it fails, is restarted by an administrator, or after system reboot.
EXD_GROUP_FMA	EXT_GROUP	Group tag for a fault manager record group that does not need to be replayed.
EXD_FMA_LABEL	EXT_STRING	Generic file label, indicating the file type. This is one of the strings `error`, `fault`, or `asru`, and is used by the fault manager to verify the log type and by `fmdump`(1M) to adopt the proper format.
EXD_FMA_VERSION	EXT_STRING	Version string indicating the version of `fmd` that created this log. The fault manager can use this version to infer the grammar of the rest of the file, permitting newer daemons to interpret older log files.
EXD_FMA_OSREL	EXT_STRING	Operating environment release that this log was created on; the reference implementation stores the value of `uname -r` here.
EXD_FMA_OSVER	EXT_STRING	Operating environment version that this log was created on; the reference implementation stores the value of `uname -v` here.
EXD_FMA_PLAT	EXT_STRING	Operating platform that this log was created on; the reference implementation stores the value of `uname -i` here.
EXD_FMA_OFFSET	EXT_UINT64	64–bit offset of a record group in another log file. The use of links to other log files is discussed below.
EXD_FMA_TODSEC	EXT_UINT64	64–bit seconds since 00:00:00 UTC, January 1, 1970.
EXD_FMA_TODNSEC	EXT_UINT64	64–bit nanoseconds since 00:00:00 UTC, January 1, 1970.

TABLE 5–1 Extended Accounting Catalog Tags *(Continued)*

Tag Name	Data Type	Description
EXD_FMA_NVLIST	EXT_RAW	An FMA protocol event name-value pair list. The raw bytes represent the XDR-encoded serialized form of a `libnvpair.so.1` name-value pair list data structure.
EXD_FMA_MAJOR	EXT_UINT32	The major number from the `st_dev` status information of another log file. The use of this tag is deprecated.
EXD_FMA_MINOR	EXT_UINT32	The minor number from the `st_dev` status information of another log file. The use of this tag is deprecated.
EXD_FMA_INODE	EXT_UINT64	The inode number from the `st_ino` status information of another log file. The use of this tag is deprecated.
EXD_FMA_UUID	EXT_STRING	A UUID referring to the unique identifier assigned to another FMA log file. The use of links to other log files is discussed below.

5.1.1 Event Times

Event times are always stored in fault manager logs as 64–bit seconds and nanoseconds since 00:00:00 UTC, January 1, 1970. This design ensures that although we currently use 32–bit fault management daemons and tools, we can simply recompile them as 64–bit later without the need to revise the log file format and without causing Year 2038 time representation issues. When events are replayed from a fault manager log, the event time is converted from its UTC format back into a non-adjustable high-resolution time for use in SERD engines and timeouts.

5.1.2 Event References

Events can contain references to other events in other log files (the `evref` element of the grammar shown above). This facility is used in the fault log to indicate which telemetry events led to a particular diagnosis, assisting engineers in debugging the diagnosis algorithms themselves. Since error logs can be rotated frequently, the references cannot refer to pathnames, and can be broken when a log file is eventually deleted. Once an event moves through the event transport from an event producer to a consuming fault manager and is written to a log, its unique identity in the universe is in effect the tuple (*file, offset*) where *file* is some unique identifier for the log file and *offset* is the 64–bit offset of that event in that particular log file. The fault manager reference implementation uses a UUID written to the header of each new log file to represent the *file* identifier, as this provides an easy way to persist links as log files are rotated periodically. These links can obviously become broken or outdated, but these cases are easily detected by the log tools and again, the links are only provided to assist debugging and are not required for correct operation of the fault manager.

The byte offsets of individual log file elements can be displayed using the `fmdump -O` option, which accepts a hexadecimal byte offset as an argument. If offset zero is specified, all log file elements are displayed. If an offset greater than zero is specified, the element at the specified offset and all subsequent elements are displayed. When the `-O` option is present, an additional column is added to the standard `fmdump` output indicating the byte offset of each record. At present, the `-O` option is not publicly documented because it exposes part of the log file implementation.

```
$ fmdump -O 0
OFFSET TIME                 CLASS
   11c Mar 30 20:45:26.1376 ereport.io.ddi.defect.fm-capability
   244 Mar 30 20:29:21.6535 ereport.io.ddi.defect.fm-capability
   3f8 Mar 30 20:29:22.1505 ereport.io.ddi.defect.fm-capability
   5ac Mar 30 20:29:22.1547 ereport.io.ddi.defect.fm-capability
   760 Mar 30 20:29:22.1598 ereport.io.ddi.defect.fm-capability
   914 Mar 30 20:29:22.1640 ereport.io.ddi.defect.fm-capability
   ac8 Mar 30 20:29:22.1693 ereport.io.ddi.defect.fm-capability
   c7c Mar 30 20:29:22.1734 ereport.io.ddi.defect.fm-capability
   e30 Mar 30 20:29:22.1789 ereport.io.ddi.defect.fm-capability
...
```

5.2 Error Log

The error log is the series of events that have been received over the inbound event telemetry transport. Events are first written to this log to ensure their persistence before they are forwarded on to subscribing modules. The content of these events is described further in the *Solaris FMA Event Protocol* document. Events in this file must be replayed if they are still in the Received state (see Chapter 3, "Events"), so they are initially written to the fault manager log file using the `EXD_FMA_RGROUP` catalog tag. Once the event has transitioned out of the Received state, the fault manager simply overwrites the catalog tag with the `EXD_FMA_GROUP` tag to indicate the event no longer needs to be replayed. The fault manager also uses the `toc` record group to store the offset of the earliest entry in the log file that may need to be replayed. The fault manager updates this offset periodically and on startup so that subsequent executions do not need to linearly scan the entire log file. Note that this update does not need to be atomic with respect to other changes as it is simply a performance optimization and does not affect correctness of replay. The error log is opened using the `O_SYNC` option to ensure that writes reach secondary storage before the fault manager proceeds.

5.3 Fault Log

The fault log is the series of `list.suspect` events corresponding to diagnoses made by modules running in the fault manager. This log file uses the `EXD_FMA_GROUP` tag exclusively and does not make use of the `toc` record group, as none of its events are replayed (fault replay is handled by the resource cache). The content of these events is described further in the *Solaris FMA Event Protocol* document. The events in the fault log do contain `evref` entries for all error reports that were associated with the case that produced a given diagnosis, so that they can be retrieved using `fmdump`(1M). The links are maintained as log files are renamed or rotated, but can be broken when an error log is deleted or moved to a different filesystem; the `fmdump` tool will display information as best it can given the links that are still valid. The fault log is opened using the `O_SYNC` option to ensure that writes to it reliably reach secondary storage before the fault manager proceeds.

◆ ◆ ◆ CHAPTER 6

Resource Cache

Aside from managing state associated with modules, the fault manager's other major responsibility is to maintain a resource cache which caches the state of resources that are the subject of fault management activities. The cache *only maintains information relevant to fault management*; information specific to the resource class is expected to be stored elsewhere. Specifically, the cache maintains only whether or not the fault manager believes the resource to be faulty, whether or not the fault manager believes the resource to be unusable (that is, disabled so as not to cause further error reports), and references to the diagnosis result that indicated it was faulty. The purpose of the resource cache is to maintain this history across restarts of the fault manager and the system so that faulty elements can be disabled again if they have not been repaired or removed from the system, and to facilitate a single way to answer the request "show me everything that is broken on my system."

6.1 Resource Model

The fault manager resource model is a very simple one, to ensure that it is applicable to basically any resource: every resource is named by an FMRI (for which new resource *schemes* can be defined by Sun), and every resource is assigned two boolean values indicating whether the resource is believed to be *faulty* and whether the resource is currently *unusable*, indicating that it has been disabled. The fault manager caches this information for resources that are Automated System Recovery Units (ASRUs), which are resources that can be automatically disabled and are named in fault events generated using the FMA Event Protocol. Note that ASRUs can refer to either hardware entities like DIMMs and CPUs, or to software abstractions such as services managed by the Solaris Service Management facility, device driver abstractions, and other entities. Together, the two boolean state values produce a matrix of four possible states for a fault managed resource:

Faulty?	Unusable?	State	Description
false	false	ok	The resource is enabled and is not believed to be faulty.
false	true	unknown	The resource is not enabled for some reason other than being faulty (for example, it has been disabled or blacklisted manually by an administrator).
true	false	degraded	The resource is faulty but is still enabled. This state typically indicates that either the resource has not yet been disabled by an agent, or it is not possible to disable.
true	true	faulted	The resource is faulty and disabled, indicating that either the resource owner or a fault management agent has disabled it.

Further details of the resource states, including the complete state machine, are described in the *FMA Event Protocol* document. The *faulty* property is set by a conviction policy that is applied to each suspect list during a call to `fmd_case_solve()` and can be cleared by the `fmadm`(1M) `repair` subcommand. The *unusable* property can be set for faulty resources by `fmd_case_close()` and `fmd_case_uuclose()`, and can be cleared by a *scheme plug-in* that queries the resource owner for its latest status. Resource properties unrelated to fault management (for example, the notion of a processor no-interrupt state or a device driver quiescent state) are *not* maintained by the fault manager and are left to the resource owner.

When the fault manager initializes, the set of persistent *resource logs*, described next, is examined to populate the resource cache. If an entry is no longer configured in the system (according to its scheme plug-in) and its age exceeds a configurable threshold, the resource log is permanently deleted. If the resource is still present and it is considered *faulty*, then the fault event which made the resource transition to either Degraded or Faulted is replayed to any subscribing modules, permitting appropriate agents to again disable the resource.

6.2 Resource Logs

The resource cache is maintained as a set of persistent resource logs, which are kept in the directory `/var/fm/fmd/rsrc`, and are considered Project Private. To aid field personnel in the event of a problem, the `fmadm`(1M) command provides facilities for manually flushing resource cache entries. Each resource log is maintained using the same Extended Accounting log format described earlier in Chapter 5, "Log Files," except that the events logged to these files are Project Private events of class `resource.fm.fmd.*`, where each event represents a change in one of the two state values and contains the UUID of the case associated with this change, and a copy of the relevant fault event for replay. By maintaining a running log of such events, Sun developers and service personnel can also apply `fmdump`(1M) to the log files to analyze the history of a particular resource with respect to fault management activities. Each resource log is named

using another UUID, referred to as the *resource identifier*, which can also be used as a software serial number for this resource if no underlying resource serial number is available.

6.3 Scheme Plug-in Interfaces

The fault manager provides a Project Private programming interface for implementing *scheme plug-ins*, which are shared libraries that implement routines to convert between FMRI name-value pair list representation and a persistent string representation, and a routine to determine the current value of the resource's *unusable* property. These libraries are named after the resource scheme and are installed in the `/usr/lib/fm/fmd/schemes` directory and are loaded on demand. If the fault manager is running low on memory, it may attempt to unload a scheme plug-in and then load it again later when it is needed.

The programming interface for the scheme libraries is described in the file `fmd/common/fmd_fmri.h`, and is described in this section. Scheme plug-ins implement one or more of the specified function entry points. If a particular function is not defined by the plug-in, an appropriate default behavior is provided. If a scheme operation fails, the fault manager routine that invoked the scheme operation will fail in an appropriate manner, such as returning the error to its caller. Scheme plug-ins must be implemented using reentrant interfaces as they can be called from any thread inside the fault manager. The fault manager associates a single lock with each scheme plug-in and holds the lock for the duration of each plug-in call, so only one plug-in entry point can be called at a time. Therefore, plug-ins do not need to implement their own mutual exclusion locking algorithms.

6.3.1 fmd_fmri_init

```
int fmd_fmri_init(void)
```

The `fmd_fmri_init()` function, if present, will be called once when the scheme is loaded. If the function does not exist, scheme initialization is considered to be successful. If the function does exist and returns zero, scheme initialization is considered to be successful. If the function does exist and returns non-zero, scheme initialization is considered to have failed and no further calls will be made to this scheme for the duration of the fault manager's lifetime. Any subsequent calls to the scheme module will automatically return failure.

6.3.2 fmd_fmri_fini

```
void fmd_fmri_fini(void)
```

The `fmd_fmri_fini()` function, if present, will be called once when the scheme is unloaded, assuming that it initialized successfully. The scheme plug-in can use this entry point to free any memory allocated by the `fmd_fmri_init()` function.

6.3.3 fmd_fmri_nvl2str

```
ssize_t fmd_fmri_nvl2str(nvlist_t *nvl, char *buf, size_t len)
```

Convert the specified name-value pair list *nvl*, which represents an FMRI of the corresponding scheme, into its string representation and place the result in *buf*. If the resulting string would exceed the buffer length *len*, the string should be truncated at *len* – 1 bytes and a NULL character should be inserted at the location *buf*[*len* - 1]. The `fmd_fmri_nvl2str()` function is required to be defined by each scheme plug-in. The `fmd_fmri_nvl2str()` function should return —1 to indicate an error, or return the number of bytes required to represent the full string FMRI, excluding the trailing NULL byte, to indicate successful conversion of the FMRI. The scheme plug-in is required to ensure that the FMRI string is free of reserved URI characters. If any reserved characters might be present in the FMRI, the `fmd_fmri_strescape()` utility function can be used to replace them with appropriate escape sequences.

6.3.4 fmd_fmri_expand

```
int fmd_fmri_expand(nvlist_t *nvl)
```

Expand the specified name-value pair list *nvl*, which represents an FMRI of the corresponding scheme, to include any FMRI elements that are part of the scheme but not present in the specified FMRI. Typically the `fmd_fmri_expand()` mechanism is used by diagnosis engines and schemes where an initial error report event includes an FMRI but does not include certain information about the resource that cannot be captured by the error handler, such as serial number data. This function should return 0 to indicate success or —1 to indicate an error. If the `fmd_fmri_expand()` function is not defined, a default version that returns 0 without modifying the FMRI is provided for the scheme.

6.3.5 fmd_fmri_present

```
int fmd_fmri_present(nvlist_t *nvl)
```

Return a status value indicating whether or not the FMRI corresponding to the specified name-value pair list *nvl* is present in the system. If serial numbers are available for the scheme, they should be employed to verify the identity of FMRI in addition to comparing resource location information. If the resource is present, the scheme should return 1 from this function. If the resource is not present, the scheme should return 0 from this function. If the FMRI is invalid or an error occurs, the scheme should return —1 from this function. The `fmd_fmri_present()` function is required to be defined by each scheme plug-in.

6.3.6 fmd_fmri_unusable

```
int fmd_fmri_unusable(nvlist_t *nvl)
```

Return a status value indicating whether or not the FMRI corresponding to the specified name-value pair list *nvl* is usable on the system or not (that is, whether it is disabled). The means for disabling a given resource is entirely specific to each resource type and scheme; examples include page retirement and CPU offlining. If the resource is unusable (currently disabled), the scheme should return 1 from this function. If the resource is usable, the scheme should return 0 from this function. If the FMRI is invalid or an error occurs, the scheme should return —1 from this function. The `fmd_fmri_unusable()` function is required to be defined by each scheme plug-in.

6.3.7 fmd_fmri_contains

`int fmd_fmri_contains(nvlist_t *`*nvl1*`, nvlist_t *`*nvl2*`)`

Return a status value indicating whether or not the FMRI corresponding to the specified name-value pair list *nvl1* contains the FMRI corresponding to the name-value pair list *nvl2*. The notion of containment for a given resource is entirely specific to each resource type and scheme. If *nvl1* contains *nvl2*, the scheme should return 1 from this function. If not, the scheme should return 0 from this function. If either FMRI is invalid or an error occurs, the scheme should return —1 from this function. If the `fmd_fmri_contains()` function is not defined by a scheme, a default version that returns —1 and indicates the operation is not supported is provided by the fault manager.

6.3.8 fmd_fmri_translate

`nvlist_t *fmd_fmri_translate(nvlist_t *`*nvl*`, nvlist_t *`*authority*`)`

Translate the specified FMRI name-value pair list *nvl* into a form suitable for use in the fault region specified by the given FMRI *authority*. If the translation succeeds, a new name-value pair list should be allocated for the translated FMRI and returned; otherwise NULL should be returned to indicate an error. The specified *fmri* and *auth* should not be modified or deallocated by this function. If the `fmd_fmri_translate()` function is not defined by a scheme, a default version that returns a new copy of the specified *fmri* is provided by the fault manager.

6.4 Scheme Plug-in Utility Functions

The fault manager provides a set of Project Private utility functions that facilitate the implementation of scheme plug-ins. The utility functions for the scheme plug—ins are also described in the file `fmd/common/fmd_fmri.h`, and are described in this section.

6.4.1 fmd_fmri_alloc

`void *fmd_fmri_alloc(size_t` *nbytes*`)`

Allocate a block of memory of size *nbytes* and return a pointer to it. The scheme plug-in is required to free the memory either before returning from the current entry point or as part of its `fmd_fmri_fini()` function. The memory is guaranteed to be of an alignment suitable for the largest C data type. No guarantees are made about its initial contents. If *nbytes* is zero, a NULL pointer will be returned. Otherwise the fault manager will sleep until sufficient memory is available. Therefore, this function cannot fail and scheme plug-ins do not need to check its return value for allocation failures.

6.4.2 fmd_fmri_zalloc

`void *fmd_fmri_zalloc(size_t` *nbytes*`)`

Allocate a block of memory of size *nbytes* as if `fmd_fmri_alloc()`, fill the memory with zeroes, and return a pointer to it. As with `fmd_fmri_alloc()`, this function cannot fail.

6.4.3 fmd_fmri_free

`void fmd_fmri_free(void *`*ptr*`, size_t` *nbytes*`)`

Free the block of memory specified by *ptr* that was previously allocated by `fmd_fmri_alloc()` or `fmd_fmri_zalloc()`. The pointer and size (*nbytes*) must correspond exactly to a previously allocated block; partial frees are not permitted. Invalid or duplicate frees are considered programming errors and will cause the fault manager to core dump with appropriate information recorded for debugging the problem.

6.4.4 fmd_fmri_set_errno

`int fmd_fmri_set_errno(int` *error*`)`

Set the scheme error code that will be recorded for the current scheme entry point to the specified *error* value and return —1. The *error* should be an appropriate value from `<errno.h>` that will help developers in debugging the problem.

6.4.5 fmd_fmri_warn

`void fmd_fmri_warn(const char *`*format*`, ...)`

Create an event of class `ereport.sunos.fmd.scheme` and record it in the error log with an informational message to assist developers in debugging a problem. The message will be formatted according to the specified *format* string as if by a call to `snprintf()`.

6.4.6 fmd_fmri_strescape

```
char *fmd_fmri_strescape(const char *str)
```

Allocate a string corresponding to the specified string *str* where reserved URI characters have been replaced by appropriate escape sequences, and return a pointer to the new string. For example, the whitespace character is replaced with the URI escape sequence `%20`. The scheme is responsible for freeing this string using `fmd_fmri_strfree()` before returning from the current entry point or as part of `fmd_fmri_fini()`. If *str* is NULL, a NULL pointer is returned.

6.4.7 fmd_fmri_auth2str

```
char *fmd_fmri_auth2str(nvlist_t *auth)
```

Allocate and format an FMRI authority string for the FMRI authority given by the name-value pair *nvl*. Elements are delimited by commas, and any characters which are not permitted to be in a URI are escaped using the RFC2396 character escape notation. The resulting string can be deallocated using `fmd_fmri_strfree()`.

6.4.8 fmd_fmri_strdup

```
char *fmd_fmri_strdup(const char *str)
```

Allocate a string with the same content as the specified string *str*, and return a pointer to the new string. The scheme is responsible for freeing this string using `fmd_fmri_strfree()` before returning from the current entry point or as part of `fmd_fmri_fini()`. If *str* is NULL, a NULL pointer is returned.

6.4.9 fmd_fmri_strfree

```
void fmd_fmri_strfree(char *str)
```

Free the memory for the string *str* that was allocated by an earlier call to `fmd_fmri_strescape()` or `fmd_fmri_strdup()`. If *str* is NULL, the function call has no effect and is silently ignored.

6.4.10 fmd_fmri_get_rootdir

```
const char *fmd_fmri_get_rootdir(void)
```

Return a pointer to the current value of the fault manager's `rootdir` property, which indicates the root directory for pathname expansions. The default value of this property is "`/`".

6.4.11 fmd_fmri_get_platform

```
const char *fmd_fmri_get_platform(void)
```

Return a pointer to the current value of the fault manager's `platform` property, which indicates the platform name. The default value of this property is equivalent to the output of `uname -i`.

6.4.12 fmd_fmri_get_drgen

```
uint64_t fmd_fmri_get_drgen(void)
```

Return the current dynamic reconfiguration generation number. This number is initialized to zero when the fault manager starts up, and is incremented each time a dynamic reconfiguration occurs on the system. A scheme can store the generation number with cached resource information and then compare the current value to the stored values to determine that the cached information is no longer valid.

6.4.13 fmd_fmri_topology

```
topo_hdl_t *fmd_fmri_topology(int version)
```

Return a handle for accessing the topology library `libtopo.so`. The *version* indicates the library API version to be used, and should be specified as `TOPO_VERSION`. See Chapter 9, “Topology” for information about topology.

◆ ◆ ◆ CHAPTER 7

Checkpoints

The fault manager provides a facility for automatically checkpointing module state to ensure that module state survives a failure of the module, fault manager, or system that is currently executing the fault manager. This chapter provides a very brief overview of the checkpointing mechanism; further details are all Project Private and are discussed in the fault manager source code comments.

7.1 Checkpoint Design

The basic design of the module checkpoint mechanism is that checkpoints are taken as necessary following the execution of any module entry point. Every resource checkpointed by the fault manager (statistics, cases, buffers, and SERD engines) is assigned a *dirty bit* indicating that it needs to be checkpointed; this bit is set when the resource is created, set again when it is modified, and cleared by the checkpoint routines in the fault manager. If the dirty bit on any module resource is set, a top-level dirty bit is set on the module indicating that the module has one or more resources that require checkpointing. The checkpoints are performed on a module-by-module basis after the module receives an event and returns from its `fmdo_recv` entry point. This design ensures that a restarted module either sees all of the state associated with a case as restored, or it sees none of it (for example, if the checkpoint failed due to lack of disk space or was subsequently deleted or corrupted).

The checkpoints are stored inside of the system directory `/var/fm/fmd/ckpt/` under a subdirectory named after each module, which is created as part of the first checkpoint. Checkpoint files are created using names that will not be read by the fault manager upon restart, and once complete and synchronized on disk, they are atomically renamed into place for restart. If the fault manager encounters an error during a checkpoint (such as running out of disk space), it logs an appropriate error and simply retains the dirty bit setting on the affected resources, hoping that they will be able to be checkpointed in the future.

The checkpoint files contain a versioned, extensible header, and use an ELF-like structure consisting of a series of section headers followed by a data section corresponding to each section header. A Project Private section identifier and data format is defined for each resource,

including statistics, cases, buffers, and SERD engines. The fault manager carefully validates each section of the checkpoint file upon restart to ensure that a corrupt file does not cause the fault manager to crash; if any portion of the checkpoint file is corrupt, the entire checkpoint is discarded and an appropriate error is logged. In the event that checkpointed state has been corrupted to the point where it must be destroyed or at the behest of service personnel, the `fmadm`(1M) `reset` subcommand can be used to unload a module, delete all of its checkpoints, and restart the module.

◆ ◆ ◆ CHAPTER 8

Daemon Configuration

Earlier chapters have described the purpose of the fault manager and the set of services it provides to its clients. In this chapter we discuss topics of interest to fault manager developers and deployers, including global properties of the fault manager and a description of the Private configuration options available for these users.

8.1 Configuration Files

The fault manager searches for its own configuration files, which can contain any of the directives described earlier in Chapter 2, "Module API," in a set of preconfigured directories when it starts up. If a configuration file is found, it is parsed and its settings modify the values of any relevant configuration properties. In addition to the property types described in Chapter 2, "Module API," the daemon configuration files may also use properties of type `path`, which is expressed as a colon-delimited list of directories that may contain any of the following pathname expansion tokens:

TABLE 8–1 Path Expansion Tokens

Token	Description
`%i`	Platform name (`platform` property)
`%m`	Machine class (`machine` property)
`%p`	Processor name (`isaname` property)
`%r`	Root directory (`rootdir` property)
`%%`	Literal `%` character

The configuration files are named `fmd.conf` by default (although this can be changed by the `conf_file` property) and are searched for in the path defined by the `conf_path` property. The Solaris reference implementation does not ship with any settings in configuration files; these

can be created as needed by developers, service, and operations and manufacturing. Unlike module configuration files, the fault manager configuration files can contain deferred property settings for other modules by specifying a property name as a module name followed by a colon (:) and a property name. For example, the directive:

```
setprop syslog-msgs:gmt true
```

would set the `gmt` property to `true` whenever a module named `syslog-msgs` is loaded into the fault manager.

