Data Management Interfaces Group

Interface Specification

for the

Data Management

Application Programming Interface

Draft for Industry Review

Version 2.1

March 1995

NOTE

This is a draft for industry review and is subject to change without notice

Contents

Figures ..... **v**

Tables ..... **vi**

1.0 Introduction **1**

1.1 Related Documents 1

1.2 Revision History 1

1.3 Terms 1

1.4 Document Organization 2

1.5 Intended Audience 2

1.6 Typographical Conventions 2

2.0 Summary **3**

2.1 Events 3

2.1.1 Tokens 3

2.1.2 Access Rights 3

2.2 Managed Regions 4

2.3 Handles 4

2.4 Sessions 4

2.5 Data Management Attributes 5

2.6 Holes 5

3.0 Interfaces **7**

3.1 Initialization 7

3.2 Handles 7

3.3 Sessions 9

3.3.1 Session Instantiation 10

3.3.2 Sessions and Event Messages 10

3.4 Tokens 12

3.5 Access Rights 13

Upgrading Access Rights 14

3.6 Finding Extents and Punching Holes 15

3.7 Invisible Read and Write 16

3.8 Managed Regions 17

3.9 File Attributes and Bulk Retrieval 19

3.10 Data Management Attributes 21

3.10.1 Non-Opaque Data Management Attributes 21

3.10.2 Opaque Data Management Attributes 21

3.11 Events 23

3.11.1 Setting Event Disposition 24

3.11.2 The *mount* Event 25

3.11.3 Setting Event Notification 27

3.11.4 Receiving and Responding to Events 28

3.11.5 Pseudo Events 32

3.12 Configuration Information 33

4.0 Event Types 35
Implementation Responsibilities 36
Interruptible Events 36
Asynchronous Namespace Event Delivery 37
4.1 File system Administration Events 38
4.1.1 mount 38
4.1.2 preunmount 38
4.1.3 unmount 38
4.1.4 nospace 38
4.1.5 debut 39
4.2 Namespace Events 40
4.2.1 create 40
4.2.2 postcreate 40
4.2.3 remove 41
4.2.4 postremove 41
4.2.5 rename 41
4.2.6 postrename 41
4.2.7 symlink 42
4.2.8 postsymlink 42
4.2.9 link 42
4.2.10 postlink 42
4.3 Data Events 43
4.3.1 read 43
4.3.2 write 43
4.3.3 truncate 43
4.4 Metadata Events 44
4.4.1 attribute 44
4.4.2 destroy 44
4.5 Pseudo Events 45
4.5.1 user event format 45
4.6 Event Summary 46
5.0 Data Structures 51
5.1 dm_attrname_t 52
5.2 dm_attrlist_t 52
5.3 dm_attrloc_t 53
5.4 dm_config_t 53
5.5 dm_dispinfo_t 53
5.6 Event message types 53
5.6.1 dm_eventmsg_t 53
5.6.2 dm_data_event_t 54
5.6.3 dm_namesp_event_t 54
5.7 dm_eventset_t 55
5.8 dm_eventtype_t 55
5.9 dm_extent_t 55
5.10 dm_extenttype_t 55
5.11 dm_fileattr_t 56
5.12 DM Handles 56
5.13 dm_inherit_t 56
5.14 dm_msgtype_t 56
5.15 dm_off_t 57
5.16 dm_region_t 57

5.17 Region Flags 57
5.18 dm_response_t 58
5.19 dm_right_t 58
5.20 dm_size_t 58
5.21 dm_ssize_t 58
5.22 dm_sessid_t 58
5.23 dm_stat_t 59
5.24 dm_token_t 61
5.25 dm_vardata_t 61
5.26 Get Attribute Defines 61
5.27 Get Config Defines 61
5.28 Miscellaneous Defines 62