The configuration file properties are as follows:

TABLE 8–2 Fault Manager Configuration Properties

Name	Type	Default	Description
`agent.path`	`path`	See description following table.	Path to use to locate external modules.
`alloc_msecs`	`uint32`	`10`	Initial number of milliseconds to sleep before retrying a sleeping allocation that has failed. This value is multipled by 10 with each retry attempt.
`alloc_tries`	`uint32`	`3`	Maximum number of allocation retries before attempting garbage collection and then aborting the daemon for lack of memory.
`chassis`	`string`	`-`	Chassis serial number string to include in diagnosis results. This can be set manually if a platform cannot determine it automatically.
`ckpt.dir`	`string`	`var/fm/fmd/ckpt`	Directory relative to `rootdir` in which checkpoint directories will be created.
`ckpt.dirmode`	`int32`	`0700`	Permissions mode to use for checkpoint directories.
`ckpt.mode`	`int32`	`0400`	Permissions mode to use for checkpoint files.
`ckpt.restore`	`boolean`	`true`	If set, restore checkpoints at module load time.
`ckpt.save`	`boolean`	`true`	If set, save checkpoints as modules receive events.
`ckpt.zero`	`boolean`	`false`	If set, delete any saved checkpoint files before loading modules.

TABLE 8–2 Fault Manager Configuration Properties *(Continued)*

Name	Type	Default	Description
`client.buflim`	`size`	`10m`	Maximum space that may be consumed by all client module buffers.
`client.dbout`	`enum (stderr,syslog)`	`none`	Destination for client module calls to `fmd_hdl_debug()` and `fmd_hdl_error()`. If the `fg` property is set to true and this property is not modified, it will default to `stderr`.
`client.debug`	`boolean`	`false`	Boolean indicating whether client debugging and error calls should be active (true) or be discarded (false).
`client.error`	`enum (unload, stop, abort)`	`unload`	Policy for module runtime errors. If the property is set to `unload`, the module will be disabled. If the property is set to `stop`, the fault manager will stop so that a debugger can be attached. If the property is set to `abort`, the fault manager will raise a `SIGABRT` and core dump for post-mortem analysis.
`client.memlim`	`size`	`10m`	Maximum dynamic memory allocations permitted by each client module.
`client.evqlim`	`uint32`	`256`	Maximum length of client module event queue before incoming events are dropped.
`client.thrlim`	`uint32`	`8`	Maximum number of auxiliary threads that a client module may have at any given time.
`client.thrsig`	`enum`	`SIGUSR1`	Signal to use for the implementation of the `fmd_thr_signal()` interface.
`client.tmrlim`	`uint32`	`1024`	Maximum number of outstanding timer requests permitted to each client.
`client.xprtlim`	`uint32`	`256`	Maximum number of event transports that each client may have open at any given time.
`client.xprtlog`	`boolean`	`false`	If set, create a debugging log of all the events received by a given transport in `fmd/xprt/`.
`client.xprtqlim`	`uint32`	`256`	Maximum length of client transport event queue before outbound events are dropped.

TABLE 8–2 Fault Manager Configuration Properties *(Continued)*

Name	Type	Default	Description
clock	enum (native, simulated)	native	Clock operation mode. In native clock mode, the fault manager time routines are the system's high-resolution and time-of-day clock. In simulated mode, the fault manager's clock is an internal, static clock which can be adjusted only by simulation directives from `fminject`.
conf_path	path	See description following table.	Path to use to locate daemon configuration files.
conf_file	string	fmd.conf	Pathname of individual configuration files (to be appended to each component of the `conf_path`).
core	boolean	false	If set, force a core dump when `fmd` exits in order to perform post-mortem memory leak detection.
dbout	enum (stderr,syslog)	none	Destination for daemon calls to its internal debug output function. If the `fg` property is set to true and this property is not modified, it will default to `stderr`.
debug	enum	none	Comma-separated list of tokens describing subsystems whose debugging output should be enabled. Use `fmd -o debug=help` to display a list of valid tokens.
dictdir	string	usr/lib/fm/dict	Directory relative to `rootdir` which is used as the default location of `libdiagcode` dictionaries.
domain	string	-	Domain identifier to be included in fault diagnosis events. This value can be set if the fault manager is deployed on a service processor on behalf of a particular domain.
fg	boolean	false	If set, run the daemon in the foreground, keeping `stdout` and `stderr` open and skipping normal backgrounding procedures such as forking and becoming a session leader.
gc_interval	time	1d	Interval at which to run module garbage collection callbacks.

TABLE 8–2 Fault Manager Configuration Properties *(Continued)*

Name	Type	Default	Description
`ids.avg`	`uint32`	`4`	Average desired length for identifier hash chains.
`ids.max`	`uint32`	`1024`	Maximum hash bucket array size for identifier hash chains.
`isaname`	`string`	`‘uname -p‘`	Processor ISA name.
`log.creator`	`string`	`fmd`	Exact log file creator tag.
`log.error`	`string`	`var/fm/ fmd/errlog`	Error log pathname.
`log.fault`	`string`	`var/fm/ fmd/fltlog`	Fault log pathname.
`log.minfree`	`size`	`2m`	Free space to keep in filesystem containing checkpoints and logs. If less than `log.minfree` bytes are free in this filesystem, `fmd` will not attempt to write log records and instead treat them as if they failed due to `ENOSPC`.
`log.rsrc`	`string`	`var/fm/ fmd/rsrc`	Resource log file directory.
`log.tryrotate`	`uint32`	`10`	Maximum number of attempts to perform a log rotation before giving up and returning failure to `fmadm rotate`.
`log.waitrotate`	`time`	`200ms`	Time to wait between each failed log rotation attempt.
`log.xprt`	`string`	`var/fm/ fmd/xprt`	Transport debugging log file directory.
`machine`	`string`	`‘uname -m‘`	Machine class name.
`nodiagcode`	`string`	`-`	String to use when no diagnosis code dictionary is found for either the module or for the fault manager.
`osrelease`	`string`	`‘uname -r‘`	Operating system release string.
`osversion`	`string`	`‘uname -v‘`	Operating system version string.
`platform`	`string`	`‘uname -i‘`	Platform name string.
`plugin.close`	`boolean`	`true`	If set, plug-in modules should be closed by `dlclose()` on unload. This option can be set to `false` for debugging purposes.
`plugin.path`	`path`	See description following table.	Path to use to locate plug—in modules.

TABLE 8–2 Fault Manager Configuration Properties *(Continued)*

Name	Type	Default	Description
product	string	smbios(7D) product string, if available	Product string to use in local FMRI authority. If no product string can be found, the value of the platform property will be used for authorities instead.
rootdir	string	\0	Root directory for pathname expansions. The empty string results in a root of /.
rpc.adm.path	string	-	If set, pathname to use to write out a file containing the RPC program number fmd is using for the FMD_ADM protocol.
rpc.adm.prog	uint32	100169	RPC program number to use for the FMD_ADM protocol.
rpc.api.path	string	-	If set, pathname to use to write out a file containing the RPC program number fmd is using for the FMD_API protocol.
rpc.api.prog	uint32	100170	RPC program number to use for the FMD_API protocol.
rpc.rcvsize	size	128k	Maximum size of RPC receive buffer to request from the underlying transport.
rpc.sndsize	size	128k	Maximum size of RPC send buffer to request from the underlying transport.
rsrc.age	time	30d	Maximum age of a resource cache entry for a resource that is no longer present in the system. The fault manager will automatically delete cache log files for missing resources that exceed the maximum age.
rsrc.zero	boolean	false	If set, delete all resource cache entries on startup.
schemedir	string	usr/lib/fm/fmd/schemes	Directory to use to locate scheme plug-in libraries.
self.name	string	fmd-self-diagnosis	Name of the diagnosis engine to use for self-diagnosis.
self.dict	string	FMD.dict	Name of the diagnosis code dictionary for the fault manager.
server	string	uname -n	Server name that is running the fault manager.

TABLE 8–2 Fault Manager Configuration Properties *(Continued)*

Name	Type	Default	Description
`strbuckets`	`uint32`	`211`	Number of hash table buckets to use for string hash tables.
`trace.mode`	`enum (none, lite, full)`	`lite (non-DEBUG), full (DEBUG)`	Internal trace buffer mode. If this mode is set to `none`, trace buffer calls are ignored. If this mode is set to `lite`, trace buffer calls record the trace string, timestamp, and `errno`. If this mode is set to `full`, all of the `lite` tracing is performed and a stack trace is captured.
`trace.recs`	`uint32`	`128`	Number of trace records to pre-allocate for each fault manager thread.
`trace.frames`	`uint32`	`16`	Maximum number of stack trace frames to sample when tracing.
`uuidlen`	`uint32`	`36`	Length of UUID strings.
`xprt.ttl`	`uint8`	`1`	Default time-to-live for events. A time-to-live of one indicates an event can make one hop over a transport before it will expire.

The default value of the `agent.path` variable is as follows:

- `%r/usr/platform/%i/lib/fm/fmd/agents:`
- `%r/usr/platform/%m/lib/fm/fmd/agents:`
- `%r/usr/lib/fm/fmd/agents`

The default value of the `conf_path` variable is as follows:

- `%r/usr/lib/fm/fmd:`
- `%r/usr/platform/%m/lib/fm/fmd:`
- `%r/usr/platform/%i/lib/fm/fmd:`
- `%r/etc/fm/fmd`

The default value of the `plugin.path` variable is as follows:

- `%r/usr/platform/%i/lib/fm/fmd/plugins:`
- `%r/usr/platform/%m/lib/fm/fmd/plugins:`
- `%r/usr/lib/fm/fmd/plugins`

8.2 Command-line Options

The fault manager supports three command-line options described in its man page that can be used to simplify life for developers. The `-f` option can be used to force the fault manager to process the specified configuration file before any other configuration files, permitting all of the default options to be modified. The `-R` option can be used to reset the value of the `rootdir` property before any configuration file processing occurs. The `-o` option can be used to set one or more individual configuration file properties *after* other configuration files have been processed (for example, `fmd -o fg=true`). The arguments to `-o` can be preceded by a module name and a colon (:) to indicate that the property should be set for the specified module when it loads (for example, `fmd -o syslog-msgs:gmt=true`).

8.3 Event Transports

The Solaris reference implementation of the fault manager uses the SysEvent facility as a default transport for incoming telemetry events. This service is provided by the built-in module `sysevent-transport`, described further in Chapter 4, "Event Transports." The details of event handling are described in further detail in Chapter 3, "Events." The event transport is also used to ensure that only one fault manager is running at a time by requesting exclusive access to the channel which is enforced by the kernel. The default event transport also provides protection against fault manager crashes by ensuring that the transport continues queuing events while the fault manager restarts, and that events are consumed in a two-phase fashion whereby the fault manager first writes an event to its log, and then confirms with the transport that it can be deleted from the transport's queue. If the fault manager is ported to other operating systems, details of alternate event transports will be added to this document.

8.4 RPC Services

The fault manager also functions as an RPC server in order to implement facilities for administrative tools (and in the future, to support external modules). The details of the RPC API are Project Private, and a set of Consolidation Private libraries provides a wrapper around the RPC invocations, permitting future FMA projects such as graphical user interfaces to be developed without forcing these clients to program to RPC directly (and thereby retaining flexibility of implementation for the underlying transport). The fault manager only advertises itself as an RPC server on loopback transports on the local Solaris system, and uses the Solaris user credentials mechanism to verify that callers have appropriate privileges for the requested operations. By default, the library uses the Solaris RPC doors transport for RPC communication, which can operate even if `rpcbind` is not running on the system.

8.5 Service Manifest

The fault manager delivers a *service manifest* to describe itself to the Solaris Service Management Facility (SMF). The service identifier used by the fault manager is `svc:/system/fmd:default`. A copy of the service manifest can be found on each Solaris system in the file `/var/svc/manifest/system/fmd.xml`. The SMF `svcs` and `svcadm` commands can be used to query the status of the fault manager and enable, disable, and restart the service. If the fault manager dies as the result of a software bug or as the victim of an underlying hardware error, SMF will restart it automatically, and the fault manager checkpoint mechanisms and event transport will restore the appropriate state. If a fault manager is deployed on another operating system that does not support SMF, additional software to support restart of the fault manager should be provided.

8.6 Signal Handling

Fault manager threads execute with all signals blocked except for SIGABRT (so that `assert()` calls may be used). The thread that executes the fault manager's `main()` function waits for external signals using `sigsuspend()` after it has completed initialization of the rest of the fault manager. By default, this thread ignores all signals except for `SIGTERM`. If `SIGTERM` is received, the main thread will shut down all other threads in an orderly fashion, checkpoint all loaded modules, and exit. If the `fg` property is set to `true`, the main thread will additionally receive `SIGHUP` and `SIGINT` and use these as alternate termination signals, and it will enable the job-control signals `SIGTTIN`, `SIGTTOU`, and `SIGTSTP` so that these may be used while debugging or executing a simulation.

8.7 Privilege Model

The fault manager uses the Solaris *least privilege* APIs to reduce its effective privilege set to the minimum number of privileges required for its operation and the operation of its client modules. The set of privileges used by the fault manager can be displayed using the Solaris `ppriv` utility, as shown in the following example:

```
# ppriv 'pgrep fmd'
100394: /usr/lib/fm/fmd/fmd
flags = PRIV_AWARE
        E: basic,file_dac_execute,file_dac_read,file_dac_search,file_dac_write,
           file_owner,proc_owner,proc_priocntl,
           sys_admin,sys_config,sys_devices,sys_res_config
        I: basic,file_dac_execute,file_dac_read,file_dac_search,file_dac_write,
           file_owner,proc_owner,proc_priocntl,
           sys_admin,sys_config,sys_devices,sys_res_config
        P: basic,file_dac_execute,file_dac_read,file_dac_search,file_dac_write,
```

```
        file_owner,proc_owner,proc_priocntl,
        sys_admin,sys_config,sys_devices,sys_res_config
    L: basic,file_dac_execute,file_dac_read,file_dac_search,file_dac_write,
        file_owner,proc_owner,
        proc_priocntl,sys_admin,sys_config,sys_devices,sys_res_config
```

The individual privileges are described in the Solaris `privileges`(5) manual page.

◆ ◆ ◆ CHAPTER 9

Topology

This chapter discusses the following topics:

To view the reference implementation of the topology library that is discussed in this chapter, see the `libtopo` driver in OpenSolaris at `http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/fm/topo/libtopo/`

9.1 What is a Topology Snapshot?

A topology snapshot is a view in-time of resources used in fault management activities (error handling, diagnosis, recovery or repair). The `libtopo` library provides interfaces for the creation and destruction of a topology snapshot in addition to interfaces to access resource instances within the snapshot. A topology snapshot is represented by a root node with FMRI scheme-specific sub-topologies for the following:

cpu	Solaris logical CPU topology
dev	Solaris device tree topology
hc	Physical hardware component topology
mod	Solaris module list
pkg	Solaris package list

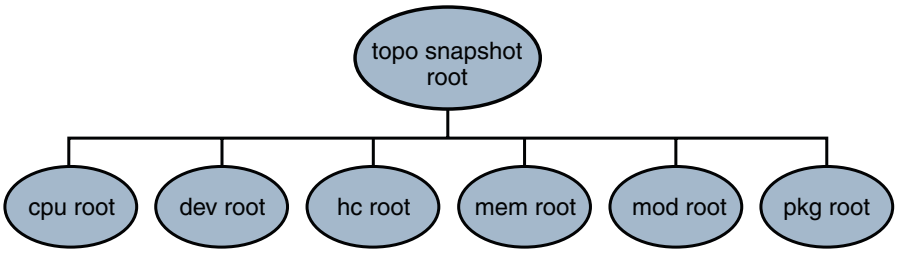


FIGURE 9–1 Topology Snapshot

Each resource instance in a topology snapshot is represented by a node. Topology nodes can be strung together as a list as in the cpu topology in the following figure, or nodes can be arranged hierarchically as in the physical paths to hardware components (hc).

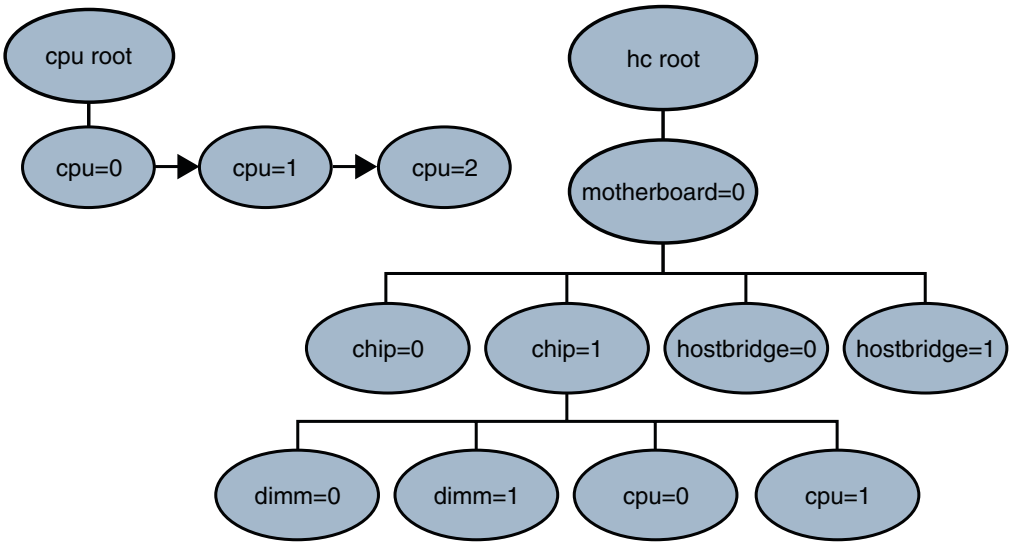


FIGURE 9–2 Topology Nodes

Each topology node must contain at least one property grouping to support the FMA Event protocol for its FMRI and optional properties for associated ASRU and FRU FMRIs or label string. Protocol property information for each topology node can be viewed with the `fmtopo`(1M) command, as shown in the following example:

```
# fmtopo -p hc://motherboard=0/hostbridge=0/pciexrc=0/pciexbus=1/pciexdev=0

hc://motherboard=0/hostbridge=0/pciexrc=0/pciexbus=1/pciexdev=0
        ASRU: dev:////pci@0,0/pci1002,5a34@2
        FRU: hc://:chassis-id=LXFR506119641092A52500/motherboard=0
        Label: MB
```

See Chapter 12, "fmtopo Utility," for more information on the fmtopo command.

The snapshot and its topologies are derived by a set of scheme-specific built-in plug-in enumerators managed by libtopo, subsystem-specific plug-in enumerators, and topology map file statements written in XML according to the libtopo DTD (document type definition) specification.

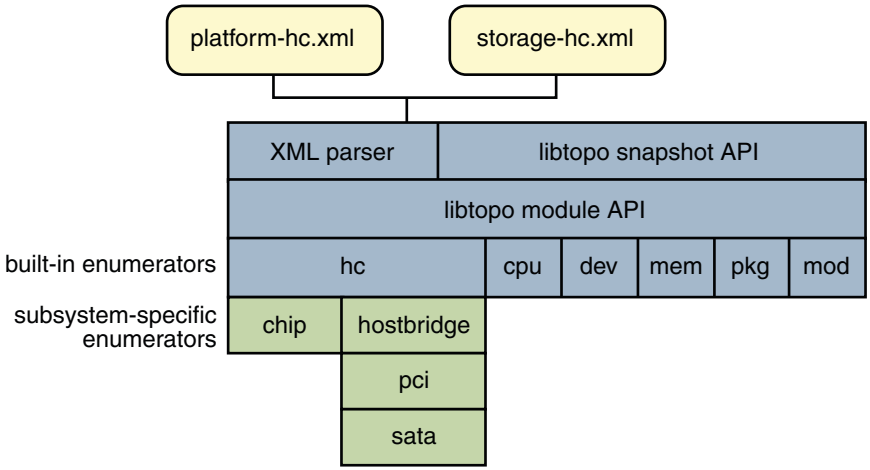


FIGURE 9–3 Topology Enumerators

9.2 Topology Snapshot API

Consumers of topology data can use the interfaces discussed in this section to initialize an opaque handle, take and release a snapshot for that handle, and access date stored in the snapshot.

Each snapshot is assigned a Universally Unique Identifier (UUID). In a future enhancement to the topology library API, this identifier will be used as the file locator to persist snapshots or lookup a previously captured snapshot. Topology handles can be reused to take new snapshots when the system configuration is known to have changed. For example, the fault manager listening for EC_DR SysEvents can release a preexisting snapshot for a given handle and take another snapshot to give its plug-in modules access to the latest system information.

9.2.1 topo_open()

```
topo_hdl_t *topo_open(int version, const char *rootdir, int *errp)
```

Return an opaque handle for accessing a topology snapshot in a subsequent call to topo_snap_hold(). The *version* indicates the library ABI version to be used. The value of

version should be specified as `TOPO_VERSION`. Plug-in modules and topology map XML files are loaded according to *rootdir* and the pathname expansions described below. If *rootdir* is NULL, *rootdir* expands to `/`.

The following list gives module plug-in pathname expansions:

- *rootdir*`/usr/platform/`*platform*`/lib/fm/topo/plugins` (where *platform* is `uname -i` by default)
- *rootdir*`/usr/platform/`*machine*`/lib/fm/topo/plugins` (where *machine* is `uname -m` by default)
- *rootdir*`/usr/lib/fm/topo/`*scheme*`/plugins`

The following list gives topology map file pathname expansions:

- *rootdir*`/usr/platform/`*platform*`/lib/fm/topo/maps` (where *platform* is `uname -i` by default)
- *rootdir*`/usr/platform/`*machine*`/lib/fm/topo/maps` (where *machine* is `uname -m` by default)
- *rootdir*`/usr/lib/fm/topo/`*scheme*`/maps`

If an error is detected during `topo_open()`, **errp* contains one of the following `libtopo` error codes:

`ETOPO_HDL_ABIVER`	Library ABI version is not supported
`ETOPO_HDL_INVAL`	Invalid *rootdir*
`ETOPO_HDL_NOMEM`	Insufficient memory to allocate handle

9.2.2 topo_close()

```
void topo_close(topo_hdl_t *thp)
```

Close a previously allocated handle. All snapshot resources associated with *thp* are released.

9.2.3 topo_snap_hold()

```
char * topo_snap_hold(topo_hdl_t *thp, const char *notused, int *errp)
```

Take a topology snapshot for a previously allocated handle, *thp*. A universally unique snapshot identifier (UUID) is returned on success. On failure, NULL is returned and **errp* is set to one of the following error codes:

`ETOPO_HDL_UUID`	Snapshot already in use for *thp*

ETOPO_HDL_NOMEM	Insufficient memory to allocate snapshot

The caller is responsible for deallocating the memory associated with the UUID string by calling topo_hdl_strfree().

9.2.4 topo_snap_release()

```
void topo_snap_release(topo_hdl_t *thp)
```

Release all snapshot resources associated with *thp*.

9.2.5 topo_walk_init()

```
topo_walk_t *topo_walk_init(topo_hdl_t *thp, const char *scheme,
    topo_walk_cb_t cb_f, void *pdata, int *errp)
```

Initialize a topology walk structure for stepping through the snapshot associated with *thp*. On success, a pointer to a snapshot walk structure is returned for the topology defined by *scheme*. Subsequent calls to topo_walk_step() invoke the supplied callback function for each topology node with *pdata*. On failure, topo_walk_init() returns NULL and **errp* is set to one of the following error codes:

ETOPO_WALK_NOTFOUND	Invalid topology *scheme*
ETOPO_WALK_NOMEM	Insufficient memory to allocate walk structure

The walker callback function is passed *thp*, a pointer to the current topology node handle, and *pdata*:

```
typedef int (*topo_walk_cb_t)(topo_hdl_t *thp, tnode_t *nodep, void *pdata)
```

The *pdata* data is private and is managed solely by the caller and callback function.

The callback can return one of the following status values:

TOPO_WALK_TERMINATE	Terminate the walk
TOPO_WALK_NEXT	Walk the next child or sibling node

9.2.6 topo_walk_step()

```
int topo_walk_step(topo_walk_t *wp, int flag)
```

Recursively step through a topology snapshot and invoke the callback function previously set up in *wp* by `topo_walk_init()`. The *wp* value is a pointer to a previously allocated topology walk structure. The *flag* value specifies the type of walk to perform:

`TOPO_WALK_CHILD`	Walk in depth-first order
`TOPO_WALK_SIBLING`	Walk in breadth-first order

The `topo_walk_step()` function returns the following status:

`TOPO_WALK_TERMINATE`	Walk terminated successfully by callback function
`TOPO_WALK_ERR`	Walk terminated with an error by callback function

9.2.7 topo_walk_fini()

```
void topo_walk_fini(topo_walk_t *wp)
```

Release the walk structure associated with *wp*.

9.3 Walker Helpers

The helper functions described in this section are designed to access topology node information from a walker callback function. These routines can also be called from enumerator plug-ins during the enumeration or method operations described in Enumeration Module Programming Model.

9.3.1 topo_node_name()

```
char *topo_node_name(tnode_t *nodep)
```

Return the name of the topology node pointed to by *nodep*.

9.3.2 topo_node_instance()

```
topo_instance_t topo_node_instance(tnode_t *nodep)
```

Return the topology instance of the topology node pointed to by *nodep*.

9.3.3 topo_node_getspecific()

```
void *topo_node_getspecific(tnode_t *nodep)
```

Return the topology node private data assigned by the node enumerator plug-in for *nodep*.

9.3.4 topo_node_fru()

```
int topo_node_fru(tnode_t *nodep, nvlist_t **fru, nvlist_t *priv, int *errp)
```

The `topo_node_fru()` function updates the name-value pair list pointer to *fru* with the FRU property in the protocol property group for *nodep*. The *priv* data is not interpreted by `libtopo` and contains enumerator private data for constructing the FRU property dynamically. On success, 0 is returned. On failure, -1 is returned and **errp* is set to one of the following error codes:

`ETOPO_PROP_NVL`	Error occurred while managing property `nvlist`
`ETOPO_PROP_NOMEM`	Insufficient memory to allocate walk structure

9.3.5 topo_node_asru()

```
int topo_node_asru(tnode_t *nodep, nvlist_t **asru, nvlist_t *priv, int *errp)
```

The `topo_node_asru()` function updates the `nvlist` pointer to *asru* with the ASRU property in the protocol property group for *nodep*. The *priv* data is not interpreted by `libtopo` and contains enumerator private data for constructing the ASRU property dynamically. On success, 0 is returned. On failure, -1 is returned and **errp* is set to one of the following error codes:

`ETOPO_PROP_NVL`	Error occurred while managing property `nvlist`
`ETOPO_PROP_NOMEM`	Insufficient memory to allocate walk structure

9.3.6 topo_node_resource()

```
int topo_node_resource(tnode_t *nodep, nvlist_t **resource, int *errp)
```

The `topo_node_resource()` function updates the `nvlist` pointer to *resource* with the resource property in the protocol property group for *nodep*. On success, 0 is returned. On failure, -1 is returned and **errp* is set to one of the following error codes:

`ETOPO_PROP_NVL`	Error occurred while managing property `nvlist`
`ETOPO_PROP_NOMEM`	Insufficient memory to allocate walk structure

9.3.7 topo_node_label()

```
int topo_node_label(tnode_t *nodep, char **label, int *errp)
```

The `topo_node_resource()` function updates the pointer to *label* with the label property in the protocol property group for *nodep*. On success, 0 is returned. On failure, -1 is returned and **errp* is set to one of the following error codes:

`ETOPO_PROP_NVL`	Error occurred while managing property `nvlist`
`ETOPO_PROP_NOMEM`	Insufficient memory to allocate walk structure

9.3.8 topo_method_invoke()

```
int topo_method_invoke(tnode_t *nodep, const char *method,
    topo_version_t version, nvlist_t *in, nvlist_t **out, int *errp)
```

Invoke the method operation for *nodep*. The version indicates the version of the method to invoke with an `nvlist` of input parameters, *in*. Upon successful completion, 0 is returned and an `nvlist` of output parameters is contained in *out*. On failure, -1 is returned and **errp* is set to one of the following error codes:

`ETOPO_METHOD_VERNEW`	Method version is newer than the method registered for this node
`ETOPO_METHOD_VEROLD`	Method version is older than the method registered for this node
`ETOPO_METHOD_FAIL`	Unknown method failure
`ETOPO_METHOD_NOTSUP`	Method not supported for this node

Methods are private interfaces between `libtopo` snapshot consumers and the enumerator plug-ins exporting the methods. `libtopo` does not interpret the method name, input, or output parameter lists.

9.4 Topology Node Properties

Topology nodes are permitted to contain property information that helps further describe the resource. Property information is organized according to property groupings. Each property group defines a name, a stability level for that name, a stability level for all underlying property data (name, type, values), a version for the property group definition, and a list of uniquely defined properties.

Property group versions are incremented when one of the following changes occurs:

- A property name changes
- A property type changes

- A property definition is removed from the group

Compatible changes such as new property definitions in the group do not require version changes.

Each property defines a name, a type, and a value that are unique within the group. Properties can be statically defined as `int32`, `uint32`, `int64`, `uint64`, `fmri`, string, or arrays of each type. Properties can also be dynamically exported via module registered methods. For example, a plug-in can register a method to export an ASRU plug-in property that is dynamically constructed when a call `to to topo_node_asru()` is invoked for a particular topology node.

Static properties are persistently attached to topology nodes during enumeration by an enumeration module using the `topo_prop_set*()` family of routines or as part of XML statements in a topology map file. Similarly, property methods are registered during enumeration or as part of statements in topology map files.

Module writers and snapshot consumers can retrieve property information by using the functions described in this section. Topology properties can be viewed by using the `fmtopo`(1M) command, as shown in the following example:

```
# fmtopo -P all hc://motherboard=0/chip=0/cpu=1

hc://motherboard=0/chip=0/cpu=1
  group: protocol                        version: 1   stability: Private/Private
    resource          fmri      hc://:product-id=Ferrari-5000:chassis-id=LXFR...
    ASRU              fmri      cpu:///cpuid=1
    FRU               fmri      hc://:product-id=Ferrari-5000:chassis-id=LXFR...
  group: authority                       version: 1   stability: Private/Private
    product-id        string    Ferrari-5000
    chassis-id        string    LXFR506119641092A52500
    server-id         string    midlife
  group: cpu-properties                  version: 1   stability: Private/Private
    cpuid             uint32    0x1
    chip_id           int32     0
    core_id           int32     1
    clog_id           int32     1
```

9.4.1 topo_prop_get_int32()

```
int topo_prop_get_int32(tnode_t *nodep, const char *pgname,
    const char *pname, int32_t *val, int *errp)
```

Get the `int32` property value named by *pname* in the *pgname* group for *nodep*. On success, 0 is returned and *val* contains the property value for *pname*. On failure, -1 is returned and **errp* is set to one of the following error codes:

ETOPO_PROP_NOENT	Property name or group does not exist
ETOPO_PROP_NVL	Property nvlist management failure

9.4.2 topo_prop_get_uint32()

```
int topo_prop_get_uint32(tnode_t *nodep, const char *pgname,
    const char *pname, uint32_t *val, int *errp)
```

Get the uint32 property value named by *pname* in the *pgname* group for *nodep*. On success, 0 is returned and *val* contains the property value for *pname*. On failure, -1 is returned and **errp* is set to one of the following error codes:

ETOPO_PROP_NOENT	Property name or group does not exist
ETOPO_PROP_NVL	Property nvlist management failure

9.4.3 topo_prop_get_int64()

```
int topo_prop_get_int64(tnode_t *nodep, const char *pgname,
    const char *pname, uint32_t *val, int *errp)
```

Get the int64 property value named by *pname* in the *pgname* group for *nodep*. On success, 0 is returned and *val* contains the property value for *pname*. On failure, -1 is returned and **errp* is set to one of the following error codes:

ETOPO_PROP_NOENT	Property name or group does not exist
ETOPO_PROP_NVL	Property nvlist management failure

9.4.4 topo_prop_get_uint64()

```
int topo_prop_get_uint64(tnode_t *nodep, const char *pgname,
    const char *pname, uint32_t *val, int *errp)
```

Get the uint64 property value named *pname* in the group *pgname* for *nodep*. On success, 0 is returned and *val* contains the property value for *pname*. On failure, -1 is returned and **errp* is set to one of the following error codes:

ETOPO_PROP_NOENT	Property name or group does not exist
ETOPO_PROP_NVL	Property nvlist management failure

9.4.5 topo_prop_get_string()

```
int topo_prop_get_string(tnode_t *nodep, const char *pgname,
    const char *pname, char **val, int *errp)
```

Get the string property value named by *pname* in the group *pgname* for *nodep*. On success, 0 is returned and **val* is updated with a pointer to a newly allocated string for property *pname*. On failure, -1 is returned and **errp* is set to one of the following error codes:

ETOPO_PROP_NOENT	Property name or group does not exist
ETOPO_PROP_NVL	Property nvlist management failure
ETOPO_PROP_NOMEM	Insufficient memory to allocate string

The caller is responsible for deallocating **val* when it is no longer needed.

9.4.6 topo_prop_get_fmri()

```
int topo_prop_get_fmri(tnode_t *nodep, const char *pgname,
    const char *pname, nvlist_t **val, int *errp)
```

Get the FMRI property value named by *pname* in the group *pgname* for *nodep*. On success, 0 is returned and **val* is updated with a pointer to a newly allocated nvlist. On failure, -1 is returned and **errp* is set to one of the following error codes:

ETOPO_PROP_NOENT	Property name or group does not exist
ETOPO_PROP_NVL	Property nvlist management failure
ETOPO_PROP_NOMEM	Insufficient memory to allocate nvlist

The caller is responsible for deallocating **val* when it is no longer needed.

9.4.7 topo_prop_get_int32_array()

```
int topo_prop_get_int32_array(tnode_t *nodep, const char *pgname,
    const char *pname, int32 **val, uint_t *nelem, int *errp)
```

Get the int32 array property value named by *pname* in the group *pgname*, for *nodep*. The value of **nelem* is updated with the size of the array. On success, 0 is returned and **val* is updated with a pointer to a newly allocated array of int32_t. On failure, -1 is returned and **errp* is set to one of the following error codes:

ETOPO_PROP_NOENT	Property name or group does not exist
ETOPO_PROP_NVL	Property nvlist management failure

ETOPO_PROP_NOMEM	Insufficient memory to allocate array

The caller is responsible for deallocating **val* when it is no longer needed.

9.4.8 topo_prop_get_uint32_array()

```
int topo_prop_get_uint32_array(tnode_t *nodep, const char *pgname,
    const char *pname, uint32 **val, uint_t *nelem, int *errp)
```

Get the uint32 array property value named by *pname* in the group *pgname*, for *nodep*. The value of **nelem* is updated with the size of the array. On success, 0 is returned and **val* is updated with a pointer to a newly allocated array of uint32_t. On failure, -1 is returned and **errp* is set to one of the following error codes:

ETOPO_PROP_NOENT	Property name or group does not exist
ETOPO_PROP_NVL	Property nvlist management failure
ETOPO_PROP_NOMEM	Insufficient memory to allocate array

The caller is responsible for deallocating **val* when it is no longer needed.

9.4.9 topo_prop_get_int64_array()

```
int topo_prop_get_int64_array(tnode_t *nodep, const char *pgname,
    const char *pname, int64 **val, uint_t *nelem, int *errp)
```

Get the int64 array property value named by *pname* in the group *pgname*for *nodep*. The value of **nelem* is updated with the size of the array. On success, 0 is returned and **val* is updated with a pointer to a newly allocated array of int64_t. On failure, -1 is returned and **errp* is set to one of the following error codes:

ETOPO_PROP_NOENT	Property name or group does not exist
ETOPO_PROP_NVL	Property nvlist management failure
ETOPO_PROP_NOMEM	Insufficient memory to allocate array

The caller is responsible for deallocating **val* when it is no longer needed.

9.4.10 topo_prop_get_uint64_array()

```
int topo_prop_get_uint64_array(tnode_t *nodep, const char *pgname,
    const char *pname, uint64 **val, uint_t *nelem, int *errp)
```

Get the `uint64` array property value named by *pname* in the group *pgname*for *nodep*. The value of **nelem* is updated with the size of the array. On success, 0 is returned and **val* is updated with a pointer to a newly allocated array of `uint64_t`. On failure, -1 is returned and **errp* is set to one of the following error codes:

`ETOPO_PROP_NOENT`	Property name or group does not exist
`ETOPO_PROP_NVL`	Property `nvlist` management failure
`ETOPO_PROP_NOMEM`	Insufficient memory to allocate array

The caller is responsible for deallocating **val* when it is no longer needed.

9.4.11 topo_prop_get_string_array()

```
int topo_prop_get_string_array(tnode_t *nodep, const char *pgname,
    const char *pname, char ***val, uint_t *nelem, int *errp)
```

Get the string array property value named by *pname* in the group *pgname* for *nodep*. The value of **nelem* is updated with the size of the array. On success, 0 is returned and ***val* is updated with a pointer to a newly allocated array of strings. On failure, -1 is returned and **errp* is set to one of the following error codes:

`ETOPO_PROP_NOENT`	Property name or group does not exist
`ETOPO_PROP_NVL`	Property `nvlist` management failure
`ETOPO_PROP_NOMEM`	Insufficient memory to allocate array

The caller is responsible for deallocating the string array (***val*) and each string element when they are no longer needed.

9.4.12 topo_prop_get_fmri_array()

```
int topo_prop_get_fmri_array(tnode_t *nodep, const char *pgname,
    const char *pname, nvlist_t ***val, uint_t *nelem, int *errp)
```

Get the FMRI `nvlist` array property value named by *pname* in the group *pgname* for *nodep*. The value of **nelem* is updated with the size of the array. On success, 0 is returned and ***val* is updated with a pointer to a newly allocated array of FMRI `nvlist`. On failure, -1 is returned and **errp* is set to one of the following error codes:

`ETOPO_PROP_NOENT`	Property name or group does not exist
`ETOPO_PROP_NVL`	Property `nvlist` management failure
`ETOPO_PROP_NOMEM`	Insufficient memory to allocate array

The caller is responsible for deallocating the `nvlist` array (***val*) and each `nvlist` element when they are no longer needed.

9.5 Snapshot Access by FMRI

The topology library supports a collection of functions designed to replace and enhance the functions exported by the `fmd`(1M) scheme plug-in API described in Section 6.3, "Scheme Plug-in Interfaces." In addition to those supported by `fmd`, `libtopo` exports a number of additional helper functions that can be invoked for specific FMRIs in a given snapshot. The programming interface for these functions is specified in the file `fm/libtopo.h` and is described in this section.

9.5.1 topo_fmri_present()

```
int topo_fmri_present(topo_hdl_t *thp, nvlist_t *fmri, int *errp)
```

Return a status value that indicates whether the specified *fmri* is present in the snapshot for *thp*. If serial identity is available for the FMRI, it is employed to verify the identity of the FMRI in addition to comparing the resource location information. If the resource is present, the function returns 1. If the resource is not present, the function returns 0. If the FMRI is invalid or an error occurs, the function returns -1 and **errp* contains one of the following error codes:

`ETOPO_FMRI_NVL`	FMRI `nvlist` management failure
`ETOPO_FMRI_VERSION`	Invalid FMRI version
`ETOPO_FMRI_MALFORM`	Malformed FMRI
`ETOPO_FMRI_NOMEM`	Insufficient memory to perform operation
`ETOPO_METHOD_NOTSUP`	Present method not supported

9.5.2 topo_fmri_contains()

```
int topo_fmri_contains(topo_hdl_t *thp, nvlist_t *fmri1, nvlist_t *fmri2, int *errp)
```

Return a status value that indicates whether the `nvlist` that corresponds to *fmri1* contains the `nvlist` *fmri2* in the snapshot represented by *thp*. The notion of containment for a given resource is entirely specific to each resource type and scheme. If *fmri1* contains *fmri2*, the function returns 1. If not, the function returns 0. If an error occurs, -1 is returned and **errp* is updated with one of the following error codes:

`ETOPO_FMRI_NVL`	FMRI `nvlist` management failure

ETOPO_FMRI_VERSION	Invalid FMRI version
ETOPO_FMRI_MALFORM	Malformed FMRI
ETOPO_FMRI_NOMEM	Insufficient memory to perform operation
ETOPO_METHOD_NOTSUP	Contains method not supported

9.5.3 topo_fmri_unusable()

```
int topo_fmri_unusable(topo_hdl_t *thp, nvlist_t *fmri, int *errp)
```

Return a status value that indicates whether the *fmri* is usable on the system snapshot represented by *thp* or whether the *fmri* is disabled. The means for disabling a given resource is specific to each resource type and scheme. Examples of this specificity include page retirement and CPU offlining. If the *fmri* is unusable (currently disabled), the function returns 1. If the *fmri* is usable, the function returns 0. If an error occurs, -1 is returned and **errp* is updated with one of the following error codes:

ETOPO_FMRI_NVL	FMRI `nvlist` management failure
ETOPO_FMRI_VERSION	Invalid FMRI version
ETOPO_FMRI_MALFORM	Malformed FMRI
ETOPO_FMRI_NOMEM	Insufficient memory to perform operation
ETOPO_METHOD_NOTSUP	Unusable method not supported

9.5.4 topo_fmri_expand()

```
int topo_fmri_expand(topo_hdl_t *thp, nvlist_t *fmri, int *errp)
```

Expand the specified *fmri* in the snapshot represented by *thp* to include any FMRI elements that are part of the scheme but not present in the specified FMRI. Typically, the `topo_fmri_expand()` mechanism is used by diagnosis engines where an initial error report event includes an FMRI but does not include certain information about the resource that cannot be captured by the error handler, such as serial number data. This function returns 0 to indicate success or -1 to indicate an error. If an error occurs, **errp* is updated with one of the following error codes:

ETOPO_FMRI_NVL	FMRI `nvlist` management failure
ETOPO_FMRI_VERSION	Invalid FMRI version
ETOPO_FMRI_MALFORM	Malformed FMRI
ETOPO_FMRI_NOMEM	Insufficient memory to perform operation

ETOPO_METHOD_NOTSUP	Expand method not supported

9.5.5 topo_fmri_nvl2str()

```
int topo_fmri_nvl2str(topo_hdl_t *thp, nvlist_t *fmri, char **buf, int *errp)
```

Convert the specified *fmri* in the snapshot represented by *thp* into its string representation and place the result in *buf*. This function returns 0 to indicate success or -1 to indicate an error. If an error occurs, **errp* is updated with one of the following error codes:

ETOPO_FMRI_NVL	FMRI `nvlist` management failure
ETOPO_FMRI_VERSION	Invalid FMRI version
ETOPO_FMRI_MALFORM	Malformed FMRI
ETOPO_FMRI_NOMEM	Insufficient memory to perform operation

The caller is responsible for deallocating the memory associated with *buf* by calling `topo_hdl_strfree()` when the string buffer is no longer needed.

9.5.6 topo_fmri_str2nvl()

```
int topo_fmri_str2nvl(topo_hdl_t *thp, const char *buf, nvlist_t **fmri, int *errp)
```

Convert the specified FMRI string, *str* to its name-value pair list representation for the snapshot represented by *thp*. On success, *fmri* is updated with a pointer to a newly allocated name-value pair list. This function returns 0 to indicate success or -1 to indicate an error. If an error occurs, **errp* is updated with one of the following error codes:

ETOPO_FMRI_NVL	FMRI `nvlist` management failure
ETOPO_FMRI_VERSION	Invalid FMRI version
ETOPO_FMRI_NOMEM	Insufficient memory to perform operation

The caller is responsible for deallocating the memory associated with **fmri* by calling `nvlist_free()` when the name-value pair list is no longer needed. See the `nvlist_alloc`(3NVPAIR) man page for more information about `nvlist_free()`.

9.5.7 topo_fmri_asru()

```
int topo_fmri_asru(topo_hdl_t *thp, nvlist_t *fmri, nvlist_t **asru, int *errp)
```

Return the ASRU associated with *fmri* for the snapshot represented by *thp*. On success, **asru* is updated with a newly allocated name-value pair list. This function returns 0 to indicate success or -1 to indicate an error. If an error occurs, **errp* is updated with one of the following error codes:

`ETOPO_FMRI_NVL`	FMRI `nvlist` management failure
`ETOPO_FMRI_VERSION`	Invalid FMRI version
`ETOPO_FMRI_NOMEM`	Insufficient memory to perform operation
`ETOPO_METHOD_NOTSUP`	ASRU construction method not supported
`ETOPO_PROP_NOENT`	ASRU property not available

The caller is responsible for deallocating the memory associated with **asru* by calling `nvlist_free()` when the name-value pair list is no longer needed.

9.5.8 topo_fmri_fru()

```
int topo_fmri_fru(topo_hdl_t *thp, nvlist_t *fmri, nvlist_t **fru, int *errp)
```

Return the FRU associated with *fmri* for the snapshot represented by *thp*. On success, **fru* is updated with a newly allocated name-value pair list. This function returns 0 to indicate success or -1 to indicate an error. If an error occurs, **errp* is updated with one of the following error codes:

`ETOPO_FMRI_NVL`	Name-value pair list management failure
`ETOPO_FMRI_VERSION`	Invalid FMRI version
`ETOPO_FMRI_NOMEM`	Insufficient memory to perform operation
`ETOPO_METHOD_NOTSUP`	FRU construction method not supported
`ETOPO_PROP_NOENT`	FRU property not available

The caller is responsible for deallocating the memory associated with **asru* by calling `nvlist_free()` when the name-value pair list is no longer needed.

9.5.9 topo_fmri_label()

```
int topo_fmri_label(topo_hdl_t *thp, nvlist_t *fmri, char **label, int *errp)
```

Return the *label* (location) property associated with *fmri* for the snapshot represented by *thp*. On success, **label* is updated with a newly allocated string. This function returns 0 to indicate success or -1 to indicate an error. If an error occurs, **errp* is updated with one of the following error codes:

`ETOPO_FMRI_NVL`	Name-value pair list management failure
`ETOPO_FMRI_VERSION`	Invalid FMRI version
`ETOPO_FMRI_NOMEM`	Insufficient memory to perform operation
`ETOPO_PROP_NOENT`	Label property not available

The caller is responsible for deallocating the memory associated with **label* by calling `topo_hdl_strfree()` when the string is no longer needed.

9.5.10 topo_fmri_compare()

```
int topo_fmri_compare(topo_hdl_t *thp, nvlist_t *fmri1, nvlist_t *fmri2, int *errp)
```

Return a status value that indicates whether the name-value pair list that corresponds to *fmri1* is equal to the name-value pair list that corresponds to *fmri2* in the snapshot represented by *thp*. The notion of "equal" for two FMRIs is specific to each resource type and scheme. If *fmri1* is equal to *fmri2*, the function returns 1. If *fmri1* is not equal to *fmri2*, the function returns 0. If an error occurs, -1 is returned and **errp* is updated with one of the following error codes:

`ETOPO_FMRI_NVL`	FMRI `nvlist` management failure
`ETOPO_FMRI_VERSION`	Invalid FMRI version
`ETOPO_FMRI_MALFORM`	Malformed FMRI
`ETOPO_FMRI_NOMEM`	Insufficient memory to perform operation

9.5.11 topo_fmri_invoke()

```
int topo_fmri_invoke(topo_hdl_t *thp, nvlist_t *fmri,
    topo_walk_cb_t callback, void *pdata, int *errp)
```

Invoke the callback function *callback* for the node represented by *fmri* in the snapshot that corresponds to *thp*. The `topo_fmri_invoke()` function walks the snapshot in depth-first order searching for a topology node that matches *fmri*. When the node is found, *callback* is invoked. The walker callback function requires pointers to *thp*, to the topology node handle, and to *pdata*:

```
typedef int (*topo_walk_cb_t)(topo_hdl_t *thp, tnode_t *nodep, void *pdata)
```

The contents of *pdata* is private data and is managed solely by the caller and callback function.