6.0 DMAPI Definitions 63

6.1 Non-iterative Functions 63
6.2 Iterating Functions 64
6.3 DMAPI Macros 65
DMEV_CLR() — removes an event from an event list 65
DMEV_ISSET() — determines if an event is set in an event list 65
DMEV_SET() — adds an event to an event list 65
DMEV_ZERO() — determines if an event is not set in an event list 65
6.4 DMAPI Functions 66
dm_clear_inherit() — clear an attribute's inherit-on-create status 66
dm_create_session() — create a session 67
dm_create_userevent() — generate a user pseudo-event message 69
dm_destroy_session() — destroy the specified session 71
dm_find_eventmsg() — get the message for the event 72
dm_get_allocinfo() — return the current residency information for a file 73
dm_get_bulkattr() — gets bulk attributes for an entire file system 75
dm_get_dirattrs() — gets bulk attributes for a directory 75
dm_init_attrloc() — initializes a bulk attribute location offset 75
dm_get_config() — return DMAPI implementation details and limits 79
dm_get_dmattr() — retrieve a data management attribute 81
dm_get_eventlist() — get the list of enabled events for an object 83
dm_get_events() — get the next available event messages 85
dm_get_fileattr() — return file attributes 87
dm_get_mountinfo() — return the information that was delivered on a mount event 89
dm_get_region() — get the managed regions for a file 91
dm_getall_disp() — get disposition of events for all file systems for a session 93
dm_getall_dmattr() — retrieve all of a file's data management attributes 94
dm_getall_inherit() — get all inheritable attributes for the specified file system 96
dm_getall_sessions() — get all extant sessions 98
dm_getall_tokens() — get all outstanding tokens for a session 99
dm_handle_cmp() — file handle comparison 100
dm_handle_hash() — hashes the contents of a handle 101
dm_handle_to_path() — get an absolute pathname 102
dm_init_service() — perform implementation-defined initialization 103
dm_move_event() — move an event from one session to another 104
dm_fd_to_handle() — creates a file handle from a path descriptor 105
dm_path_to_fshandle() — creates a file system handle from a path name 105
dm_path_to_handle() — creates a file handle from a path name 105
dm_handle_free() — frees the storage allocated for a handle 105
dm_punch_hole() — punchs a hole in a file freeing all associated disk blocks 107
dm_probe_hole() — returns the rounded result of the area where the hole is to be punched 107

dm_query_right() — determine the set of access rights to an object 110
dm_query_session() — query a session for information 111
dm_read_invis() — reads a file bypassing DMAPI events 112
dm_write_invis() — writes to a file bypassing DMAPI events 112
dm_release_right() — release all access rights to an object 114
dm_remove_dmattr() — remove a data management attribute 115
dm_request_right() — request a specific access right to an object 117
dm_respond_event() — respond to an event 119
dm_send_msg() — send a message to the indicated session 121
dm_set_disp() — set the disposition of events on a file system for a session 123
dm_set_dmattr() — create or replace a data management attribute 125
dm_set_eventlist() — set the list of events to be enabled for an object 127
dm_set_fileattr() — set file timestamps, ownership and mode 129
dm_set_inherit() — mark an attribute inheritable 131
dm_set_region() — set the managed regions for a file 133

7.0 Implementation Notes 135

7.1 select(2) and Sessions 135
7.2 Event Encodings 135
7.3 Event Ordering 136
7.4 Lock Releasing 136
7.5 Tokens, Messages, and Handles 136
7.6 mmap 136
7.7 Invisible I/O 136
7.8 Generation of Events 137
7.9 Locking Across Operations 137
7.10 Tokens and Multiple Handles 137
7.11 Structure Lists 137
7.12 Undeliverable Event Messages 138

8.0 Using the Data Management Interfaces 139

8.1 Recovery and Cleanup 140
8.2 Initialization 143
8.3 Dump 145
8.4 Migration 148
8.4.1 Stageout 148
8.4.2 Stagein 149
8.5 Multi-threaded Application 153

Figures

Figure 1: Taxonomy of handles 8
Figure 2: Message states 11
Figure 3: Overlapping of events across managed regions 18
Figure 4: Disposition of event delivery 24
Figure 5: Duplicate event registrations on a file system 25
Figure 6: Mount event propagation 26
Figure 7: Event generation with no rights 29
Figure 8: Requesting access rights after event generation 30
Figure 9: Continuing an event with access rights 30
Figure 10: Interrupting a synchronous message 36

Tables

Table 1: Access right properties for files 14
Table 2: Event summary 46
Table 3: Event generation objects 47
Table 4: Event data structures 48
Table 5: Field use in the dm_namesp_event structure 49
Table 6: Structures used in lists 51
Table 7: Non-iterating function types 63
Table 8: Iterating functions 64

1.0 Introduction

1.1 Related Documents

- *Data Management API: Architecture Overview,* Rev 2.0, March 25, 1993.
- *Data Management Interfaces Group: Introductory Document.*
- *Data Management Interfaces Group: Top Level Requirements*, Rev 1.2, August 1993.
- *Requirements Specification for the Data Management Interface*, Rev 1.6.1, August 1994.
- *Yet Another DMIG Proposal*, John Carmichael, VERITAS Software, January 6, 1994.
- *XPG4.2 - System Interfaces and Headers*, Issue 4, Version 2 (aka Spec 1170).
- *Portable Operating System Interface for Computer Environments*, (POSIX) P1003.1, 1990.

1.2 Revision History

Preliminary versions:

1.0	*February 1994*, Initial revision based on Yet Another DMIG Proposal.
1.0.1	*March 1994*, Revised based on email reflector traffic and conference calls.
1.1	*April 1994*, Revised based on March 1994 meeting.
1.2	*May 1994,* Revised based on April 1994 meeting.
1.2.1	*June 1994,* Revised based on preliminary review.
1.3	*July 1994*, Revised based on June 1994 meeting.
1.3.1	*July 1994,* Revised during July 1994 meeting based on feedback.
1.4	*August 1994*, Revised based on July 1994 meeting.
1.5	*September 1994*, Revised based on review of version 1.4.
1.6	*October 1994,* Revised based on September 1994 meeting.

Draft for Industry Review versions:

2.0	*November 1994,* Revised based on October 1994 meeting.
2.1	*March 1995*, Revised based on review of version 2.0.

1.3 Terms

DMIG	Data Management Interfaces Group. This term refers to the group that is working to standardize the interface between the Unix kernel and data management applications.
DMAPI	Data Management Application Programming Interface. The term that refers to the interface being developed by the DMIG.
DMI	(*outdated*) Data Management Interface. This is an old term referring to the interface being developed by the DMIG. Use of this acronym has been dropped to avoid confusion with the Desktop Management Interface, being developed by the Desktop Management Task Force**.**
DM application	A Data Management application is any application that uses the DMAPI.
event	A notification from the operating system kernel to a DM application about an operation on a file. For example, a DM application can arrange to be notified about attempts to read a particular file.

hole	An area of a file that consists entirely of bytes of zeros. Some file system implementations may not need to consume any media resources to maintain such an area.
managed file	A file that is being monitored by a DM application for events.
managed region	A contiguous span of a file given as an (offset, length) pair, together with an associated event generation specification.
multiple DM apps	In this document, the term *multiple DM apps* refers to multiple, non-cooperating and distinct applications that use the DMAPI. Note that a single product that happens to be constructed using multiple processes is not an example of "multiple DM applications", whereas something like a compression product from vendor A and a backup product from vendor B is.
operating system	In the context of the DMAPI, the code that includes the base kernel as well as any file system or extensions that may have been added to the kernel.
process	A single Unix process. For purposes of this document, a process and a thread are equivalent.
product	A specific DM application from a single vendor. A single product may consist of one or more processes working together.
token	A reference to state associated with a synchronous event message.

1.4 Document Organization

Section two provides a high-level overview of the features and functionality of the DMAPI. Section three outlines the specific interfaces and describes their use. Section four covers the event messages. Section five describes DMAPI data structures. Section six contains man pages. Section seven provides implementation notes intended to highlight subtle or complex areas of the DMAPI definition. Lastly, section eight contains some example uses of the DMAPI.

1.5 Intended Audience

This document is intended for developers writing DMAPI compliant applications or implementing the DMAPI in an operating system. It is assumed the reader has a good understanding of standard Unix semantics, is familiar with the `C` language, and understands the issues and tradeoffs in writing data management software. The reader should also be familiar with the *Requirements Document for the Data Management Interface*, Version 1.6.1. The requirements document presents a better high-level overview of the DMAPI than exists in this Interface Specification.

1.6 Typographical Conventions

`Courier` font is used for the names of all DMAPI functions and data structures.

Italics are used for initial definitions and section names.

`Courier italic` is used for the names of events in the body of the document.

2.0 Summary

To adequately support data management applications, a file system and its host OS must include a set of functions and semantics not found in most POSIX-compliant systems. These extensions include the capability to monitor events on files and special interfaces to manage and maintain the data in a file.

This document describes the Data Management Applications Programming Interface (DMAPI), an interface that satisfies the requirements outlined in the *Requirements Specification for the Data Management Interface, Version 1.6.1.* The reader should refer to this requirements document and the other documents listed in section 1.1 for more information regarding the DMAPI.

The important features of the DMAPI as described in this document are the use of events to notify DM applications about operations on files, the ability of DM applications to store arbitrary attribute information with a file, support for managed regions within a file, and the use of DMAPI access rights to control access to a file object.

Data management applications such as file backup and recovery, file migration, and file replication typically function as logical extensions of the operating system. They often must have special permissions (such as root or file system credentials) to operate. The interfaces and data structures described in this document are all designed for these specific applications. These interfaces are not intended for direct use by typical end-users or unprivileged processes.