The callback can return one of the following as status:

`TOPO_WALK_TERMINATE`	Terminate the walk
`TOPO_WALK_ERR`	A callback error occurred

If a matching node is found and the callback invocation is successful, `topo_fmri_invoke()` returns 0. If a matching node is not found or if an error occurs, **errp* is updated with one of the following error codes:

`ETOPO_FMRI_VERSION`	Invalid FMRI version
`ETOPO_FMRI_MALFORM`	Malformed FMRI
`ETOPO_FMRI_NOMEM`	Insufficient memory to perform operation
`TOPO_WALK_ERR`	Walk error

9.6 Snapshot Memory Management and Debugging

The topology library provides a set of functions for string deallocation and debug observability for the convenience of its clients. These functions are provided to ease programming and testing of `libtopo` enumerator modules and snapshot consumers. The Solaris reference implementation of the topology snapshot uses `libumem.so.1` to implement the memory allocation routines, offering developers additional debugging facilities. See "Topology Library Debugging" in Chapter 13, "Debugging," for more information.

9.6.1 `topo_hdl_strfree()`

```
void topo_hdl_strfree(topo_hdl_t *thp, char *string)
```

Free the memory associated with *string*, where *string* must refer to the result of a previous call to `topo_snap_hold()`, `topo_fmri_label()`, `topo_fmri_nvl2str()`, `topo_prop_get_string()` or `topo_prop_get_string_array()`. If *string* is NULL, `topo_hdl_strfree()` always succeeds and has no effect.

Note – The Solaris reference implementation of the topology library uses `libumem.so.1` to perform memory allocation, and therefore internally computes the size of the string to be freed by applying `strlen()` to it. Therefore, callers of `topo_hdl_strfree()` should take care not to insert additional `\0` characters in the string or to remove the trailing `\0`.

9.6.2 topo_strerror()

```
const char *topo_strerror(int errno)
```

Return an error message associated with a topology library *errno*. The library manages the memory allocated for the returned string.

9.6.3 topo_debug_set()

```
void topo_debug_set(topo_hdl_t *thp, const char *dbmode, const char *dout)
```

Set the debug messaging mode and output file for the snapshot represented by *thp*. The value of *dbmode* must be set to one of the valid mode strings:

`error`	Turn on debug messaging for all error conditions
`module`	Turn on debug messaging for enumerator modules
`walk`	Turn on debug messaging for snapshot walker functions
`modulesvc`	Turn on debug messaging for all module service functions
`xml`	Turn on debug messaging for XML processing
`all`	Turn on all debug messaging

The value of *dbout* must be set to `stderr` or `stdout`. The default output file is `stdout`.

Alternately, the environment variables `TOPO_DEBUG` and `TOPO_DEBUG_OUT` can also be used to turn on debug messaging. `TOPO_DEBUG` can be set to a comma-separated list of debug modes. `TOPO_DEBUG` can be used to set the debug mode for a topology library application client to observe all library errors and XML processing as shown in the following example:

```
# TOPO_DEBUG=error,xml; export TOPO_DEBUG
```

Similarly, `TOPO_DEBUG_OUT` can be set to either `stderr` or `stdout` to change the output file. The default is `stdout`.

9.7 Enumeration Module Programming Model

In a system made of various resource types, it is not possible to write a monolithic system topology enumerator for all possible resource FMRIs that you might want to snapshot for the purpose of fault management. For this reason, the topology library defines some common abstractions that we expect all topology enumeration software to require.

The enumeration programming model includes support for:

- A plug-in API for shared library enumerator module loading and unloading
- A set of common operations for enumerator registration and topology discovery
- A set of debug facilities for observing module behaviors
- A set of functions to support node and property creation
- A set of functions to support method operations for topology nodes

9.7.1 Plug-in Modules

Enumerator modules are delivered as shared library plug-ins that are installed as separate binary objects and loaded into the `fmd`(1M) address space using `dlopen`(3C). Plug-in modules are designed to enumerate scheme-specific topologies for a particular subsystem and to export method operations that can be applied to a given topology node based on its resource FMRI. Plug-in modules are installed in one of the predefined plug-in module directories according to their subsystem relevance. For example, the PCI bus module for i86pc (`pcibus.so`) is installed in the directory `/usr/platform/i86pc/lib/fm/topo/plugins/`.

Plug-in modules are loaded from these directories in the following search order:

- *rootdir*`/usr/platform/`*platform*`/lib/fm/topo/plugins`
  - The value of *rootdir* is determined by the snapshot initializer. The snapshot initializer is the caller of `topo_open()`.
  - The value of *platform* is `uname -i` by default.
- `/usr/platform/`*machine*`/lib/fm/topo/plugins`
  - The value of *machine* is `uname -m` by default.
- `/usr/lib/fm/topo/plugins`

9.7.2 Threading Model

The topology library exports a single-threaded programming model to its plug-in modules in order to simplify their design, coding, and testing. The library guarantees that only one thread will execute in any of a given module's entry points at any given time, and that only one module entry point of a given module will be executing at any time. The topology library, however, does permit multiple application threads to access a given topology snapshot, so there are some important considerations for the development and compilation of enumerator modules.

First, code that resides in separate modules can and will be executed simultaneously by the library using different threads. A module can make no assumption about how other modules are configured or are executing at a given time. Second, a module cannot assume that any particular thread is associated with its execution. Although the topology library will not execute

multiple module entry points simultaneously, no guarantees are made that the thread that executes one entry point is the same thread that will next execute that entry point. As a result, module writers must not cache thread identifiers such as `pthread_self()` persistently or use thread-specific data as a mechanism for storing module state.

A module must also be written using reentrant interfaces because multiple modules will be executing inside a given topology snapshot simultaneously. For example, modules must use `strtok_r()` rather than `strtok()` because `strtok()` uses static data inside of `libc.so.1`, and therefore multiple modules calling `strtok()` simultaneously could corrupt one another. However, all modules can use static data in their own module source code because within the module, the library guarantees that only one thread will be executing in the module code at a time. Similarly, the use of mutexes to protect data within a module is not required. Module writers can assume a single-threaded programming model based on the previously described rules.

A module must avoid using global variables to store private topology state. Global data can change unexpectedly from operation to module. For example, a global variable to cache the `devinfo` handle from `topo_mod_devinfo()` might not be valid on subsequent calls to an enumeration entry point or method operation.

9.7.3 Error Handling Model

Library errors caused by module enumeration or method operations are persisted in the opaque module handle. Access to the current error code is available by calling `topo_mod_errno()`. The error code is guaranteed to be valid immediately following invocation of a module helper function. Modules can also set the error code to a predefined set of error codes as required. See `topo_mod_seterrno()` below for a list of possible module error codes.

9.7.4 Module Loading

When a topology snapshot is taken, enumerator modules are loaded as specified in the topology map files from the search paths shown in "Plug-in Modules" above or by calling the `topo_mod_load()` function from an enumerator module that is already loaded. Modules are assigned the name that corresponds to the basename of the module object with any trailing `.so` suffix removed. Module names are kept in a single per-snapshot list managed by the topology library, and only one module of a given name is permitted at a time. If the library encounters a module in a search path whose name corresponds to an already loaded module, then the second module is silently ignored and is not loaded. The search paths for each plug-in are defined so that platform or machine-class specific modules take precedence over common modules. This architecture permits easy deployment of generic modules and then architecture-specific modules that can overload generic behavior on appropriate platforms.

The topology library guarantees that all modules specified in topology map files are loaded when a snapshot is taken. Other modules can be loaded as "helper" modules to further process new or existing resource instances. For example, the platform-specific `hostbridge` plug-in module responsible for enumerating Host Bridge FMRIs on sun4u loads in the PCI bus enumerator module to discover and enumerate PCI FMRIs. The `/usr/platform/sun4u/lib/fm/topo/plugin/hostbridge.so` file is loaded automatically by the library while processing the platform `hc-scheme` topology map file according to the following XML rule:

```
<range name='hostbridge' min='0' max='254'>
       <enum-method name='hostbridge' version='1' />
     </range>
```

The `hostbridge` module subsequently loads in `/usr/lib/fm/topo/hc/pcubus.so` to complete the discovery and enumeration of FMRIs present in the PCI subsystem.

Modules can also be loaded to provide programmatic methods applicable to a given type of FMRI.

When a snapshot is no longer need, all modules are unloaded and any memory associated with resource instances and properties is deallocated.

9.7.4.1 `_topo_init()`

`void _topo_init(topo_mod_t *`*handle*`)`

The `_topo_init()` function is called once when the module is initialized, and is required to be implemented by all modules. This function receives as a parameter a pointer to an opaque handle, which is associated with this particular instance of a loaded module. The handle initially is unregistered in that it has no configuration information associated with it. The `_topo_init()` routine is responsible for performing one-time initialization of the module and registering the handle with the topology library using `topo_mod_register()`, described below. If the handle has been registered when `_topo_init()` returns, then module initialization is considered successful. If `_topo_init()` returns without registering the handle (or, after registering, the handle is then subsequently unregistered), the module initialization is considered to be unsuccessful and the module is unloaded as the result of failing to register.

The same handle is passed to all of the other module entry points. Handles are associated with particular modules, and any attempt to pass a handle that is not owned by the caller or is not valid causes an error.

9.7.4.2 `_topo_fini()`

`void _topo_fini(topo_mod_t *`*handle*`)`

The _topo_fini() function is an optional entry point that a module can implement in order to provide any one-time cleanup activities prior to unloading. It is not necessary to unregister the handle associated with the module in _topo_fini(), although the module is free to do so. If the handle is not unregistered in _topo_fini(), it is unregistered automatically by the library once the function returns. If _topo_init() did not register a handle, or if an error occurred during _topo_init(), _topo_fini() will not be called.

9.7.5 Handle Registration

The _topo_init() function receives an opaque pointer known as a handle associated with the module instance, and is required to register this handle with the topology library in order to describe the module's metadata to the library and trigger processing of the module's enumeration and other methods. The handle is passed as the first parameter to all module entry points. The functions in this section can be used by the modules to register and unregister handles, and store and retrieve module-specific data.

9.7.5.1 topo_mod_register()

```
int topo_mod_register(topo_mod_t *handle, int version,
    const topo_modinfo_t *info)
```

Register the specified handle with libtopo and complete module initialization by processing any configuration file that is present for this module. The topo_mod_register() function returns an integer that indicates whether it succeeded (zero) or failed (non-zero), permitting modules to deallocate any memory that was allocated in _topo_init() prior to calling topo_mod_register().

The module is required to specify the version of the application programming interface that it compiled against (using the constant TOPO_API_VERSION provided in the header file) and a pointer to a structure that describes the module's entry points and metadata, defined as follows:

```
typedef struct topo_mod_info {
        const char *tmi_desc;           /* module description */
        const char *tmi_scheme;         /* enumeration scheme type  */
        topo_version_t tmi_version;     /* module version string */
        topo_mod_ops_t *tmi_ops;        /* module ops vector for module */
} topo_modinfo_t;
```

The tmi_desc member should point to an ASCII string that briefly describes the module's purpose (for example, "Chip Enumerator for i86pc"). The tmi_version member should contain the module version string. Modules are expected to maintain their own version numbers.

The tmi_ops member must point to a valid topo_mod_ops_t structure, defined as follows:

```
typedef int topo_enum_f(topo_mod_t *, tnode_t *, const char *, topo_instance_t,
    topo_instance_t, void *);
typedef void topo_release_f(topo_mod_t *, tnode_t *);

typedef struct topo_mod_ops {
        topo_enum_f *tmop_enum;          /* enumerator function */
        topo_release_f *tmop_release;    /* enumeration release function */
} topo_mod_ops_t;
```

These members in turn should be initialized to the functions in the module that implement the corresponding entry points. The semantics of each entry point are discussed in the next section. All of the entry points are optional. Entry points that are not implemented can be defined either as an empty routine or as a NULL pointer.

9.7.5.2 topo_mod_unregister()

`void topo_mod_unregister(topo_mod_t *`*handle*`)`

The `topo_mod_unregister()` function unregisters a handle that was previously registered with the topology library using `topo_mod_register()`. When a handle is unregistered, all of its topology nodes are removed from the snapshot, no further entry points are called, and the module can be unloaded. Modules do not typically need to call `topo_mod_unregister()` because `topo_mod_unregister()` will be called automatically following the completion of `_topo_fin()`. The `topo_mod_unregister()` function is provided in case a module needs to cause a module load failure in `_topo_init()` after `topo_mod_register()` succeeds. The `topo_mod_unregister()` function also is provided for symmetry. If the specified handle is not registered, `topo_mod_unregister()` has no effect.

9.7.5.3 topo_mod_setspecific()

`void topo_mod_setspecific(topo_mod_t *`*handle*`, void *`*data*`)`

The `topo_mod_setspecific()` function can be used to associate a data pointer with the specified handle for the duration of the module's lifetime. This pointer can be subsequently retrieved using `topo_mod_getspecific()`. If the pointer is used to refer to dynamically allocated memory, the module is responsible for freeing this memory in its `_topo_fini()` entry point before the module is unloaded.

9.7.5.4 topo_mod_getspecific()

`void *topo_mod_getspecific(topo_mod_t *`*handle*`)`

Return the handle-specific data pointer that was previously associated with *handle* using `topo_mod_setspecific()`. If the module has never called `topo_mod_setspecific()` on this *handle*, `topo_mod_getspecific()` returns NULL.

9.7.6 Entry Points

Modules are expected to implement functions that correspond to one or more of the module entry points described in the `topo_mod_ops_t` structure. The `topo_mod_ops_t` structure is described in `topo_mod_register()` above. This section describes the syntax and semantics for each entry point.

9.7.6.1 `tmi_enum()`

```
int <module>_enum(topo_mod_t *handle, tnode_t *node, topo_instance_t min,
    topo_instance max, void *data);
```

The `tmi_enum()` entry point is called to start enumeration of resource instances inclusively within *min* and *max* for FMRIs that are organizationally related to the resource represented by the opaque node *node*. The node handle can be used to bind new resource instances (nodes) to the topology snapshot and access relevant property information.

9.7.6.2 `tmi_release()`

```
int <module>_release(topo_mod_t *handle, tnode_t *node)
```

The `tmi_release()` entry point is called by the topology library when a node is released. The module should use this opportunity to release any node private data previously allocated during enumeration or method operations.

9.7.7 Memory Allocation

The topology library provides a set of functions for dynamic memory allocation and string duplication for the convenience of its client modules. These functions are provided to ease programming and testing of modules. The Solaris reference implementation uses `libumem.so.1` to implement the memory allocation routines, offering developers additional debugging facilities. See "Topology Library Debugging" in Chapter 13, "Debugging," for more information.

9.7.7.1 `topo_mod_alloc()`

```
void *topo_mod_alloc(topo_mod_t *mod, size_t size)
```

Allocate *size* bytes of memory and return the address of the start of this memory. The memory is aligned to permit storage of the largest C data structure, and no guarantees are made about its initial contents. If the library is unable to allocate the memory, `topo_mod_alloc()` returns NULL.

9.7.7.2 topo_mod_zalloc()

```
void *topo_mod_zalloc(topo_mod_t *mod, size_t size)
```

Allocate *size* bytes of memory as if by `topo_mod_alloc()`, and then fill the contents of the allocation with zeroes if the result is not a NULL pointer.

9.7.7.3 topo_mod_free()

```
void topo_mod_free(topo_mod_t *mod, void *data, size_t size)
```

Deallocate the *size* bytes referred to by the data pointer, which should have been obtained using a previous call to `topo_mod_alloc()` or `topo_mod_zalloc()`. The *size* must exactly match the size used to allocate the buffer. It is legal to free a NULL data pointer by specifying a size of zero. It is not legal to perform a partial or duplicate free.

9.7.7.4 topo_mod_strdup()

```
char *topo_mod_strdup(topo_mod_t *mod, const char *string)
```

Duplicate the specified *string* by allocating memory as if by `topo_mod_alloc()` for the length of the string plus an additional byte for the trailing `\0`, and then copy the source string into this newly allocated memory. The address of the new string is returned. If the library is unable to allocate the memory, `topo_mod_strdup()` returns NULL.

9.7.7.5 topo_mod_strfree()

```
void topo_mod_strfree(topo_mod_t *mod, char *string)
```

Free the memory associated with *string*, where *string* must refer to the result of a previous call to `topo_mod_strdup()`. If *string* is NULL, this function always succeeds and has no effect.

Note – The Solaris reference implementation of the fault manager uses `libumem.so.1` to perform memory allocation, and therefore internally computes the size of the string to be freed by applying `strlen()` to it. Therefore, callers of `topo_mod_strfree()` should take care not to insert additional `\0` characters into the string or to remove the trailing `\0`.

9.7.7.6 topo_mod_nvalloc()

```
int topo_mod_nvalloc(topo_mod_t *mod, nvlist_t **nvlp, uint_t nvflag)
```

The `topo_mod_nvalloc()` function allocates a new name-value pair list and updates *nvlp* to point to the handle. The *nvflag* argument specifies *nvlist* properties to remain persistent across packing, unpacking, and duplication. If NV_UNIQUE_NAME was specified for *nvflag*, existing nvpairs with matching names are removed before the new nvpair is added. If

NV_UNIQUE_NAME_TYPE was specified for *nvflag*, existing nvpairs with matching names and data types are removed before the new nvpair is added. See `nvlist_alloc`(3NVPAIR) for more information.

Module writers should not use `nvlist_alloc`(3NVPAIR) because the topology library installs its own name-value pair list constructors and manages the memory on behalf of the module.

9.7.7.7 topo_mod_nvdup()

```
int topo_mod_nvdup(topo_mod_t *, nvlist_t *nvl, nvlist_t **nvlp)
```

Copies *nvl* and updates *nvlp* to point to the copy. Memory allocated for the name-value pair list copy is allocated as if by `topo_mod_nvalloc()`.

9.7.8 Debugging Support

The functions described in this section set and clear debug mode and manipulate error messages.

9.7.8.1 topo_mod_setdebug()

```
void topo_mod_setdebug(topo_mod_t *mod)
```

Turns on debug messaging for the module specified by the opaque *mod* handle. Subsequent messages generated by `topo_mod_dprintf()` are sent to the output device specified by the caller of `topo_debug_set()` for the snapshot for which this module is loaded. For more information, see `topo_debug_set()` above.

9.7.8.2 topo_mod_clrdebug()

```
void topo_mod_clrdebug(topo_mod_t *)
```

Turns off debug messaging for the module specified by the opaque *mod* handle.

9.7.8.3 topo_mod_dprintf()

```
void topo_mod_dprintf(topo_mod_t *mod, const char *format, ...)
```

Generate a debug message using the specified *format*. The format and any additional arguments are formatted using `snprintf`(3C). The message is either printed to `stderr` or `stdout` according to current debug settings set by `topo_debug_set()`.

9.7.8.4 topo_mod_errmsg()

```
const char *topo_mod_errmsg(topo_mod_t *mod)
```

Returns a string associated with the current error associated with the module's opaque handle, *mod*. The topology library manages the memory associated with the message string.

9.7.8.5 topo_mod_errno()

```
int topo_mod_errno(topo_mod_t *mod)
```

Returns the current error code associated with the module's opaque handle, *mod*.

9.7.8.6 topo_mod_seterrno()

```
int topo_mod_seterrno(topo_mod_t *mod, int error)
```

Sets the module's current error code to *error*. The following error codes are supported:

EMOD_NOMEM	Module memory limit exceeded
EMOD_PARTIAL_ENUM	Module completed partial enumeration
EMOD_METHOD_INVAL	Method arguments invalid
EMOD_METHOD_NOTSUP	Method not supported
EMOD_FMRI_NVL	FMRI nvlist allocation failure
EMOD_FMRI_VERSION	Invalid FMRI scheme version
EMOD_FMRI_MALFORM	Malformed FMRI
EMOD_VER_ABI	Registered with invalid ABI version
EMOD_VER_OLD	Attempt to load obsolete module
EMOD_VER_NEW	Attempt to load a newer module
EMOD_NVL_INVAL	Invalid nvlist
EMOD_NONCANON	Non-canonical component name requested
EMOD_MOD_NOENT	Module lookup failed
EMOD_UKNOWN_ENUM	Unknown enumeration error

9.7.9 Enumeration and Module Loading

9.7.9.1 topo_mod_enumerate()

```
int topo_mod_enumerate(topo_mod_t *mod, tnode_t *node, const char *enum,
    const char *name, topo_instance_t min, topo_instance_t max, void *data)
```

A range of potential topology nodes is created between *min* and *max* and named by *name* and linked to *node*. The enumeration method exported by the *enum* plug-in is called upon to enumerate the topology within that range. The enumerator can enumerate the topology below that range but is responsible for freeing any resources it allocates during processing. Enumerator private data can be passed to the enumerator but is not interpreted by `libtopo`. The *enum* plug-in must have been previously loaded by a call to `topo_mod_load()`.

If a topology is successfully enumerated, `topo_mod_enunmerate()` returns 0. Otherwise -1 is returned and an error code is set in the opaque *mod* handle. Valid error codes include:

`EMOD_MOD_NOENT` Module lookup failed

`EMOD_UKNOWN_ENUM` Unknown enumeration error

Valid error codes also include the valid list of error codes defined for the module entry point *module*`_enum()`.

9.7.9.2 `topo_mod_enummap()`

```
int topo_mod_enummap(topo_mod_t *mod, tnode_t *node, const char *name,
    const char *scheme)
```

Searches the following standard paths for a topology map XML file:

- *rootdir*`/usr/platform/`*platform*`/lib/fm/topo/maps/`*name*`.xml` (where *platform* is `uname -i` by default)
- *rootdir*`/usr/platform/`*machine*`/lib/fm/topo/maps/`*name*`.xml` (where *machine* is `uname -m` by default)
- *rootdir*`/usr/lib/fm/topo/`*scheme*`/map/`*name*`.xmls`

The values of *rootdir*, *platform*, and *machine* are established during snapshot initialization in `topo_open()`. If no map files are found in the platform-specific directories, the standard scheme directories are searched in alphabetical order.

If a topology map file is found, `topo_mod_enummap()` parses and initiates enumeration of the topology specified in the map file. The *scheme* must match the topology FMRI scheme specification in the map file and the scheme already established for *node*. The topology map file must follow the topology DTD (document type definition) described in "Topology Map Files" below. The topology enumerated via the map file is linked to *node*. Any resources allocated during the enumeration process are the responsibility of the enumerators called upon to perform the enumeration.

If the topology map file is successfully opened and parsed without enumeration error, `topo_mod_enummap()` returns 0. Otherwise, -1 is returned and an error code is set in the opaque *mod* handle. Valid error codes are:

`ETOPO_NOMEM` Memory limit exceeded

ETOPO_MOD_XRD	Unable to read topology map file
ETOPO_MOD_XENUM	Unable to enumerate from a topology map file
ETOPO_MOD_NOENT	Module path invalid

9.7.9.3 topo_mod_load()

```
topo_mod_t *topo_mod_load(topo_mod_t *mod, const char *name,
    topo_version_t version)
```

Searches the following standard path for a shared library denoted by name:

- *rootdir*/usr/platform/*platform*/lib/fm/topo/plugins/*name*.so (where *platform* is uname -i by default)
- *rootdir*/usr/platform/*machine*/lib/fm/topo/plugins/*name*.so (where machine is uname -m by default)
- *rootdir*/usr/lib/fm/topo/*scheme*/plugins/*name*.so

The values of *rootdir*, *platform*, and *machine* are established during snapshot initialization in topo_open(). If no libraries are found in the platform-specific directories, the standard scheme directories are searched in alphabetical order.

If a shared library is found, topo_mod_load() uses dlopen(3C) to load the enumerator plug-in into libtopo. The *version* is checked against the version specified by the module during registration (see topo_mod_register()).

If the shared library plug-in is successfully found and loaded, topo_mod_load() returns 0. Otherwise, -1 is returned and an error code is set in the *mod* handle. Valid error codes are:

ETOPO_MOD_VER	Module version mismatch while loading
ETOPO_MOD_NOENT	Module path invalid

9.7.9.4 topo_mod_unload()

```
void topo_mod_unload(topo_mod_t *mod)
```

Unloads a previously loaded module via dlclose(3C).

9.7.10 Topology Node Management

As part of enumerating the system topology, modules are expected to create topology nodes that are specific to their platform or subsystem type. These topology nodes are linked together

by the topology library to form a complete picture of FMRIs. The following sections describe a list of functions used in module node creation, linkage, and destruction. Library error codes are recorded in the module's error code.

9.7.10.1 topo_node_range_create()

```
int topo_node_range_create(topo_mod_t *mod, tnode_t *pnode,
    const char *name, topo_instance_t min, topo_instance_t max)
```

Create a range of potential topology nodes. All nodes within the range share the same name with no more than *max* and no less than *min* nodes in the range. The range is linked to *pnode* in the topology snapshot and no memory for actual nodes is allocated until specific nodes are verified to exist and bound into the topology. The name must be unique to the ranges linked to *pnode*. If the range is successfully created, 0 is returned. Otherwise, -1 is returned.

9.7.10.2 topo_node_range_destroy()

```
void topo_node_range_destroy(tnode_t *pnode, const char *name)
```

Destroy a previously allocated topology node range. The range is unlinked from *pnode* and all memory (including that reserved for nodes within the range) is deallocated.

9.7.10.3 topo_node_bind()

```
tnode_t * topo_node_bind(topo_mod_t *mod, tnode_t *pnode,
    const char *name,  topo_instance_t inst, nvlist_t *fmri)
```

Returns a new opaque node handle and binds it to the topology snapshot. The new node resides in the range of nodes specified by *name* and has the instance number specified by *inst*. A protocol property group is created for the node with the resource property assigned to *fmri*.

On failure, `topo_node_bind()` returns NULL.

9.7.10.4 topo_node_unbind()

```
void topo_node_unbind(tnode_t *node)
```

Removes the node from the node range and effectively unlinks it from the topology snapshot. All memory associated with the node handle is deallocated on return.

9.7.10.5 topo_node_setspecific()

```
void topo_node_setspecific(tnode_t *node, void *data)
```

The `topo_node_setspecific()` function can be used to associate a data pointer with the specified handle for the duration of the node's lifetime. This pointer can be subsequently

retrieved using `topo_node_getspecific()`. If the pointer is used to refer to dynamically allocated memory, the module is responsible for freeing this memory in its `tmi_release()` entry point before the node is unbound.

9.7.10.6 `topo_node_getspecific()`

```
void *topo_node_getspecific(tnode_t *node)
```

Return the node-specific data pointer previously associated with handle using `topo_node_setspecific()`. If the module has never called `topo_node_setspecific()` on this handle, `topo_node_getspecific()` returns NULL.

9.7.11 Property Installation

This section discusses how modules can statically set property groups and property values or install methods for accessing property values. For more information on property group organization and accessing property values, see "Topology Node Properties."

9.7.11.1 `topo_pgroup_create()`

```
int topo_pgroup_create(tnode_t *node, const topo_pgroup_info_t *pinfo, int *err)
```

Create a property group according to the property group information contained in *pinfo* for *node*. The *pinfo* argument is a pointer to a `topo_pgroup_info_t` data structure comprised of the following members:

```
typedef struct topo_pgroup_info {
        const char *tpi_name;           /* property group name */
        topo_stability_t tpi_namestab;  /* stability of group name */
        topo_stability_t tpi_datastab;  /* stability of all property values */
        topo_version_t tpi_version;  /* version of topo_pgroup_info_t */
    } topo_pgroup_info_t;
```

The *tpi_namestab* and *tpi_datastab* values are assigned according to the `attributes`(5) stability levels and can take the following values:

`TOPO_STABILITY_INTERNAL`	Private to `libtopo`
`TOPO_STABILITY_PRIVATE`	Private
`TOPO_STABILITY_OBSOLETE`	Obsolete
`TOPO_STABILITY_EXTERNAL`	Volatile
`TOPO_STABILITY_UNSTABLE`	Uncommitted
`TOPO_STABILITY_EVOLVING`	Uncommitted

TOPO_STABILITY_STABLE	Committed
TOPO_STABILITY_STANDARD	Committed

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is assigned to one of the following error codes:

ETOPO_PROP_DEFD	Static property already defined
ETOPO_NOMEM	Memory limit exceeded

9.7.11.2 topo_pgroup_destroy()

```
void topo_pgroup_destroy(tnode_t *node, const char *pname)
```

Destroy a previously created property grouping named by *pname* and assigned to *node*. All properties in the group are also destroyed. If any one of the properties in this group has been inherited by another node, the memory allocated to the property group and that particular property remains allocated but no longer assigned to *node*.

9.7.11.3 topo_prop_set_int32()

```
int topo_prop_set_int32(tnode_t *node, const char *pgname, const char *pname,
    int flag, int32_t val, int *err)
```

Create and assign property *pname* to *val*. The *flag* value can be set to either TOPO_PROP_IMMUTABLE or TOPO_PROP_MUTABLE. If it is set to TOPO_PROP_MUTABLE, a subsequent call to `topo_prop_set_int32()` changes *val* as specified in the calling parameters. Otherwise, a subsequent call to `topo_prop_set_int32()` yields an error.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is set to one of the following error codes:

ETOPO_PROP_TYPE	Invalid property type
ETOPO_PROP_DEFD	Static property already defined and immutable
ETOPO_PROP_NVL	Malformed property `nvlist`
ETOPO_PROP_NOMEM	Memory limit exceeded during property allocation

9.7.11.4 topo_prop_set_uint32()

```
int topo_prop_set_uint32(tnode_t *node, const char *pgname, const char *pname,
    int flag, uint32_t val, int *err)
```

Create and assign property *pname* to *val*. The *flag* value can be set to either TOPO_PROP_IMMUTABLE or TOPO_PROP_MUTABLE. If it is set to

TOPO_PROP_MUTABLE, a subsequent call to `topo_prop_set_uint32()` changes *val* as specified in the calling parameters. Otherwise, the call yields an error.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is set to one of the following error codes:

`ETOPO_PROP_TYPE`	Invalid property type
`ETOPO_PROP_DEFD`	Static property already defined and immutable
`ETOPO_PROP_NVL`	Malformed property `nvlist`
`ETOPO_PROP_NOMEM`	Memory limit exceeded during property allocation

9.7.11.5 `topo_prop_set_int64()`

```
int topo_prop_set_int64(tnode_t *node, const char *pgname, const char *pname,
    int flag, int64_t val, int *err)
```

Create and assign property *pname* to *val*. The *flag* value can be set to either TOPO_PROP_IMMUTABLE or TOPO_PROP_MUTABLE. If it is set to TOPO_PROP_MUTABLE, a subsequent call to `topo_prop_set_int64()` changes *val* as specified in the calling parameters. Otherwise, the call yields an error.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is set to one of the following error codes:

`ETOPO_PROP_TYPE`	Invalid property type
`ETOPO_PROP_DEFD`	Static property already defined and immutable
`ETOPO_PROP_NVL`	Malformed property `nvlist`
`ETOPO_PROP_NOMEM`	Memory limit exceeded during property allocation

9.7.11.6 `topo_prop_set_uint64()`

```
int topo_prop_set_uint64(tnode_t *node, const char *pgname, const char *pname,
    int flag, uint64_t val, int *err)
```

Create and assign property *pname* to *val*. The *flag* value can be set to either TOPO_PROP_IMMUTABLE or TOPO_PROP_MUTABLE. If it is set to TOPO_PROP_MUTABLE, a subsequent call to `topo_prop_set_uint64()` changes *val* as specified in the calling parameters. Otherwise, the call yields an error.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is set to one of the following error codes:

`ETOPO_PROP_TYPE`	Invalid property type

ETOPO_PROP_DEFD	Static property already defined and immutable
ETOPO_PROP_NVL	Malformed property nvlist
ETOPO_PROP_NOMEM	Memory limit exceeded during property allocation

9.7.11.7 topo_prop_set_string()

```
int topo_prop_set_string(tnode_t *node, const char *pgname, const char *pname,
    int flag, const char *val, int *err)
```

Create and assign property *pname* to *val*. The *flag* value can be set to either TOPO_PROP_IMMUTABLE or TOPO_PROP_MUTABLE. If it is set to TOPO_PROP_MUTABLE, a subsequent call to `topo_prop_set_string()` changes *val* as specified in the calling parameters. Otherwise, the call yields an error.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is set to one of the following error codes:

ETOPO_PROP_TYPE	Invalid property type
ETOPO_PROP_DEFD	Static property already defined and immutable
ETOPO_PROP_NVL	Malformed property nvlist
ETOPO_PROP_NOMEM	Memory limit exceeded during property allocation

9.7.11.8 topo_prop_set_fmri()

```
int topo_prop_set_fmri(tnode_t *node, const char *pgname, const char *pname,
    int flag, const nvlist_t *fmri, int *err)
```

Create and assign property *pname* to *val*. The *flag* value can be set to either TOPO_PROP_IMMUTABLE or TOPO_PROP_MUTABLE. If it is set to TOPO_PROP_MUTABLE, a subsequent call to `topo_prop_set_fmri()` changes *val* as specified in the calling parameters. Otherwise, the call yields an error.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is set to one of the following error codes:

ETOPO_PROP_TYPE	Invalid property type
ETOPO_PROP_DEFD	Static property already defined and immutable
ETOPO_PROP_NVL	Malformed property nvlist
ETOPO_PROP_NOMEM	Memory limit exceeded during property allocation

9.7.11.9 topo_prop_set_int32_array()

```
int topo_prop_set_int32_array(tnode_t *node, const char *pgname,
    const char *pname, int flag, int32_t *val, uint_t nelems, int *err)
```

Create and assign property *pname* to the *val* array of *nelems* number of elements. The *flag* value can be set to either TOPO_PROP_IMMUTABLE or TOPO_PROP_MUTABLE. If it is set to TOPO_PROP_MUTABLE, a subsequent call to `topo_prop_set_int32_array()` changes *val* as specified in the calling parameters. Otherwise, the call yields an error.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is set to one of the following error codes:

`ETOPO_PROP_TYPE`	Invalid property type
`ETOPO_PROP_DEFD`	Static property already defined and immutable
`ETOPO_PROP_NVL`	Malformed property `nvlist`
`ETOPO_PROP_NOMEM`	Memory limit exceeded during property allocation

9.7.11.10 topo_prop_set_uint32_array()

```
int topo_prop_set_uint32_array(tnode_t *node, const char *pgname,
    const char *pname, int flag, uint32_t *val, uint_t nelems, int *err)
```

Create and assign property *pname* to the *val* array of *nelems* number of elements. The *flag* value can be set to either TOPO_PROP_IMMUTABLE or TOPO_PROP_MUTABLE. If it is set to TOPO_PROP_MUTABLE, a subsequent call to `topo_prop_set_uint32_array()` changes *val* as specified in the calling parameters. Otherwise, the call yields an error.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is set to one of the following error codes:

`ETOPO_PROP_TYPE`	Invalid property type
`ETOPO_PROP_DEFD`	Static property already defined and immutable
`ETOPO_PROP_NVL`	Malformed property `nvlist`
`ETOPO_PROP_NOMEM`	Memory limit exceeded during property allocation

9.7.11.11 topo_prop_set_int64_array()

```
int topo_prop_set_int64_array(tnode_t *node, const char *pgname,
    const char *pname, int flag, int64_t *val, uint_t nelems, int *err)
```

Create and assign property *pname* to the *val* array of *nelems* number of elements. The *flag* value can be set to either TOPO_PROP_IMMUTABLE or TOPO_PROP_MUTABLE. If it is set to

TOPO_PROP_MUTABLE, a subsequent call to `topo_prop_set_int64_array()` changes *val* as specified in the calling parameters. Otherwise, the call yields an error.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is set to one of the following error codes:

`ETOPO_PROP_TYPE`	Invalid property type
`ETOPO_PROP_DEFD`	Static property already defined and immutable
`ETOPO_PROP_NVL`	Malformed property `nvlist`
`ETOPO_PROP_NOMEM`	Memory limit exceeded during property allocation

9.7.11.12 `topo_prop_set_uint64_array()`

```
int topo_prop_set_uint64_array(tnode_t *node, const char *pgname,
    const char *pname, int flag, uint64_t *val, uint_t nelems, int *err)
```

Create and assign property *pname* to the *val* array of *nelems* number of elements. The *flag* value can be set to either TOPO_PROP_IMMUTABLE or TOPO_PROP_MUTABLE. If it is set to TOPO_PROP_MUTABLE, a subsequent call to `topo_prop_set_uint64_array()` changes *val* as specified in the calling parameters. Otherwise, the call yields an error.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is set to one of the following error codes:

`ETOPO_PROP_TYPE`	Invalid property type
`ETOPO_PROP_DEFD`	Static property already defined and immutable
`ETOPO_PROP_NVL`	Malformed property `nvlist`
`ETOPO_PROP_NOMEM`	Memory limit exceeded during property allocation

9.7.11.13 `topo_prop_set_string_array()`

```
int topo_prop_set_string_array(tnode_t *node, const char *pgname,
    const char *pname, int flag, const char **val, uint_t nelems, int *err)
```

Create and assign property *pname* to the *val* array of *nelems* number of elements. The *flag* value can be set to either TOPO_PROP_IMMUTABLE or TOPO_PROP_MUTABLE. If it is set to TOPO_PROP_MUTABLE, a subsequent call to `topo_prop_set_string_array()` changes *val* as specified in the calling parameters. Otherwise, the call yields an error.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is set to one of the following error codes:

`ETOPO_PROP_TYPE`	Invalid property type

ETOPO_PROP_DEFD	Static property already defined and immutable
ETOPO_PROP_NVL	Malformed property nvlist
ETOPO_PROP_NOMEM	Memory limit exceeded during property allocation

9.7.11.14 topo_prop_set_fmri_array()

```
int topo_prop_set_fmri_array(tnode_t *node, const char *pgname,
    const char *pname, int flag, const nvlist_t **val, uint_t nelems, int *err)
```

Create and assign property *pname* to the *val* array of *nelems* number of elements. The *flag* value can be set to either TOPO_PROP_IMMUTABLE or TOPO_PROP_MUTABLE. If it is set to TOPO_PROP_MUTABLE, a subsequent call to `topo_prop_set_fmri_array()` changes *val* as specified in the calling parameters. Otherwise, the call yields an error.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is set to one of the following error codes:

ETOPO_PROP_TYPE	Invalid property type
ETOPO_PROP_DEFD	Static property already defined and immutable
ETOPO_PROP_NVL	Malformed property nvlist
ETOPO_PROP_NOMEM	Memory limit exceeded during property allocation

9.7.11.15 topo_prop_inherit()

```
int topo_prop_inherit(tnode_t *node, const char *pgname, const char *name,
    int *err)
```

Inherit from the immediate parent of *node* the group-property combination specified by *pgname* and *name*. If successfully found, the property is reference counted and will not be de-allocated until all referring nodes have been destroyed. Properties declared as TOPO_PROP_MUTABLE cannot be inherited.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is assigned to one of the following error codes:

ETOPO_PROP_NOENT	Property does not exist at parent node
ETOPO_PROP_NOINHERIT	Property is declared as TOPO_PROP_MUTABLE
ETOPO_NOMEM	Memory limit exceeded during property allocation

9.7.11.16 topo_prop_method_register()

```
int topo_prop_method_register(tnode_t *node, const char *pgname,
    const char *pname, topo_type_t ptype, const char *mname,
    const nvlist_t *args, int *err)
```

Register a method to dynamically obtain property information. A property named by *pname* is reserved in the group *pgname*, and a binding to a previously registered method named *mname* is created. Subsequent property lookups are performed by the type specific `topo_prop_get_*()` routines described in "Topology Node Properties" according to *ptype*. When one of these routines is called, a `topo_method_f()` function associated with *mname* is invoked.

```
typedef int topo_method_f(topo_mod_t *mod, tnode_t *node,
    topo_version_t version, nvlist_t *input_args, nvlist_t **output_args);
```

The *mod* argument represents the module that registered the method for *node*, and *version* is set to the desired method version. The *args* name-value pair list registered with the property-method binding is passed to `topo_method_f()` as a name-value pair list in *input_args*. If *args* is NULL, *input_args* will be NULL.

The `topo_method_f()` function is expected to determine the property tuple (name, type, value) and return it as a name-value pair list named as "property" in **output_args*. The property name-value pair list must contain three sub-components:

`"property-name"`	`char *`
`"property-type"`	`uint32_t`
`"property-value"`	`nvlist_t`

The following definitions are provided to simplify coding:

`TOPO_PROP_ARGS`	`"args"`
`TOPO_PROP_VAL`	`"property"`
`TOPO_PROP_VAL_NAME`	`"property-name"`
`TOPO_PROP_VAL_TYPE`	`"property-type"`
`TOPO_PROP_VAL_VAL`	`"property-value"`

TOPO_PROP_VAL_VAL

See `topo_method_register()` below for more information on module method registration and method function invocation.

Upon success, 0 is returned. Otherwise, -1 is returned and **err* is updated to contain one of the following error codes:

`ETOPO_PROP_NOMEM`	Memory limit exceeded during property allocation

ETOPO_METHOD_DEFD	Method *op* already bound for this property

9.7.12 Method Registration

Modules are permitted to register method functions to dynamically obtain property tuples or to invoke a node specific service by consumers of the topology snapshot.

9.7.12.1 topo_method_register()

```
int topo_method_register(topo_mod_t *mod, tnode_t *node,
    const topo_method_t *mp)
```

Register a method service function for *node* according to *mp*. The *mp* is a pointer to a topo_method_t data structure:

```
typedef struct topo_method {
        const char *tm_name;                    /* Method name */
        const char *tm_desc;                    /* Method description */
        const topo_version_t tm_version;        /* Method version */
        const topo_stability_t tm_stability;    /* Attribute(5) stability */
        topo_method_f *tm_func;                 /* Method function */
} topo_method_t;
```

The *tm_name* is used to register and look up the method function, tm_func(). The version of the method function and its parameter list are specified in *tm_version*. The version is checked by the library before invoking the function when used via topo_method_invoke(). If the method is used to obtain property values, the parameter list is defined as described in topo_prop_method_register() and the version is set to 0.

The *tm_stability* defines the attributes(5) stability levels as:

TOPO_STABILITY_INTERNAL	Private to libtopo
TOPO_STABILITY_PRIVATE	Private
TOPO_STABILITY_OBSOLETE	Obsolete
TOPO_STABILITY_EXTERNAL	Volatile
TOPO_STABILITY_UNSTABLE	Uncommitted
TOPO_STABILITY_EVOLVING	Uncommitted
TOPO_STABILITY_STABLE	Committed
TOPO_STABILITY_STANDARD	Committed

The method function, *tm_func*, is defined as:

```
typedef int topo_method_f(topo_mod_t *mod, tnode_t *node,
    topo_version_t version, nvlist_t *input_args, nvlist_t **output_args);
```

The *mod* argument represents the module that registered the method for *node*. The *version* argument is set to the version used in `topo_method_invoke()` or to 0 if called to acquire node properties. The *input_args* argument is not interpreted by the library and is passed from the caller of `topo_method_invoke()` or as described in `topo_prop_method_register()`. The **output_args* argument contains the function's output argument list as a name-value pair list or a TOPO_PROP_VAL name-value pair list as described in `topo_prop_method_register()`.

Upon successful return, *tm_func* returns 0. Otherwise, -1 is returned and the opaque module handle is updated to return one of the following error codes:

`ETOPO_METHOD_VERNEW`	Unknown method failure
`ETOPO_METHOD_VEROLD`	Caller is compiled to use obsolete method
`ETOPO_METHOD_FAIL`	Caller is compiled to use obsolete method

Upon successful return from `topo_method_register()`, 0 is returned. Otherwise, -1 is returned and the opaque module handle is updated to contain one of the following error codes:

`ETOPO_METHOD_INVAL`	Invalid method registration
`ETOPO_METHOD_NOMEM`	Memory limit exceeded during method register

9.7.12.2 `topo_method_unregister()`

```
void topo_method_unregister(topo_mod_t *mod, tnode_t *node, const char *name)
```

Unregister the previously registered method named by *name* for *node*.

9.7.12.3 `topo_method_unregister_all()`

```
void topo_method_unregister_all(topo_mod_t *mod, tnode_t *node)
```

Unregister all methods for *node*. This interface is useful during calls to the *module*`_release()` entry point.

9.7.13 Module Convenience Functions

This section describes a set of convenience functions to ease programming tasks for module writers.

9.7.13.1 `topo_mod_devinfo()`

```
di_node_t topo_mod_devinfo(topo_mod_t *mod)
```

Returns a root handle to a snapshot of the kernel device tree. See `libdevinfo`(3LIB) for more information and error conditions.

9.7.13.2 topo_mod_prominfo()

```
di_prom_handle_t topo_mod_prominfo(topo_mod_t *mod)
```

Returns a root handle to a snapshot of the PROM device tree. See `libdevinfo`(3LIB) for more information and error conditions.

9.7.13.3 topo_mod_auth()

```
nvlist_t * topo_mod_auth(topo_mod_t *mod, tnode_t *node)
```

Returns the FMRI authority for node as a name-value pair list. The authority elements are strings specified by the FMA Event Protocol and FMRI specification and in `/usr/include/sys/fm/fm.h` as:

```
FM_FMRI_AUTH_PRODUCT    "product"

FM_FMRI_AUTH_CHASSIS    "chassis"

FM_FMRI_AUTH_SERVER     "server"
```

If no authority is associated to node, or if memory limits have been exceeded, `topo_mod_auth()` returns NULL.

9.7.13.4 topo_mod_cpufmri()

```
nvlist_t * topo_mod_cpufmri(topo_mod_t *mod, int version,
    uint32_t cpu_id, uint8_t cpumask, const char *serial)
```

Returns a FMRI as a name-value pair list according to the FMA Event Protocol and FMRI specification for the cpu scheme. The `topo_mod_cpufmri()` function uses *cpu_id*, *cpumask*, and *serial* to construct the FMRI according to the version of scheme specification. The *version* should be set to FM_CPU_SCHEME_VERSION as defined in `/usr/include/sys/fm/fm.h`.

The `topo_mod_cpufmri()` function returns NULL if an invalid *cpu_id* is passed in, memory limits have been exceeded, or a failure occurs during the name-value pair list construction.

9.7.13.5 topo_mod_devfmri()

```
nvlist_t * topo_mod_devfmri(topo_mod_t *mod, int version,
    const char *dev_path,  const char *devid)
```

Returns a FMRI as a name-value pair list according to the FMA Event Protocol and FMRI specification for the dev scheme. The `topo_mod_devfmri()` function uses *dev_path*, and *devid*

to construct the FMRI according to the version of scheme specification. The *version* should be set to FM_DEV_SCHEME_VERSION as defined in `/usr/include/sys/fm/fm.h`.

The `topo_mod_devfmri()` function returns NULL if an invalid *cpu_id* is passed in, memory limits have been exceeded, or a failure occurs during the name-value pair list construction.

9.7.13.6 topo_mod_hcfmri()

```
nvlist_t *topo_mod_hcfmri(topo_mod_t *mod, tnode_t *pnode,
    int version, const char *name, topo_instance_t inst, nvlist_t *hc_specific,
nvlist_t *auth, const char *part, const char *rev, const char *serial)
```

Returns a FMRI as a name-value pair list according to the FMA Event Protocol and FMRI specification for the hc scheme. The FMRI associated with *pnode*, is used to create a hardware component path terminating with *name=inst*. Optional members, *hc_specific*, *auth*, *part*, *rev*, and *serial* may be set to NULL. The *version* should be set to FM_HC_SCHEME_VERSION as defined in `/usr/include/sys/fm/fm.h`.

The `topo_mod_hcfmri()` function returns NULL if an invalid version or parent node is specified or a failure occurs during the name-value pair list construction.

9.7.13.7 topo_mod_memfmri()

```
nvlist_t * topo_mod_memfmri(topo_mod_t *mod, int version,
    uint64_t pa, uint64_t offset, const char *unum, int flags)
```

Returns a FMRI as a name-value pair list according to the FMA Event Protocol and FMRI specification for the mem scheme. The `topo_mod_memfmri()` function uses *unum* to construct the FMRI according the scheme specification. Optional members *pa* and *offset* are considered valid if *flags* is the logical OR of TOPO_MEMFMRI_PA and TOPO_MEMFMRI_OFFSET.

The `topo_mod_memfmri()` function returns NULL if a failure occurs during the name-value pair list construction.