2.1 Events

Events and managed regions are the foundations of the DMAPI. In the event paradigm, the operating system informs a DM application running in user space when a particular event occurs. For example, a DM application in user space can request that it be notified whenever a read of a certain area of a file occurs. When the operating system determines that it is going to read data from the target area, it notifies the DM application via a DMAPI event. In this manner, DM applications can monitor and manage files and specific regions within those files.

Files can be divided into distinct portions known as managed regions. Some types of DMAPI events are produced on a per region basis; this allows for finer granularity of event notification and reduces the number of messages between the operating system and the DM application.

2.1.1 Tokens

One of the fundamental concepts of the DMAPI is that of a token. A token is a reference to state associated with a synchronous event message. This state may include lists of files affected by the event, the access rights that are currently in force for those files, and so forth. The token provides DM applications with a simple method of referencing this state. Most DMAPI functions require a token as one of their parameters.

2.1.2 Access Rights

The DMAPI provides DM applications with the ability to control access to a file object. There are three forms of access rights: `DM_RIGHT_NULL`, `DM_RIGHT_SHARED`, and `DM_RIGHT_EXCL`. These rights are mutually exclusive; it is not possible to have a `DM_RIGHT_SHARED` right in effect on an object at the same time as a `DM_RIGHT_EXCL` right.

An application presents a token to the DMAPI to reference and identify a specific access right; in this context, the application is said to be the *holder* of the access right. A DM_RIGHT_NULL right grants no rights to its holder; this is called a *null right*. A DM_RIGHT_SHARED right grants its holder the right to read or query a file system object; this is called a *shared right*. A DM_RIGHT_EXCL right grants its holder all of the DM_RIGHT_SHARED rights plus the right to modify a file system object; this is called an *exclusive right*. As the names imply, there may be multiple DM_RIGHT_SHARED rights granted on an object at one time. There can be only one DM_RIGHT_EXCL right granted, and it excludes DM_RIGHT_SHARED rights as well.

Access rights to a file system object may be conveyed to a DM application as part of sending a synchronous event message. DM applications must use the dm_query_right(), dm_request_right(), and dm_release_right() interfaces to determine the access rights that have been conveyed.

DMAPI access rights provide a mechanism for a DM application to synchronize accesses by other processes. If a token references an outstanding message that conveys the DM_RIGHT_EXCL access right for a file, then only processes presenting that token as an argument to a DMAPI call may access the file; all other attempted accesses to the file (including from within the operating system) are blocked.

These access rights provide a level of abstraction from internal operating system locks. Each OS will have different locking requirements and implementations. Using the DMAPI access rights provides DM applications a consistent interface to the OS, regardless of the internal locking model.

For more information on access rights, tokens, and synchronous messages, see the *Sessions and Event Messages* , *Token Management* , and *Access Right Management* sections.

2.2 Managed Regions

A managed region designates a specific portion of a file that is managed by a DM application. Managed regions allow exception events to be generated only for those parts of the file in which the DM application has interest.

A managed region consists of the following information:

- starting offset in the file.
- length of the region.
- a set of event flags.

Persistence of this information is implementation specific, and can be determined through a DMAPI configuration routine.

2.3 Handles

The DMAPI uses handles to identify file system objects, as described in the *Requirements Document*. All of the interfaces in this document use handles as opposed to file descriptors. A special handle that references a file system as a whole is provided along with another one which allows receiving events when file systems are being mounted.

2.4 Sessions

The interface between the operating system and the DM application is session based. Sessions identify the recipient of an event message, and can also provide the DMAPI implementation with the ability to track, audit, and control the use of DMAPI facilities. Internally, sessions can be used by the

DMAPI implementation as a mechanism to identify the queues that messages are placed on. Sessions are not persistent across reboots.

2.5 Data Management Attributes

DM applications typically use extended attributes to store pertinent information about a file. Attributes in this context are termed *opaque* since these attributes only have meaning to the application that created them. *Non-opaque* attributes provide information that is visible to the DMAPI implementation, such as managed region layout or event lists.

For example, the following are typical examples of opaque extended attributes:

- bitfile IDs.
- location on secondary or tertiary storage.
- indication that portions of the file are nonresident.
- inheritable policy bits such as *lock on magnetic storage*.

Note that the term "extended attributes" in this context does not imply persistent storage associated with the file by the underlying file system. It is the responsibility of the DM application to manage and coordinate these attributes.