9.7.13.8 topo_mod_nvl2str()

```
int topo_mod_nvl2str(topo_mod_t *mod, nvlist_t *fmri, char **buf)
```

Convert the specified *fmri* into its string representation and place the result in *buf*. This function returns 0 to indicate success or -1 to indicate an error.

The caller is responsible for deallocating the memory associated with *buf* by calling `topo_mod_strfree()`.

9.7.13.9 topo_mod_str2nvl()

```
int topo_mod_str2nvl(topo_mod_t *mod, const char *fmristr, nvlist_t **fmri)
```

Convert the specified FMRI string *fmristr* to its name-value pair list representation. On success, *fmri* is updated with a pointer to a newly allocated name-value pair list. This function returns 0 to indicate success or -1 to indicate an error.

The caller is responsible for deallocating the memory associated with **fmri* by calling `nvlist_free()` when the name-value pair list is no longer needed.

9.8 Topology Map Files

The topology library utilizes an XML-based file format to marshal the description of a FMRI topology based upon scheme type. This file is known as a *topology map*. The primary form of a topology map is an inventory of FMRIs that are provided for a system of hardware for a subsystem, product, or platform.

The DTD (document type definition) that describes the topology map is provided at `/usr/share/lib/xml/dtd/topology.dtd.1`.

A complete topology description consists of the following:

- A set of topology node ranges that identify resources with the same ancestry and name
- A set of topology node instances within a range
- A set of protocol properties that identify a node instance FMRI resource, and optionally its FRU, ASRU, and label
- A set of authority properties that identify a node instance product, chassis, and serial identification
- A set of properties that identify each node instance
- An enumeration method for creating node instances within a topology range
- A set of methods and arguments for accessing properties

The DTD for the topology map provides markup to define each of these aspects of a topology. The attributes and tags are fully described in the commented DTD. The topologies supplied with the operating system provide examples of correctly formed topology maps.

Topology maps are stored and loaded by the library from the following directory locations:

- *rootdir*`/usr/platform/`*machine*`/lib/fm/topo/maps/`*machine*`/`*platform*`-`*scheme*`-topology.xml`
  - The value of *machine* is `uname -m` by default.
  - The value of *platform* is `uname -i` by default, or it is the product name obtained from `smbios`(1M).

- *rootdir*`/usr/lib/fm/topo/map/`*scheme*`-topology.xml`
  - The value of *machine* is `uname -m` by default.

By default, the topology library searches for a platform-specific or product-specific topology map before searching in `rootdir/usr/lib/fm/topo/map`. If no topology map is found is either location, the topology snapshot will be empty.

Subsystem-specific topology map files can be loaded via enumerator plug-in calls to `topo_mod_enummap()` or by `propmap` specifications in another topology map.

◆ ◆ ◆ CHAPTER 10

fminject Utility

This chapter describes the `fminject` error event injector that can be used to test and debug the fault manager and its clients. The error event injector allows developers and test groups to create files that describe a set of error event name-value pair structures to create and send to the fault manager, and can also be used to replay events from an existing `errlog` file. On Solaris systems, the `fminject` command is available at `/usr/lib/fm/fmd/fminject`. `fminject` input files can also be used with the fault manager simulator, `fmsim`, described in Chapter 11, "`fmsim` Utility."

The `fminject` utility is an error *event* injector, not an error injector. That is, it only creates FMA error events, not the underlying errors that correspond to them. Therefore, `fminject` does not exercise or test the underlying error handling code or the code that creates and publishes error events into an event transport. `fminject` is complementary to error injector utilities because it can allow developers of diagnosis software and error handling code to make progress simultaneously when developing new FMA functionality, and can allow developers to more easily debug diagnosis software once it is written.

10.1 Options

The `fminject` utility accepts the following command-line options:

`fminject` `[-nqv] [-c` *channel*`] [`*file*`]`

`-c` Publish events on the specified *channel* instead of the default SysEvent channel, `FM_ERROR_CHAN`.

`-n` Compile input but do not publish any events. This option can be used to syntactically check an input file without actually sending any events to a fault manager.

`-q` Set quiet mode. If the `-q` option is present, `fminject` will not report any status to `stdout` as events are published.

`-v` Set verbose mode. If the `-v` option is present, `fminject` will display verbose information about events as they are published.

If a *file* is specified on the command-line, events are read from the specified input file. If the file is a fault manager log file, events are replayed verbatim from the log with appropriate time delays inserted in between each event, corresponding to the event times recorded in the log. If the file is not a fault manager log, it is assumed to be a text file containing one or more event descriptions and publication commands using the syntax described in the next section. If no *file* is specified on the command-line, `fminject` reads statements from `stdin`.

10.2 Syntax

`fminject` utility accepts a series of declarations and commands delimited by semicolons (`;`). Whitespace between tokens is ignored, and comments can be enclosed in `/*` and `*/` as in C. A typical input file declares one or more FMRIs and event specifications, and then executes one or more commands to publish instances of these events, as shown in the following example:

```
evdef ereport.cpu.usii.ue {
    enum type { persistent, sticky, intermittent };
    int8_t int8[4];
    int16_t int16;
    string str;
    string strarr[];
};

event ereport.cpu.usii.ue ue_1 = {
    persistent,
    [ 1, 2, 3, 4],
    0xfff,
    "hello",
    [ "hello", "there", "a", "b" ]
};

ue_1;
```

Strings are enclosed in double quotes (`" "`). Integers can be specified in decimal (the default), octal (prefixed with a leading `0`), or hexadecimal (prefixed with a leading `0x` or `0X`).

10.2.1 Event Class Definitions

Event classes are introduced using the `evdef` keyword in a declaration of the form:

`evdef` *class* `{` *member-list* `};`

where *class* is a dot-delimited string representing the event class, such as `ereport.cpu.usii.ue`. The members are defined using a semicolon-delimited list of type and

member names that define the name-value pair list to be constructed for each event of this class, resembling a C `struct` declaration. The member types that can be used are as follows:

`int8_t`	Signed 8–bit integer.
`int16_t`	Signed 16–bit integer.
`int32_t`	Signed 32–bit integer.
`int64_t`	Signed 64–bit integer.
`uint8_t`	Unsigned 8–bit integer.
`uint16_t`	Unsigned 16–bit integer.
`uint32_t`	Unsigned 32–bit integer.
`uint64_t`	Unsigned 64–bit integer.
`boolean`	Boolean value.
`string`	Variable-length string.
`enum` *enum-name* `{` *enum-list* `}`	Enumerated type consisting of the comma-separated list of enumerators specified in `{ }` braces. Unlike C enumerators, name-value pair list enumerators may not be assigned specific integer values.
`event` *event-name*	Embedded name-value pair list described by a previous `evdef` declaration.
`fmri` *fmri-name*	Embedded name-value pair list representing an FMRI described by a previous `fmridef` declaration, as described below.
`auth` *auth-name*	Embedded name-value pair list representing an FMRI authority described by a previous `authdef` declaration, as described below.

Members may also be declared with an optional `[]` or `[`*n*`]` for some integer value *n* indicating that the member type is to be a variable-length or fixed-size array of the base data type.

The `class` member of the event is defined automatically for each new event and set to the appropriate value for each subsequent event declaration. If an `ena` member is not defined for an event class, it will be added to the event definition and ENA values will be automatically generated for each event that is published. The following example illustrates a declaration equivalent to the actual definition for the PCI `ereport.io.pci.sec-ma` event:

```
fmridef dev_t {
    uint8_t version;
    string scheme;
```

```
    string device-path;
};

evdef ereport.io.pci.sec-ma {
    fmri dev_t detector;
    uint64_t pci-sec-status;
    uint64_t pci-bdg-ctrl;
};
```

10.2.2 FMRI and Authority Definitions

FMRI and FMRI authority classes can be defined using `fmridef` and `authdef` declarations similar to the following:

`authdef` *auth-name* `{` *member-list* `};`
`fmridef` *fmri-name* `{` *member-list* `};`

Similar to `evdef` declarations, each `fmridef` and `authdef` member list consists of a series of member types and member names that describe the form of the name-value pair lists of the corresponding type.

10.2.3 Event Declarations

Once an event class is defined, events of the class can be declared in order to assign values to each member of the name-value pair list described by the class definition. Event declarations assign a unique identifier name to the event that is used in subsequent statements to publish the event, and resemble C structure initialization statements. Event declarations resemble the following example:

`event` *class name* `{` *value-list* `};`

The following example declaration uses the earlier definition of the event class `ereport.io.pci.sec-ma` to define an event `ma1` that can be published to the event transport:

```
event ereport.io.pci.sec-ma ma1 = {
    { 0, "dev", "/ssm@0,0/pci@1b,700000/pci@2" },      /* detector */
    0x2280,                                            /* pci-sec-status */
    0x23                                               /* pci-bdg-ctrl */
};
```

10.2.4 Event Statements

Events are published to the event transport using a statement resembling one of the following:

```
event-name ;
randomize { probability-list } ;
repeat count event-name ;
repeat count randomize { probability-list } ;
```

The simplest event statement is the name of a previously declared event. For example, the statement "`ma1;`" would publish the `ereport.io.pci.sec-ma` event declared as `ma1`. For brevity, the `repeat` statement can be used to publish an event *count* times. The `randomize` statement can be used to select an event at random from a *probability-list*. The following example shows a probability list that selects event `ma1` 75% of the time and another event `ma2` 25% of the time:

```
randomize {
    { ma1, 75 },
    { ma2, 25 }
};
```

10.2.5 Control Statements

`fminject` supports the following control statements that can be used to control the spacing of events with respect to time:

`addhrtime` *hrtime*`;` — Send a control event to the fault manager indicating that the simulated clock should advance by the specified *hrtime*, representing an offset from the current time in nanoseconds. The time can be specified using any of the time suffixes used in fault manager configuration files (for example, "`1d`" for one day's worth of nanoseconds). Refer to "2.7.3 setprop" on page 35 for a list of time suffixes.

`endhrtime;` — Send a control event to the fault manager indicating that the simulated clock should advance to the simulated apocalypse, fire all pending event timers, and then exit as if the fault manager had received a `SIGTERM` exactly as the end of time was reached.

`sleep` *seconds*`;` — Pause execution of `fminject` for the specified number of *seconds* before continuing to the next statement.

◆ ◆ ◆ CHAPTER 11

fmsim Utility

The fmsim utility automates the process of starting up an alternate fault manager, loading additional modules and configuration files, and executing one or more error event injection scripts. fmsim can be used by developers to more easily debug new functionality without having to interrupt or restart the default fault manager active on the test system. Any number of fmsim simulations may be active simultaneously. fmsim can also be used by test groups to execute regression or test scenarios against a particular fault manager configuration, and can be used to replay recorded customer fault scenarios from the field. On Solaris systems, the fmsim command is available at /usr/lib/fm/fmd/fmsim.

11.1 Description

The fmsim utility copies the fault manager and its associated private libraries and configuration files to a *simulation world* that resembles a sparsely populated filesystem root, and then starts a fault manager inside of this world. As a result, log files such as the errlog and fltlog will be created for the simulation run in the directory var/fm/fmd relative to the root of the simulation world, and will not interfere with the default system fault manager running on the test system. In addition, fmsim can copy any number of additional configuration files such as modules under development or alternate topology configurations into the simulation world, and can inject events into the simulation using fminject. Once event injection is complete, or at the discretion of the user, the simulation ends and fmsim displays the contents of the fault log from the simulation world. The entire simulation world is then retained for inspection and verification by the user.

fmsim will attempt to copy the fault manager and its associated libraries and files from one of the following locations:

- If the CODEMGR_WS environment variable is set, fmsim will attempt to create the simulation world from the root directory $CODEMGR_WS/proto/root_'uname –p'.
- If the ROOT environment variable is set, fmsim will attempt to create the simulation world from the root directory $ROOT.

- Otherwise `fmsim` will create the simulation world from the directory `/`.

After all the injector input files specified on the command-line have been processed, `fminject` will send an event to `fmd` to cause the fault manager to advance the simulated clock to the end of time and then exit, ending the simulation. To start the fault manager without injecting any events and wait for user input, use the `-i` option as shown in the following example:

```
# /usr/lib/fm/fmd/fmsim -i
fmsim: creating simulation world /tmp/fmd.105296 ... done.
fmsim: populating /var ... done.
fmsim: populating /usr/lib/fm from / ... 3744 blocks
fmsim: populating /usr/lib/locale/C from / ... 3536 blocks
fmsim: populating /usr/sbin from / ... 192 blocks
fmsim: adding customizations: done.
fmsim: generating script ... done.
fmsim: simulation 105296 running fmd(1M) version 1.1 (DEBUG)
fmd: [ loading modules ... done ]
fmd: [ awaiting events ]
fmsim: rpc adm requests can rendezvous at 1073741824
fmsim: injectors should use channel com.sun:fm:fmd105296
fmsim: debuggers should attach to PID 105312
^C
TIME                 UUID                                 SUNW-MSG-ID
/tmp/fmd.105296/usr/sbin/fmdump: /tmp/fmd.105296/var/fm/fmd/fltlog is empty
#
```

As shown in the example, `fmsim` places the simulation world in `/tmp` by default; to change the location use the `-d` option. The `fmsim` output also includes the PID of `fmd`, its SysEvent *channel* for use with the `fminject -c` option, and its RPC program number for use with the `fmstat` and `fmadm -P` options.

Note – If you want to execute a series of `fmadm` and `fmstat` commands against the `fmd` running in simulation, you can set the `FMD_PROGRAM` environment variable to the RPC program number (1073741824 in the example) to indicate that `fmadm` and `fmstat` should use this program number instead of the default.

`fmsim` modifies several fault manager options in order to facilitate debugging. The fault manager clock is set to use a simulated clock instead of the system clock, indicating that time should appear to stand still until an `addhrtime` directive is received from `fminject`. `fminject` will insert `addhrtime` directives automatically if you replay a captured error event log. In addition to the simulated clock, `fmsim` modifies the following fault manager options:

`clock`	The `clock` property is set to `simulated` by default to indicate the fault manager should use the simulation clock.
`fg`	The `fg` property is set to `true` to cause the fault manager to run in the foreground and respond to `SIGINT`.

`rootdir`	The `rootdir` property is set to the pathname of the simulation world.
`rpc.adm.path`	The `rpc.adm.path` property is set to the pathname of a file in the simulation world so that `fmsim` can discover it.
`rpc.adm.prog`	The `rpc.adm.prog` property is set to zero to indicate that the fault manager should obtain a transient RPC program binding.
`sysevent-transport:channel`	The `channel` property for the `sysevent-transport` module is set to an alternate SysEvent error channel name derived from the PID of the fault manager.
`sysevent-transport:device`	The SysEvent transport replay device is set to `/dev/null` so that no error events will be replayed from a device on startup.

You can modify additional fault manager options or reconfigure the `fmsim` options using the `-o` command-line option or by specifying an `fmd.conf` file as an operand.

11.2 Options

The `fmsim` utility accepts the following command-line options:

`fmsim -ehisvVwx -d` *dir* `-D` *file.d* `-o` *opt=val* `-t` *args* `[`*file* `...]`

`-d` Specify an alternate simulation directory instead of the default directory located in `/tmp`.

`-D` Execute the fault manager under control of `dtrace`(1M) and enable instrumentation according to the D script *file.d*. The `$target` macro variable will expand to the process-ID of the fault manager. For more information about DTrace, refer to the *Solaris Dynamic Tracing Guide*.

`-e` Display the error event log at the end of the simulation rather than the fault event log.

`-h` Print a usage message for `fmsim` and exit.

`-i` Do not issue an `endhrtime` simulation control directive to the fault manager after input files have been processed, and instead wait for the fault manager to exit either as the result of an interrupt (Control-C) or an external control directive or signal. The `-i` option can be used with no input files to simply start the fault manager in the foreground awaiting further input or control from a debugger.

`-o` Add the specified *opt=val* directive to the command-line of the fault manager, changing the value of the specified property. Fault manager properties are described in more detail in Chapter 8, “Daemon Configuration.”

`-s` Set up the specified simulation but do not execute it. The simulation directory can be manually inspected and modified and the `run` script stored in the simulation directory can be used to start the simulation.

`-t` Start the fault manager under control of `truss`(1) and add the specified *args* to the `truss` command-line. If multiple command-line options must be specified to truss, they can be quoted to avoid interpretation by the shell. For example, to execute `truss -t open` to trace `open()` system calls, use the command:

```
# /usr/lib/fm/fmd/fmsim -t '-t open' ...
```

`-v` Add the `-v` option to display verbose event detail to the event display shown at the end of the simulation.

`-V` Add the `-V` option to display very verbose event detail to the event display shown at the end of the simulation.

`-w` Wait for the user to press a key before `fmsim` exits.

`-x` Delete the simulation results directory after the fault manager exits if it returns a successful (zero) exit status. If the `-x` option is not present, the simulation directory is retained after the simulation completes.

11.3 Operands

The `fmsim` utility accepts zero or more filename operands that represent additional fault manager modules, configuration files, and `fminject` input files. The files are recognized and processed automatically based upon their suffix and file contents. The supported file types are as follows:

`fmd.conf` The specified fault manager configuration file is copied to `etc/fm/fmd/fmd.conf`.

file`.conf` The specified plug-in module configuration file is copied to `usr/lib/fm/fmd/plugins/`.

file`.dict` The specified `libdiagcode.so.1` event dictionary is copied to `usr/lib/fm/dict/`.

file`.inj` The specified `fminject` input file is parsed and executed once the simulation is up and running.

file`.log` The specified fault manager error event log file is used as input to `fminject` once the simulation is up and running.

file`.mo` The specified message object is installed in `usr/lib/locale/$LANG/LC_MESSAGES/`.

file`.so` The specified shared library is examined and installed in `usr/lib/fm/fmd/plugins/`, `usr/lib/fm/fmd/schemes/`, or `usr/lib/fm/topo/`, depending on the entry points detected in the shared library.

file`.xml` The specified topology configuration file is copied to `usr/lib/fm/topo/`. See Chapter 9, "Topology" for information about topology.

◆ ◆ ◆ CHAPTER 12

fmtopo Utility

The fmtopo(1M) utility displays the contents of a topology. Developers can use fmtopo to more easily debug new functionality without interrupting or restarting the default fault manager that is active on the system. Any number of fmtopo commands can be active simultaneously. Test groups can use fmtopo to execute regression or test scenarios against a particular system configuration.

12.1 Description

The fmtopo utility opens a topology handle and takes a snapshot of the current configuration according to the specified FMRI scheme. By default, the scheme used to snapshot the topology is hardware components (hc).

12.2 Options

The fmtopo utility accepts the following command-line options:

fmtopo [-CedpSVx] [-P *group*.*property*[=*type*:*value*]] [-R *root*] [-s *scheme*] [*fmri*]

-C Execute and generate a core file

-d Execute in enumerator plug-in debug mode. Plug-in debug messages are displayed to stderr

-e Display output in eft.so format.

-h Print help information.

-p Display all protocol property groups for each topology node. This is the equivalent of the following command:

```
# /usr/lib/fm/fmd/fmtopo -P protocol
```

`-P` Display or set the specified group or group and property. For example, to display the `product` property in the `authority` group, use the following command:

```
# /usr/lib/fm/fmd/fmtopo -P authority.product-id *hostbridge=10
```

If the reserved property group `all` is used, `fmtopo` displays all property groups, names, types, and values for the specified topology.

```
# /usr/lib/fm/fmd/fmtopo -P all
```

`-R` Set the relative root directory for loading topology map files and plug-in modules. This command option can also be used in a simulation environment for displaying topology. See Chapter 11, "`fmsim` Utility," for more details.

`-s` Display the topology for the specified *scheme*. Valid values are:

```
cpu
dev
hc
mod
pkg
```

`-S` Display FMRI status for present and unusable.

`-V` Set verbose mode for displaying properties.

`-x` Display an XML formatted topology. This command option can be used to produce a complete topology map for the current snapshot. The file can be used to in simulation environments to test and debug problems.

12.3 Operands

The `fmtopo` utility accepts a FMRI operand. The specified *fmri* is displayed along with any property information requested with command-line options. For example, to display the FMRI `hc://motherboard=0/chip=0/cpu=0`, use the command:

```
# /usr/lib/fm/fmd/fmtopo hc://motherboard=0/chip=0/cpu=0
```

The FMRI can be a complete path or can be constructed by using patterns supported by `fnmatch`(5). For example, to display all FMRIs with `cpu` in their path, use the command:

```
# /usr/lib/fm/fmd/fmtopo *cpu*
```

◆ ◆ ◆ CHAPTER 13

13 Debugging

This chapter discusses debugging techniques for the fault manager and client modules, including documentation for Project Private MDB debugger commands and pointers to debugger commands for `libumem.so.1`. You should be familiar with the concepts described in Chapter 2, "Module API," Chapter 5, "Log Files," Chapter 6, "Resource Cache," and Chapter 7, "Checkpoints" before you read this chapter. Some of the debugging techniques presented in this chapter rely on tools and libraries found in the Solaris OS. These tools and libraries might not be available on other systems. You might also need to refer to the fault manager source code in order to explore a particular area in more detail.

13.1 MDB Debugging Support

The fault manager provides a companion module for customized debugging support that loads whenever MDB is applied to a live running fault manager or a fault manager core file. For more information on MDB, refer to the *Solaris Modular Debugger Guide*. You can use the MDB command `::dmods -l fmd` to list the debugging commands available for the fault manager. In addition, the fault manager is compiled with CTF symbolic debugging information that permits MDB to view the C data structures associated with the fault manager. For example, you can view the fault manager's global data structures using the following command:

```
> fmd::print
{
    d_version = _fmd_version "1.1"
    d_pname = 0xffbfff3c "fmd"
    d_pid = 0x18c53
    d_key = 0x1
    d_signal = 0
    d_running = 0x1
    d_fmd_debug = 0
    d_fmd_dbout = 0
    d_hdl_debug = 0
```

```
    d_hdl_dbout = 0
    ...
```

13.1.1 fmd_case

You can use the `::fmd_case` dcmd to display a table of the active fault manager cases, as shown in the following example:

```
> ::fmd_case
ADDR        STATE REF DATA     UUID
32fd80      UNSLV 1   28b370   c82e1dbc-d3f9-efa3-ba80-9df52b01ba52
```

13.1.2 fmd_module

You can use the `::fmd_module` dcmd to display a table of the active fault manager modules, as shown in the following example:

```
> ::fmd_module
ADDR        OPS              DATA        FLAG USTAT    NAME
2c7e40      0                0           0x00 27df18   fmd
2c7cc0      fmd_bltin_ops    300b8       0x01 27de38   fmd-self-diagnosis
2c7b40      fmd_rtld_ops     298300      0x01 27dce8   cpumem-diagnosis
2c76c0      fmd_rtld_ops     3b9e20      0x01 27d968   cpumem-retire
2c7540      fmd_rtld_ops     3b9880      0x01 27d818   syslog-msgs
2c73c0      fmd_rtld_ops     3b9360      0x01 27d738   eft
2c6f40      fmd_rtld_ops     3b8800      0x01 27d508   io-retire
```

Threads that are not associated with any client module are associated with the built-in `fmd` module, which always appears first in the module list.

13.1.3 fmd_timer

The `::fmd_timer` dcmd displays a table of the pending timers, the high-resolution time at which they are set to expire, the owning module, and callback function that will be executed when the timer fires.

```
> ::fmd_timer
ADDR     MODULE               ID   HRTIME             ARG      FUNC
6cf6c0   fmd                     5 0x189939670d8c0    49958    fmd_gc
```

13.1.4 fmd_trace

The fault manager keeps a per-thread trace buffer corresponding to events of interest as it executes. The trace buffers provide a "black box" view of recent events of interest that can retrieved from a fault manager core file. The `trace.mode` property can be set to the tokens `none`, `lite`, or `full` to indicate how much trace data should be recorded. The `lite` token records trace metadata only. The `full` token records trace metadata and a complete stack trace at each trace point indicated in the fault manager source code. By default, a `DEBUG` fault manager has the `full` trace enabled and a non-`DEBUG` fault manager has the `lite` trace enabled.

The `trace.recs` property can be used to tune the maximum number of trace records per thread, and the `trace.frames` property can be used to tune the maximum number of stack trace frames that are recorded for each event. Each thread in the fault manager pre-allocates the maximum trace buffer size in advance and uses the space as a ring buffer where new events overwrite old ones once the buffer is full. By default, the `::fmd_trace` dcmd merges together the output of all thread trace buffers and displays it in time order from newest to oldest, as shown in the following example:

```
> ::fmd_trace
TID TIME             TAG  ERRNO MSG
 11 0002c1109af2ca50 0004 0     hold 2c73c0 (eft/17)
  5 00013aff05347610 0080 0     tmr fmd:4 exec end
  5 00013aff05222f00 0080 0     timer fmd:5 insert +86400000000000ns
  5 00013aff05113150 0004 0     garbage collect end
  5 00013aff04faabb0 0004 0     garbage collect start
  5 00013aff04e7d080 0080 0     tmr fmd:4 exec start (hrt=13aff04892760)
  5 0000ec6a73546300 0080 0     tmr fmd:3 exec end
```

The `MSG` column is a descriptive message recorded by the corresponding `TRACE()` call in the fault manager source code, and the `ERRNO` column shows that thread's `errno` value if it is considered relevant. To display the C source code location of the trace point instead of the descriptive message, use the `::fmd_trace -c` option, as shown in the following example:

```
> ::fmd_trace -c
TID TIME             TAG  FILE:LINE
 11 0002c1109af2ca50 0004 ../common/fmd_module.c: 785
  5 00013aff05347610 0080 ../common/fmd_timerq.c: 201
  5 00013aff05222f00 0080 ../common/fmd_timerq.c: 88
  5 00013aff05113150 0004 ../common/fmd.c: 536
  5 00013aff04faabb0 0004 ../common/fmd.c: 534
  ...
```

If the trace mode is set to `full`, the stack trace for each trace record can be retrieved using the `::fmd_trace -s` option:

```
> ::fmd_trace -s
TID TIME             TAG  ERRNO MSG
```

```
 11 0002c1109af2ca50 0004 0     hold 2c73c0 (eft/17)
       fmd_trace+0x40
       fmd_module_hold+0x28
       fmd_dispq_insert+0x34
       fmd_hdl_subscribe+0x38
       eft.so‘dosubscribe+0x48
       eft.so‘lut_walk+0x18
       ...
```

13.1.5 fmd_ustat

The `::fmd_ustat` dcmd displays the contents of a statistics hash table either from the root module or from one of the fault manager's client modules. The statistics hash table addresses can be obtained from the `USTAT` column of the `::fmd_module` dcmd. To retrieve the statistics for the `cpumem-retire` module shown in the earlier example, you would use the following command:

```
> 27d968::fmd_ustat
ADDR        TYPE NAME                            VALUE
fe6f4518    ui64 cpu_supp                        0
37c5a0      time fmd.wlastupdate                 346341952612240
37c290      ui64 fmd.dequeued                    4
37c760      size fmd.memtotal                    0
37c8b0      size fmd.buflimit                    10485760
37c370      ui64 fmd.accepted                    0
fe6f44a8    ui64 cpu_blfails                     0
...
```

13.1.6 fmd_xprt

The `::fmd_xprt` dcmd displays the details of a particular transport handle. To obtain the list of transport handles, for a given module, apply the `fmd_xprt` walker to the module address, obtained from the output of `::fmd_module`. For example:

```
812d7c0::fmd_xprt
ADDR     ID   VERS FLAGS STATE
812d7c0  2    0    17    _fmd_xprt_state_run
```

With the `-s` option, `::fmd_xprt` will display the cached list of subscriptions associated with the transport handle:

```
812d7c0::fmd_xprt -s
ADDR     ID   VERS FLAGS STATE
812d7c0  2    0    17    _fmd_xprt_state_run
```

```
ADDR     REFS CLASS
834e160  1    ereport.io.sca1000.hw.device
834e0a0  1    ereport.io.pci.sec-dpe
83b7ee0  1    ereport.io.pci.target-mdpe
83b7d60  1    ereport.io.pci.sec-ma
83b7c20  1    ereport.io.pci.dto
83b7de0  1    ereport.io.pci.target-rta
83b7ce0  1    ereport.io.pci.sec-rserr
...
```

13.2 Memory Leaks and Corruption

The fault manager Solaris reference implementation uses the `libumem.so.1` allocator for all of its memory allocation. As a result, a rich set of debugging capabilities are available for debugging memory leaks and memory corruption in both the fault manager and its client modules. More information about using `libumem.so.1` for debugging can be found in the `umem_debug`(3MALLOC) man page. By default, a `DEBUG` fault manager has `UMEM_DEBUG` enabled and a non-`DEBUG` fault manager does not. You can determine the compilation mode of the fault manager using the `-V` command-line option:

```
# /usr/lib/fm/fmd/fmd -V
/usr/lib/fm/fmd/fmd: version 1.1 (DEBUG)
```

If the `DEBUG` flag was not part of the compilation mode, you can set the `UMEM_DEBUG` environment variable prior to starting the fault manager to enable memory debugging features.

The dcmds described in this section are typically applied to a core file of the fault manager, either after a failure or by forcing the fault manager to core dump. To grab a core file of a live running fault manager, use the `gcore`(1) command. To force the fault manager to core dump just before exiting after a testing or simulation run, set the `core` property to `true`. Additional debugging techniques with the Solaris memory allocator are described further in the *Solaris Modular Debugger Guide*.

13.2.1 findleaks

The `::findleaks` command can be used to check a fault manager core file for memory leaks. This dcmd prints a report of each leak, aggregated by stack trace. If any leaks are found, use the `::bufctl_audit` dcmd to retrieve the stack trace and other information relating to each leak. This dcmd will detect leaks in the fault manager, its client modules, and even in any libraries it is using. Verify the stack trace before filing any bugs to be sure that the leak is actually related to the fault manager and not to some other software such as a library that is being used by the daemon or one of its client modules.

If a core file is forced after a plug-in module has been unloaded, MDB will not be able to translate the stack trace program counters from `::bufctl_audit` to symbolic names because

the plug-in module is no longer present in the address space (that is, after `dlclose()` has been called). To simplify debugging of memory leaks that occur after modules are unloaded, you can set the `plugin.close` property to `false` to tell the fault manager not to `dlclose()` plug-in modules after unloading them. Module developers should test unloading their module and checking for leaks in this manner to ensure that `fmadm reset` on the corresponding module does not induce memory leaks in the fault manager.

13.2.2 umem_verify

The `::umem_verify` command can be used to check a fault manager core file for memory corruption, including latent corruption in any memory caches that has not yet been accessed. Developers should apply `::umem_verify` as well as `::findleaks` to a core file of the fault manager at the end of every testing run to ensure that no latent bugs exist that have not yet been detected.

13.3 Debug Messages

The fault manager contains a small set of debug messages corresponding to major events in non-performance-critical paths. The global `debug` property can be set to a comma-separated list of tokens to enable messages for various subsystems. The token `all` can be used to enable all debug messages. To list the set of debug message tokens, use the command `fmd -o debug=help`. The global `dbout` property can be set to the tokens `stderr` or `syslog` to indicate the destination for the fault manager's debug messages. If the `fg` property is set to `true`, the `dbout` property assumes `stderr` as its default value.

The fault manager's client modules can also report debug messages using the `fmd_hdl_debug()` and `fmd_hdl_vdebug()` functions described in Chapter 2, "Module API." These messages can be enabled by setting the global `client.debug` property to `true`. The global `client.dbout` property can be set to the tokens `stderr` or `syslog` to indicate the destination for client module debug messages. If the `fg` property is set to `true`, the `client.dbout` property assumes `stderr` as its default value.

13.4 Checkpoint Files

The fault manager keeps a checkpoint file for each module, as described in Chapter 7, "Checkpoints." The `fmd.so` MDB debugging module also provides debugging commands for examining FMD Checkpoint File (FCF) data stored in memory or saved on disk. The FCF format is defined in `fmd/common/fmd_ckpt.h` and is considered Project Private. To apply `mdb` to a checkpoint file, execute the 32–bit debugger and specify the checkpoint file as an argument, and then load `fmd.so`, as shown in the following example:

```
# /usr/bin/sparcv7/mdb /var/fm/fmd/ckpt/cpumem-diagnosis/cpumem-diagnosis
> ::load /usr/lib/mdb/proc/fmd.so
```

At present, `fmd` is a 32–bit process and therefore the 32–bit debugger `/usr/bin/sparcv7/mdb` or `/usr/bin/i86/mdb` must be used to debug FCF files. Once the `fmd.so` module is loaded, any of the debugging commands described in this section can be applied to the checkpoint file. You can also use any of the MDB data formatting commands to examine the content of the file, such as `::dump` or `/X` and so forth. The *Solaris Modular Debugger Guide* has more information on using MDB to debug raw data files.

13.4.1 fcf_hdr

The `::fcf_hdr` dcmd displays the checkpoint file header, which includes the data model and endianness of the checkpoint, the version of the data format, and other metadata. The `fcfh_cgen` field indicates the checkpoint generation number, which is incremented each time a successful checkpoint is written for each client module.

```
> ::fcf_hdr
fcfh_ident.id_magic = 0x7f, F, C, F
fcfh_ident.id_model = ILP32
fcfh_ident.id_encoding = MSB
fcfh_ident.id_version = 1
fcfh_flags = 0x0
fcfh_hdrsize = 64
fcfh_secsize = 32
fcfh_secnum = 12
fcfh_secoff = 64
fcfh_filesz = 1522
fcfh_cgen = 2
```

13.4.2 fcf_sec

The `::fcf_sec` dcmd can be used to display the section header table of the checkpoint file. Each section has an integer index shown in the left-hand column and various other attributes. Sections refer to one another using the section indices. The `OFF` column indicates the byte offset of the section data within the checkpoint file. You can use this offset with a data formatting dcmd such as `::dump` to view the raw section data. The `fmd.so` debugging module provides dcmds corresponding to some section types, such as `::fcf_case`.

```
> ::fcf_sec
NDX     ADDR TYPE       ALIGN FLAGS ENTSZ OFF    SIZE
  0       40 none       1     0     0     1c0    0
  1       60 buffer     8     0     0     1c0    0x38
  2       80 bufs       4     0     0x8   1f8    0x8
  3       a0 case       4     0     0     200    0x18
  4       c0 events     8     0     0x28  218    0x28
  5       e0 serd       8     0     0x18  240    0x18
```

```
  6      100 buffer     8     0     0     258    0x80
  7      120 buffer     8     0     0     2d8    0xb0
  8      140 buffer     8     0     0     388    0x13c
  9      160 bufs       4     0     0x8   4c4    0x18
 10      180 module     4     0     0     4dc    0x14
 11      1a0 strtab     1     0     0     4f0    0x102
```

If a section has a non-zero entry size (`ENTSZ`), then the section data is an array of data objects that are each `ENTSZ` bytes and the total number of objects is `SIZE / ENTSZ`. When section objects contain variable-length strings, these are usually kept in a single string table at the end of the file, shown above as section 11. For example, section 9 is a buffer table that contains, for each global buffer in the checkpoint, the name of the buffer and the index of the section that stores the module-specific data for that buffer. This information corresponds to the structure `fcf_buf_t` in `fmd/common/fmd_ckpt.h`. To view the array of `fcf_buf_t` structures from section 9 above, you can use the following MDB command:

```
> 4c4,18%8/XDn
0x4c4:          70              6
                7b              7
                87              8
```

The output shows three global buffers for this module, stored in sections 6, 7, and 8 respectively. The names for the buffers can be retrieved by adding the string table offset to the offset of the string table, as shown in the following example:

```
> 4f0+70/s
0x560:          cpu_fru_16
```

13.4.3 fcf_case

The contents of a case section can be formatted using the `::fcf_case` dcmd and the offset of the section data, as shown in the following example. Similar to other sections, the `fcfc_uuid` member contains the offset within the string table of the case's UUID string.

```
> 200::fcf_case
fcfc_uuid = 0x26
fcfc_state = 0
fcfc_bufs = 2
fcfc_events = 0
fcfc_suspects = 0
> 4f0+26/s
0x516:          c82e1dbc-d3f9-efa3-ba80-9df52b01ba52
```

13.4.4 fcf_event

The individual event references in an event section can be formatted using the `::fcf_event` dcmd and the offset of an event structure. The representation of events in checkpoints is identical to that used for log files, and is described further in "5.1.2 Event References" on page 87.

```
> 218::fcf_event
fcfe_todsec = 1095284089 (2004 Sep 15 14:34:49)
fcfe_todnsec = 649233720
fcfe_major = 85
fcfe_minor = 650
fcfe_inode = 7321
fcfe_offset = 47364
```

13.4.5 fcf_serd

The individual SERD engines in a SERD section can be formatted using the `::fcf_serd` dcmd and the offset of a SERD structure. Similar to other sections, the `fcfd_name` member contains the offset within the string table of the SERD engine's name string.

```
> 240::fcf_serd
fcfd_name = 0x4b
fcfd_events = 4
fcfd_n = >3
fcfd_t = 43200000000000ns
> 4f0+4b/s
0x53b:          c82e1dbc-d3f9-efa3-ba80-9df52b01ba52
```

13.5 Topology Library Debugging

This section discusses debugging techniques for the topology library and plug-in modules, including documentation for Project Private MDB debugger commands and pointers to debugger commands for `libumem.so.1`. You should be familiar with the concepts described in Chapter 9, "Topology" before you read this sections. You might also need to refer to the `libtopo` source code to explore a particular area in more detail.

13.5.1 MDB Debugging Support

The topology library provides a companion module for customized debugging support that loads whenever MDB is applied to a process or core file linked to `libtopo.so.1`. You can use the following dcmd to list the debugging commands available for the topology library.

```
::dmods -l libtopo.so.1
```

13.5.1.1 MDB Dcmds

fmtopo

You can use the `::fmtopo` dcmd to display a topology snapshot as you would see with the `fmtopo` utility.

```
> 27d960::fmtopo
hc://motherboard=0
hc://motherboard=0/hostbridge=0
hc://motherboard=0/hostbridge=0/pcibus=0
hc://motherboard=0/hostbridge=0/pcibus=0/pcidev=4
hc://motherboard=0/hostbridge=0/pcibus=0/pcidev=4/pcifn=0
hc://motherboard=0/hostbridge=0/pcibus=0/pcidev=32
hc://motherboard=0/hostbridge=0/pcibus=0/pcidev=32/pcifn=0
hc://motherboard=0/hostbridge=0/pcibus=1
hc://motherboard=0/hostbridge=0/pcibus=1/pcidev=5
hc://motherboard=0/hostbridge=0/pcibus=1/pcidev=5/pcifn=0
hc://motherboard=0/hostbridge=0/pcibus=1/pcidev=5/pcifn=1
hc://motherboard=0/hostbridge=0/pcibus=1/pcidev=5/pcifn=2
hc://motherboard=0/hostbridge=0/pcibus=1/pcidev=5/pcifn=3
hc://motherboard=0/hostbridge=0/pcibus=1/pcidev=6
hc://motherboard=0/hostbridge=0/pcibus=1/pcidev=6/pcifn=0
hc://motherboard=0/hostbridge=0/pcibus=1/pcidev=6/pcifn=1
hc://motherboard=0/hostbridge=0/pcibus=1/pcidev=32
hc://motherboard=0/hostbridge=0/pcibus=1/pcidev=32/pcifn=0
hc://motherboard=0/cpu=0
hc://motherboard=0/cpu=1
```

Use the `::fmtopo` dcmd with the `-s` option to display an alternate topology based on *scheme*.

```
::fmtopo -s scheme
```

Valid scheme values are:

```
cpu
dev
hc
mod
pkg
```

Use the `-v` option to display the address of each `tnode_t` `struct` and redirect to the `topo_node` dcmd described below.

Use the `-P` *property-group* option to display the name of a property group and the entire property listing for that group as shown below.

```
> 27d960::fmtopo -P authority
hc://:product-id=SUNW,Sun-Blade-1000/motherboard=0
  group: authority                       version: 1, stability: Private/Private
    product-id                      string          value: 3fa28
    server-id                       string          value: 3f9f8
hc://:product-id=SUNW,Sun-Blade-1000/motherboard=0/hostbridge=0
  group: authority                       version: 1, stability: Private/Private
    product-id                      string          value: 3fa28
    server-id                       string          value: 3f9f8
hc://:product-id=SUNW,Sun-Blade-1000/motherboard=0/hostbridge=0/pcibus=0
  group: authority                       version: 1, stability: Private/Private
    product-id                      string          value: 3fa28
    server-id                       string          value: 3f9f8
hc://:product-id=SUNW,Sun-Blade-1000/motherboard=0/hostbridge=0/pcibus=0/pcidev=4
  group: authority                       version: 1, stability: Private/Private
    product-id                      string          value: 3fa28
    server-id                       string          value: 3f9f8
. . .
```

The property value address can be decoded by piping it to the `::nvlist` dcmd.

topo_handle

The `::topo_handle` command formats the contents of a topology handle (`topo_hdl_t`).

```
> 35f80::topo_handle
FIELD        VALUE                              DESCR
th_lock      0xffbfeb60                         Mutex lock protecting handle
th_uuid      ...                                UUID of the topology snapshot
th_rootdir   /                                  Root directory of plugin paths
th_platform  SUNW,Sun-Blade-1000                Platform name
th_isa       sparc                              ISA name
th_machine   sun4u                              Machine name
th_product   SUNW,Sun-Blade-1000                Product name
th_di        0x0                                Handle to the root of the devinfo tree
th_pi        0x0                                Handle to the root of the PROM tree
th_modhash   0x39fb8                            Module hash
th_trees                                        Scheme-specific topo tree list
  l_prev       0x3fd88
  l_next       0x3ff80
th_alloc     0x37fb0                            Allocators
tm_ernno     1035                               errno
tm_debug     0                                  Debug mask
tm_dbout     0                                  Debug channel
```

topo_module

The `::topo_module` command formats the contents of a topology plug-in module handle (`topo_mod_t`).

```
> 41ce8::topo_module
FIELD        VALUE                              DESCR
tm_lock      0xffbfeb98                         Lock for tm_cv/owner/flags/refs
tm_cv        0xffbfeb88                         Module condition variable
tm_busy      <NULL>                             Busy indicator
tm_next      0x41d58                            Next module in hash chain
tm_hdl       0x35f80                            Topo handle for this module
tm_alloc     0x37fb0                            Allocators
tm_name      hc                                 Basename of module
tm_path      <0>                                Full pathname of module
tm_rootdir   /                                  Relative root directory of module
tm_refs      5                                  Module reference count
tm_flags     5                                  Module flags
             TOPO_MOD_INIT                      Module init completed
             TOPO_MOD_REG                       Module registered
tm_debug     0                                  Debug printf mask
tm_data      0xff38ae28                         Private rtld/builtin data
tm_mops      0xff38ae48                         Module class ops vector
tm_info      0x3bd90                            Module info registered with handle
tm_ernno     1015                               Module errno
```

topo_node

The `::topo_node` command formats the contents of a topology node (`tnode_t`).

```
> 35aa0::topo_node
FIELD        VALUE                          DESCR
tn_lock      0xffbfec10                     Mutex lock protecting node members
tn_name      cpu                            Node name
tn_instance  0                              Node instance
tn_state     4                              Node state
             TOPO_NODE_BOUND
tn_fflags    0                              FMRI flags
tn_parent    0x35bc0                        Node parent
tn_phash     0xd36a0                        Parent hash bucket
tn_hdl       0x35f80                        Topo handle
tn_enum      0x41b28                        Enumerator module
tn_children                                 Hash table of child nodes
  l_prev       0x0
  l_next       0x0
tn_pgroups   0xffbfecc0                     Property group list
tn_methods   0xffbfecc8                     Registered method list
tn_priv      0x0                            Private enumerator data
```

```
tn_refs      2                              Node reference count
```

13.5.1.2 MDB Walkers

topo_tree

The `::topo_tree` walker walks the list of scheme-specific topology trees for a given topology snapshot.

```
> 35f80::walk topo_tree
3ff80
3ff38
3fe18
3fde8
3fdb8
3fd88
```

To get more detailed information, pipe the output of the `topo_tree` walker to the `::print` dcmd to dump the `ttree_t` structure as shown below:

```
> 35f80::walk topo_tree | ::print ttree_t
{
    tt_list = {
        l_prev = 0
        l_next = 0x3ff38
    }
    tt_scheme = 0x3df98 "cpu"
    tt_mod = 0x41f18
    tt_root = 0x35f20
    tt_walk = 0x3ff68
}
{
    tt_list = {
        l_prev = 0x3ff80
        l_next = 0x3fe18
    }
    tt_scheme = 0x3df60 "dev"
    tt_mod = 0x41ea8
    tt_root = 0x35e00
    tt_walk = 0x3ff20
}
. . .
```

topo_nodehash

The `::topo_nodehash` walker walks the table of topology sub-nodes for the specified node address.

```
> 34ea0::walk topo_nodehash
34e40
34de0
34d80
34ba0
0
0
0
0
0
```

To get more detailed information on each node in the node hash, pipe the `::topo_nodehash` output to the `::topo_node` dcmd.

topo_pgroup

The `::topo_pgroup` walker walks the property group list for the specified node.

```
> 34ea0::walk topo_pgroup
c6f48
c6e10
```

topo_proplist

The `::topo_proplist` walker walks the list of properties for the specified property group address.

```
> c6f48::walk topo_proplist
c04a0
c0400
c03f0
```

To get detailed information on each property, pipe the output of `::topo_proplist_t` to `::print topo_propval_t` and then to `::nvlist` as shown below.

```
> c04a0::print topo_proplist_t tp_pval | ::print topo_propval_t tp_val | ::nvlist
property-value
    version=00
    scheme='hc'
    hc-root=''
    authority
        product-id='SUNW,Sun-Blade-1000'
        server-id='grim'
    hc-list[0]
        hc-name='motherboard'
        hc-id='0'
    hc-list[1]
```

```
        hc-name='hostbridge'
        hc-id='0'
    hc-list[2]
        hc-name='pcibus'
        hc-id='1'
    hc-list[3]
        hc-name='pcidev'
        hc-id='5'
```

topo_module

The `::topo_module` walker displays the table of plug-in module opaque handle addresses.

```
> 35f80::walk topo_module
41b28
41c08
```

To get detailed information on each module, pipe the output to the `::topo_module` dcmd.

13.5.2 Memory Leaks and Corruption

See findleaks and umem_verify.

◆ ◆ ◆ CHAPTER 14

syslog-msgs Agent

Sun has defined a messaging standard for fault messages associated with `list.suspect` events. Fault messages are produced when the diagnosis of a problem requires a human administrator to do something or to be aware of a problem that may impact system availability or service levels. The fault manager expects that at least one of its modules implement a messaging service for `list.suspect` events; in the Solaris reference implementation this module is the `syslog-msgs` agent and it produces a FMA standard message to the system console and system log file using the `syslogd`(1M) service. This chapter briefly describes the design of this agent and lists the Private configuration properties associated with this module.

14.1 Design Overview

The `syslog-msgs` agent is perhaps the simplest agent in that it has no persistent state considerations and subscribes only to a single event class, `list.suspect`. For each event it receives, it formats a corresponding message for the `syslogd`(1M) service and emits the message to `/dev/log` and `/dev/sysmsg` (the `syslog()` routine cannot be used here due to the multi-line nature of the message). Here's an example message associated with `fmd`:

```
SUNW-MSG-ID: FMD-8000-0W, TYPE: Defect, VER: 1, SEVERITY: Minor
EVENT-TIME: Fri Jan 23 18:33:31 PST 2004
PLATFORM: SUNW,Sun-Fire-V440, CSN: -, HOSTNAME: mix
SOURCE: fmd-self-diagnosis, REV: 1.0
EVENT-ID: e9390b15-bcb8-4a3d-c10c-fe1cb4a67998
DESC: The Solaris Fault Manager received an event from a component to which no
automated diagnosis software is currently subscribed. Refer to
http://sun.com/msg/FMD-8000-0W for more information.
AUTO-RESPONSE: Error reports from the component will be logged for examination by Sun.
IMPACT: Automated diagnosis and response for these events will not occur.
REC-ACTION: Run pkgchk -n SUNWfmd to ensure that fault management software is
installed properly.  Contact Sun for support.
```

Notice that the message includes a UUID for the diagnosis (the field labeled “EVENT-ID”) and a static message identifier (the field labeled “SUNW-MSG-ID”). The UUID is the UUID of the case that the diagnosis engine used to gather information for this diagnosis. The message identifier is a code computed from the class strings of the individual fault events that are part of the suspect list; it is generated by `libdiagcode.so.1` (see PSARC 2003/323 and PSARC 2004/601. The UUID can be used as an argument to `fmdump`(1M) to retrieve all of the telemetry information associated with the diagnosis or the details of the individual suspected faults. The message identifier can be used as an argument to a CGI script on Sun's web site to retrieve a knowledge article explaining more about the problem and appropriate responses. The FMA team is working with Enterprise Services to build this web site and populate it automatically from a company-wide registry of all FMA events.

The content of the DESC, AUTO-RESPONSE, IMPACT, and REC-ACTION fields is derived statically for the event based upon the SUNW-MSG-ID value, and is intended to be stored in the Sun FMA event registry, and then checked into the source base of the corresponding product. The messages themselves are stored in a portable message object file suitable for processing by `msgfmt`(1) so that the message can be localized. For example, the message above is stored in the message object file `/usr/lib/locale/C/LC_MESSAGES/FMD.po`, which is derived from the source file `FMD.mo` which is stored with the fault manager source code. This message object corresponds to the fault manager's event code dictionary `/usr/lib/fm/dict/FMD.dict`, and contains the following entries for the above message:

```
msgid "FMD-8000-0W.type"
msgstr "Defect"
msgid "FMD-8000-0W.severity"
msgstr "Minor"
msgid "FMD-8000-0W.description"
msgstr "The Solaris Fault Manager received an event from a component to which\
no automated diagnosis software is currently subscribed. Refer to %s for more\
information."
msgid "FMD-8000-0W.response"
msgstr "Error reports from the component will be logged for examination by Sun."
msgid "FMD-8000-0W.impact"
msgstr "Automated diagnosis and response for these events will not occur."
msgid "FMD-8000-0W.action"
msgstr "Run pkgchk -n SUNWfmd to ensure that fault management software is\
installed properly.  Contact Sun for support."
```

The `%s` specifier in the description is substituted with the URL of the appropriate knowledge article, derived from the `url` property and the message identifier. Note that in the first version of `syslog-msgs`, no facility for other dynamic content is provided that can be derived from other content in the `list.suspect` event. The FMA team plans to investigate additions of such dynamic content as future work.

14.2 Properties

The `syslog-msgs` agent supports the following properties:

TABLE 14–1 `syslog-msgs` properties

Name	Type	Default	Description
`console`	`boolean`	`true`	If set, emit messages to `/dev/sysmsg`, which sends them to the system console and any configured alternate consoles.
`facility`	`string`	`LOG_DAEMON`	Specify the syslog facility to use for messages as a string. The value may be set either to LOG_DAEMON or to one of LOG_LOCAL[0–7].
`gmt`	`boolean`	`false`	If set, emit diagnosis time in GMT rather than the current local timezone.
`syslogd`	`boolean`	`true`	If set, emit messages to `/dev/log`, which will then forward them to `syslogd`(1M). The `syslog.conf` file will determine what happens from there, according to the facility, except that no console messages will occur; these are controlled using the `console` property.
`url`	`string`	`http://sun.com/msg/`	The URL to use as a prefix for indicating the location of knowledge articles. This property is provided to permit sites to customize knowledge articles with local procedures as necessary.

◆ ◆ ◆ CHAPTER 15

snmp-trapgen Agent

Like `syslog-msgs`, the `snmp-trapgen` agent generates messages in response to `list.suspect` events. For each `list.suspect` event that is received, `snmp-trapgen` sends an SNMPv1 trap and/or SNMPv2c notification containing a subset of information available about the diagnosed fault. A complementary plug-in for the *System Management Agent (SMA)*, if configured, provides additional information as specified by the Fault Management MIB. The Solaris SMA is a port of the OpenSource NetSNMP software, so the reference implementation here can be used on other systems that support NetSNMP. See http://net-snmp.sourceforge.net for more information about NetSNMP. The information provided by the Fault Management MIB and SNMP trap is similar to that logged by the `syslog-msgs` agent or accessible via `fmadm`(1M) and `fmdump`(1M) utilities.

15.1 Design Overview

The design of `snmp-trapgen` is very similar to that of `syslog-msgs` described in Chapter 14, "`syslog-msgs` Agent." The agent subscribes to a single event class, `list.suspect`, and generates a single trap in response to each event. The underlying NetSNMP library formats this trap data appropriately and sends the trap to destinations determined by the SNMP agent configuration. The contents of the trap are defined by the Sun Fault Management MIB and are limited in size to improve simplicity and reliability. The following example illustrates a trap as displayed by `snmptrapd`(1M) on a network management station:

```
2006-02-07 16:36:34 stomper [192.xx.xx.xx]:
    DISMAN-EVENT-MIB::sysUpTimeInstance = Timeticks: (2266748911) 262 days, 8:31:29.11
    SNMPv2-MIB::snmpTrapOID.0 = OID: SUN-FM-MIB::sunFmProblemTrap
    SUN-FM-MIB::sunFmProblemUUID."a58aa105-4fab-6e16-8557-ab7687113de7" =
        STRING: "a58aa105-4fab-6e16-8557-ab7687113de7"
    SUN-FM-MIB::sunFmProblemCode."a58aa105-4fab-6e16-8557-ab7687113de7" =
        STRING: SUN4U-8000-KA
    SUN-FM-MIB::sunFmProblemURL."a58aa105-4fab-6e16-8557-ab7687113de7" =
        STRING: http://sun.com/msg/SUN4U-8000-KA
```

Notice that the trap includes the UUID for the diagnosis (`sunFmProblemUUID`) and the static message identifier (`sunFmProblemCode`). The UUID is that of the case opened by the diagnosis engine, and can be used as an index into tables in the Sun Fault Management MIB. The message identifier is the `libdiagcode.so.1` identifier computed from the fault events in the suspect list. It can be used with an application on Sun's web site to retrieve a knowledge base article containing additional information about the problem and appropriate responses. The `sunFmProblemURL` trap varbind provides the URL of the Sun knowledge article corresponding to this diagnosis. The information in the trap alone is sufficient to direct administrators to both more detailed information and instructions for taking immediate action.

The agent is capable of generating both SNMPv1 traps and SNMPv2c notifications independently, and can send these to any number of trap sinks specified either by the existing SMA configuration file located on Solaris systems at `/etc/sma/snmp/conf/snmpd.conf` or an optional supplementary configuration located at `/etc/sma/snmp/conf/fmd-trapgen.conf`. Traps or notifications will be sent, as appropriate, to any sink specified by either configuration file. The optional supplemental configuration also uses the `snmpd.conf`(4) syntax. Both configuration files are read only once, when the module is first loaded.

15.2 Sun Fault Management MIB

The Sun Fault Management MIB definition is installed in `/etc/sma/snmp/mibs/SUN-FM-MIB.mib` on Solaris systems and can be made available to SNMP command-line utilities such as `snmpwalk`(1M) by adding appropriate directives to `/etc/sma/snmp/snmp.conf`. Configuring `snmp.conf`(4) is not strictly necessary to browse the MIB; it is possible to use any standards-based SNMP *Network Management Station (NMS)*, or to use numeric *object identifiers (OIDs)* with the SMA utilities. Access to the MIB is provided by an architecture-specific SMA plug-in module, `/usr/lib/fm/libfmd_snmp.so.1`, or its 64–bit equivalent. The SMA master agent, `snmpd`(1M), will load this plug-in if `snmpd.conf`(4) directs it to do so. The plug-in requires no configuration.

The tables below describe the information available from the MIB; similar descriptions are available by executing the `snmptranslate` command with the `-Tp` and `-Td` options. All of the objects in the MIB are rooted at the OID `.iso.org.dod.internet.private.enterprises.sun.products.fm.sunFmMIB` (`.1.3.6.1.4.1.42.2.195.1`); for brevity, the tables below show only the last two OID components.

TABLE 15–1 Fault Management MIB Top-Level Objects

OID	Name	Description
sunFmMIB.1	sunFmProblemTable	Table listing all of the known problems that have been diagnosed by the fault manager associated with this managed system element that are still present in that system. This table is indexed by UUID.
sunFmMIB.2	sunFmFaultEventTable	List of individual suspect defects or faults associated with a problem diagnosis, as shown by `fmdump -v -u` *UUID*. This table is indexed by UUID and an integer index ranging from 1 to the number of suspects associated with each diagnosis.
sunFmMIB.3	sunFmModuleTable	List of modules configured in `fmd`(1M).
sunFmMIB.4	sunFmResourceCount	The number of managed elements currently believed by the fault manager to be faulty.
sunFmMIB.5	sunFmResourceTable	Contains information about all faulty resources known to the fault manager. This is the same list provided by the `fmadm faulty` command.
sunFmMIB.7.0	sunFmTraps	Traps or notifications delivered by `snmp-trapgen` in response to fault management events. All traps are compatible with SNMPv1 and SNMPv2.

The problem table contains one `sunFmProblemEntry` for each diagnosis made by the fault manager. Each entry is indexed by UUID and offers the following information about the diagnosis:

TABLE 15–2 `sunFmProblemEntry` elements

OID	Name	Description
sunFmProblemEntry.2	sunFmProblemUUID	The Universal Unique Identifier (UUID) for this problem, as recorded by `fmd`(1M) and shown by `fmadm`(1M) or `fmdump`(1M).

TABLE 15–2 `sunFmProblemEntry` elements *(Continued)*

OID	Name	Description
sunFmProblemEntry.3	sunFmProblemCode	The SUNW-MSG-ID static message identifier for this class of problem, as recorded by `fmd`(1M) and shown by `fmdump`(1M).
sunFmProblemEntry.4	sunFmProblemURL	The URL of an appropriate knowledge base article providing more detailed information about this problem.
sunFmProblemEntry.5	sunFmProblemDiagEngine	The Sun FMRI of the Fault Manager diagnosis engine that performed the diagnosis of this problem, including its version.
sunFmProblemEntry.6	sunFmProblemDiagTime	The date and time at which the problem was diagnosed.
sunFmProblemEntry.7	sunFmProblemSuspectCount	The number of individual suspect defects or faults associated with this problem diagnosis, as shown by `fmdump -v -u` *UUID*.

The fault event table is effectively a part of the problem table. Logically, all `sunFmFaultEventEntry` objects indexed by a given UUID are a part of the problem table entry for that UUID. The `sunFmProblemSuspectCount` value indicates the number of separate fault events associated with the diagnosis and is the maximum secondary index into that diagnosis's entries in this table.

TABLE 15–3 `sunFmFaultEventEntry` elements

OID	Name	Description
sunFmFaultEventEntry.3	sunFmFaultEventProblemUUID	UUID of the problem diagnosis associated with this event. An event may appear multiple times in association with different diagnoses.
sunFmFaultEventEntry.4	sunFmFaultEventClass	Sun Fault Management event class string.
sunFmFaultEventEntry.5	sunFmFaultEventCertainty	Percentage likelihood associated with this suspect for this diagnosis.

TABLE 15–3 `sunFmFaultEventEntry` elements *(Continued)*

OID	Name	Description
sunFmFaultEventEntry.6	sunFmFaultEventASRU	Sun FMRI of the *Automated System Reconfiguration Unit (ASRU)* that is believed to contain the specified fault or defect.
sunFmFaultEventEntry.7	sunFmFaultEventFRU	Sun FMRI of the *Field Replaceable Unit (FRU)* that should be replaced in order to repair the specified fault or defect.
sunFmFaultEventEntry.8	sunFmFaultEventResource	Sun FMRI of the underlying resource that has been diagnosed as faulty or defective.

The module table provides configuration information equivalent to the output of the `fmadm config` command. The table contains one entry for each module, which has a unique integer index. These indices do not change until the SNMP agent is restarted. Each `sunFmModuleEntry` is defined as follows:

TABLE 15–4 `sunFmModuleEntry` elements

OID	Name	Description
sunFmModuleEntry.2	sunFmModuleName	Name of the fault management module.
sunFmModuleEntry.3	sunFmModuleVersion	Version string associated with the fault management module.
sunFmModuleEntry.4	sunFmModuleStatus	Current status of the fault management module.
sunFmModuleEntry.5	sunFmModuleDescription	A text description of the fault management module

The resource table provides information comparable to that offered by the `fmadm faulty` command. It consists of one `sunFmResourceEntry` for each resource, defined as follows:

TABLE 15–5 `sunFmResourceEntry` elements

OID	Name	Description
sunFmResourceEntry.2	sunFmResourceFMRI	Sun FMRI of the ASRU which the fault manager believes to be faulty.

TABLE 15–5 `sunFmResourceEntry` elements *(Continued)*

OID	Name	Description
sunFmResourceEntry.3	sunFmResourceStatus	The current status of the resource, as shows by the `fmadm faulty` command.
sunFmResourceEntry.4	sunFmResourceDiagnosisUUID	The UUID for the problem associated with the fault in this resource, as recorded by `fmd`(1M) and shown by `fmadm`(1M).

All traps and notifications sent by the agent are defined by the MIB. The following traps and notifications are supported:

TABLE 15–6 Traps and notifications

OID	Name	Description
sunFmTraps.1	sunFmProblemTrap	Trap notification that a diagnosis has been made or the fault manager `fmd`(1M) has restarted and the corresponding problem is still believed to be present in the managed entity.

15.3 Properties

The `snmp-trapgen` agent supports the following properties:

TABLE 15–7 `snmp-trapgen` properties

Name	Type	Default	Description
`trap_all`	`boolean`	`false`	If set, emit traps even for events that request no messaging.
`url`	`string`	`http://sun.com/msg/`	The URL to use as a prefix for indicating the location of knowledge articles. This property is provided to permit sites to customize knowledge articles with local procedures as necessary.

◆ ◆ ◆ APPENDIX A

ARC Interface Tables

This appendix lists interfaces imported and exported by the Fault Manager, describes briefly how each interface is versioned, and provides references to other Sun ARC materials that describe these interfaces.

A.1 Imported Interfaces

Interface	Classification	Comments
SMA Configuration Tokens	Evolving	See PSARC 2003/134
SMA Module Developer API	Stable	See PSARC 2003/103
SMA SDK Programming APIs	Evolving	See PSARC 2003/134
Sun FMA Event Protocol	Sun Private	See PSARC 2002/412
`libdiagcode.so.1`	Sun Private	See PSARC 2003/323
`libexacct.so.1`	Evolving	See PSARC 1999/119 and 2000/549. This project also uses and initiated the interfaces defined in 2003/796.
`libnvpair.so.1`	Evolving	See PSARC 2000/212, 2003/121, 2003/252, 2003/355, 2003/587, 2003/590, and 2004/055.
`libsysevent.so.1`	Evolving	The Fault Manager uses only the General Purpose Event Channels interface, described in PSARC 2002/321
`libumem.so.1`	Evolving	See PSARC 2002/088
`libuuid.so.1`	Evolving	See PSARC 2002/094

A.2 Exported Interfaces

Interface	Classification	Comments
`EC_FM, ESC_FM_ERROR`	Sun Private	SysEvent class and subclass definitions for FMA error event transport added to `sys/sysevent/eventdefs.h`
`EXD_FMA_*` catalog tags	Sun Private	Extended accounting tags for log files; see Chapter 5, "Log Files."
`/etc/fm`	Project Private	Directory for editable Fault Management configuration information.
`/etc/fm/fmd`	Project Private	Directory for editable `fmd` configuration information.
`/etc/fm/fmd.conf`	Project Private	Private configuration file for `fmd`. Nothing is shipped here at present; the syntax and parameters are only documented in the PRM and are only intended for use by Sun at present.
`/etc/sma/snmp/mibs/SUN-FM-MIB.mib`	Stable	Fault Management SNMP MIB. See Chapter 15, "`snmp-trapgen` Agent."
`/usr/include/fm`	Sun Private	Directory for Fault Management library include files. Originally introduced in PSARC 2003/323.
`/usr/include/fm/fmd_adm.h`	Contracted Consolidation Private	Interfaces for `libfmd_adm.so.1`. Clients of the programming interface must specify an integer version number in each call to `fmd_adm_open()` in order to permit compatible extension of the data structures used by the interfaces or to establish a compatibility match between client and library.
`/usr/include/fm/fmd_api.h`	Contracted Consolidation Private	Interfaces for `fmd` plug-ins and agents. Clients of the programming interface must specify an integer version number in each call to `fmd_hdl_register()` in order to permit compatible extension of the data structures used by the interfaces or to establish a compatibility match between client modules and `fmd`.
`/usr/include/fm/fmd_fmri.h`	Project Private	Interfaces for `fmd` scheme libraries.

Interface	Classification	Comments
`/usr/include/fm/fmd_log.h`	Contracted Consolidation Private	Interfaces for `libfmd_log.so.1`. Clients of the programming interface must specify an integer version number in each call to `fmd_log_open()` in order to permit compatible extension of the data structures used by the interfaces or to establish a compatibility match between client and library.
`/usr/include/fm/fmd_snmp.h`	Project Private	Interfaces for `libfmd_snmp.so.1`. See Chapter 15, "`snmp-trapgen` Agent" for more information about SNMP support.
`/usr/lib/fm`	Evolving	Directory for Fault Management libraries and event dictionaries. Originally introduced in PSARC 2003/323.
`/usr/lib/fm/dict`	Evolving	Directory for `libdiagcode.so.1` event dictionaries.
`/usr/lib/fm/dict/FMD.dict`	Sun Private	`libdiagcode.so.1` event dictionary for the Fault Manager.
`/usr/lib/fm/fmd`	Evolving	Directory for `fmd` libraries and other read-only configuration information.
`/usr/lib/fm/fmd/fmd`	Evolving	Solaris Fault Manager daemon. The file location and command-line options are intended to be Evolving; all other interfaces inside of and exported by `fmd` are listed separately in this table.
`/usr/lib/fm/fmd/fminject`	Sun Private	Error event injector utility. See Chapter 10, "`fminject` Utility."
`/usr/lib/fm/fmd/fmsim`	Sun Private	Fault manager simulation utility. See Chapter 11, "`fmsim` Utility."
`/usr/lib/fm/fmd/plugins`	Contracted Consolidation Private	Directory for `fmd` plug-in modules. In the future the interfaces to build plug-ins may be offered as Sun Private or Evolving; for now we will be working with specific project teams on a contracted basis only for the introduction of modules.
`/usr/lib/fm/fmd/plugins/ip-transport.so`	Project Private	Reference implementation of an event transport; see Chapter 4, "Event Transports."
`/usr/lib/fm/fmd/plugins/ip-transport.conf`	Project Private	Transport module configuration file; see Chapter 4, "Event Transports."

Interface	Classification	Comments
`/usr/lib/fm/fmd/plugins/snmp-trapgen.so`	Stable	SNMP trap generation module; see Chapter 15, "`snmp-trapgen` Agent."
`/usr/lib/fm/fmd/plugins/snmp-trapgen.conf`	Project Private	SNMP trap generation module configuration file; see Chapter 15, "`snmp-trapgen` Agent."
`/usr/lib/fm/fmd/plugins/syslog-msgs.so`	Evolving	Agent to message `list.suspect` events to `syslogd`(1M); see Chapter 14, "`syslog-msgs` Agent."
`/usr/lib/fm/fmd/plugins/syslog-msgs.conf`	Project Private	Agent configuration file; see Chapter 14, "`syslog-msgs` Agent."
`/usr/lib/fm/fmd/schemes`	Project Private	Directory for `fmd` protocol resource scheme libraries. In the future the interfaces to build schemes may be offered as Sun Private or Evolving; for now only the project team will be able to create new schemes.
`/usr/lib/fm/libfmd_adm.so.1`	Contracted Consolidation Private	Library providing administrative and monitoring interfaces for `fmd`. This library is used to implement the `fmadm` and `fmstat` utilities. Its binary interfaces are versioned under a `SUNWprivate` version label for now.
`/usr/lib/fm/libfmd_log.so.1`	Contracted Consolidation Private	Library providing access to `fmd` log files. This library is used to implement the `fmdump` utility. Its binary interfaces are versioned under a `SUNWprivate` version label for now.
`/usr/lib/fm/libfmd_snmp.so.1`	Stable	NetSNMP (SMA) extension module for Fault Management MIB.
`/usr/lib/mdb/proc/fmd.so`	Project Private	`mdb` debugging module for `fmd`.
`/usr/sbin/fmadm`	Command-line Syntax: Evolving; Human-readable Output: Unstable	Administrator and service tool to examine `fmd` status and perform service tasks.

Interface	Classification	Comments
`/usr/sbin/fmdump`	Command-line Syntax: Evolving; Default Human-readable Output: Evolving; `fmdump -v` Human-readable Output: Unstable; `fmdump -V` Human-readable Output: Sun Private	Administrator and service tool to examine `fmd` error, fault, and resource log files.
`/usr/sbin/fmstat`	Command-line Syntax: Evolving; Human-readable Output: Unstable	Administrator and service tool to examine `fmd` performance statistics and client-specific statistics provided by each plug-in module.
`/var/fm`	Evolving	Directory containing Fault Management state such as writable logs and other environment-specific variable files.
`/var/fm/fmd`	Evolving	Directory containing `fmd` log files, checkpoint files, and resource cache.
`/var/fm/fmd/errlog`	Sun Private	Log file for system error events received over the Solaris error event transport.
`/var/fm/fmd/fltlog`	Evolving	Log file for system `list.suspect` events containing the results of automated diagnosis.
`/var/fm/fmd/ckpt`	Project Private	Directory containing `fmd` checkpoint files.
`/var/fm/fmd/rsrc`	Project Private	Directory containing `fmd` resource cache files.
`/var/fm/fmd/xprt`	Project Private	Directory containing `fmd` transport debugging logs.

Glossary

agent A generic term used to describe fault manager **modules** that subscribe to `fault.*` events or `list.*` events. Agents are used to retire faulty resources, message diagnosis results for humans, and bridge to higher-level management frameworks.

ASRU An Automated System Recovery Unit is a **resource** that can be disabled by software or hardware in order to isolate a problem in the system and suppress further error reports.

buffer A fixed-size persistent region of memory allocated for use by a module that is maintained persistently in the module's checkpoint file. The format and content of each buffer is left up to the module writer. Buffers are kept in two namespaces, one global to the module and one associated with each case, and each buffer is named by a string.

case A metaphor for all of the state that is maintained in the fault manager for a particular problem. A case is named by a Univeral Unique Identifier (UUID) that eventually becomes associated with a `list.suspect` event when the case is *solved*.

diagnosis engine A fault management **module** whose purpose is to diagnose problems by subscribing to one or more classes of incoming error events and using these events to solve **cases** associated with each problem on the system.

error An invalid signal, datum, or result. An error is the *symptom* of a problem on the system, and each problem typically produces many different kinds of errors. Errors may also lead to secondary effects that produce still other kinds of errors. In a traditional computer system, errors are converted to human-readable strings and left in a log file for a human administrator to diagnose. In a self-healing system, errors are converted into events that are sent to a fault manager, where **diagnosis engines** use them to identify the underlying problems.

error event The data structure representing an instance of an error report. Error events are represented as name-value pair lists. Inside of the fault manager, the `libnvpair.so.1` interfaces are using to create and manipulate error events.

error report The data captured with a particular error. Error report formats are defined in advance by creating a **class** naming the error report and defining a schema using the FMA Event Registry.

FMRI A Fault Managed Resource Identifier is a URL-like identifier that acts as the canonical name for a particular **resource** in the fault management system. Each FMRI includes a **scheme** that identifies the type of resource, and one or more values that are specific to the scheme. An FMRI can be represented as a URL-like string or as a name-value pair list data structure.

FRU A Field Replaceable Unit is a **resource** that can be replaced in the field by a customer or service provider. FRUs can be defined for hardware (e.g. a system board) or for software (e.g. a package).

module A fault manager component that receives events and performs some activity in the fault management system, such as diagnosing a particular class of problems, acting on the results of a diagnosis to disable an affected resource, or producing messages for a higher-level management framework.

resource A generic term for some hardware or software object. If a resource is visible to or operated on by the fault management system, it is given a name similar to a URL called a Fault Managed Resource Identifier.

scheme A class of resources that are observed by or acted on by the fault management system. Each scheme has a unique string name assigned by Sun, such as `cpu` or `svc`, and a corresponding plug-in module that is used by the fault manager.

topology snapshot A topology snapshot is a view in-time of resources used in fault management activities (error handling, diagnosis, recovery or repair).

transport A software component that is used to propagate events to and from a fault manager. For example, the SysEvent facility on Solaris is used as a transport for error events flowing from the Solaris kernel to the Solaris Fault Manager.