The DMAPI provides mechanisms for storage of both opaque and non-opaque persistent (across reboots) per-file attributes. Persistence of these attributes is a DMAPI implementation option; DM applications should use the `dm_get_config()` call to determine what the implementation provides.

2.6 Holes

Holes in a file can be created in two ways. First, as defined by POSIX 1003.1 - 1990:

> The `lseek()` function shall allow the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads of data in the gap shall return bytes with the value zero until data is actually written into the gap.

Holes can also be created by the `dm_punch_hole()` operation. It is the intent of this specification that a file system optimize the representation of holes so that large holes can be represented without consuming media resources. Most hierarchical-storage style DM application implicitly require the ability of a file system to represent at least one hole (covering the entire file) efficiently.

3.0 Interfaces

Many of the interfaces in the following sections accept and return variable length data structures. For example, handles are variable length, as described in the *Handles* section. Event messages are also variable length. For detailed information on accessing individual elements within a data structure, refer to the *Data Structures* and *Events* sections.

3.1 Initialization

An application that needs to use the features and functionality of the DMAPI must first call an initialization routine. This allows implementations of the DMAPI to perform internal initialization procedures before providing service to an application. The DMAPI specification allows undefined behavior if applications do not use the initialization call.

The following routine exists for initializing the DMAPI:

```
int
dm_init_service(void)
```

Perform implementation defined initialization.

3.2 Handles

A *handle* is an opaque identifier for an entity manipulated by the DMAPI. There are three fundamental categories of handles; the *global handle*, *file system handles*, and *object handles*.

The *global handle* is a fixed constant of the implementation, and is used primarily in the `dm_set_disp()` call when setting event disposition for the *`mount`* event. There is exactly one global handle in any DMAPI implementation.

File system handles are used by many DMAPI routines to identify a file system. There is one file system handle per file system. They are persistent and unique over time (per host).

Object handles are the most common; they are the handles used to represent all types of file system objects. Object handles are persistent and unique over time (per host). They are governed by the following properties:

- A unique object handle shall exist for each object visible in the file system name space. Each object handle shall be unique within at least the context of a single host.
- A unique object handle shall exist for each file system object not visible in the name space, such as unlinked but still-open files.
- Object handles must meet the following specific requirements with regard to uniqueness:
    - Object handles shall remain valid across any number of system reboots as long as the corresponding file system object exists.
    - Object handles shall remain valid if the file object they identify is renamed within a file system.
    - Object handles shall remain valid regardless of any operations on the file object they identify. The only time an object handle becomes invalid is when the object it identifies ceases to exist.
- Object handles may be reused, as determined by the rules specified in the Requirements Document.

Some interfaces can only operate on object handles representing a specific type of file system object. For example, `dm_write_invis()` can only be used to write to a regular file. When necessary to

describe restricted forms of object handles, the terms *file handle*, *directory handle*, *symlink handle*, etc., are used in this document. The taxonomy of handle terminology is shown in the following figure.

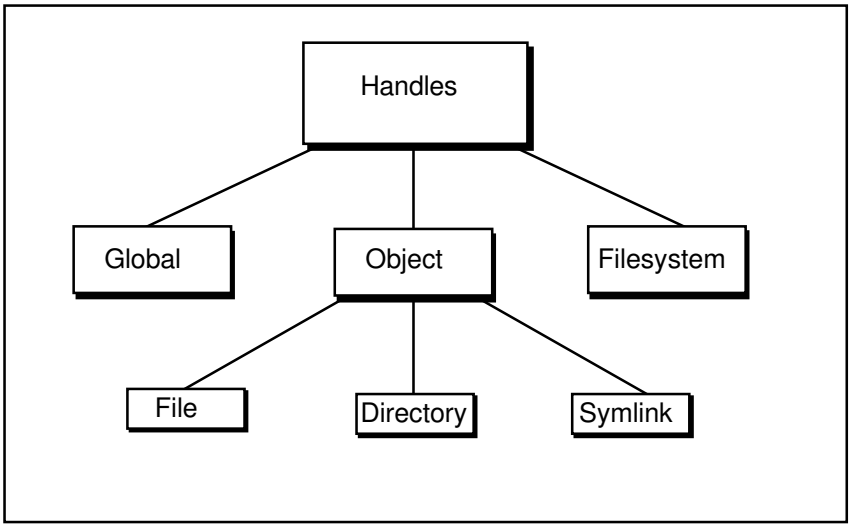

Figure 1: Taxonomy of handles

Handles are opaque; the length of a handle is implementation defined. DM applications should make no assumptions about the length of a handle, as handles may be differing lengths even on the same file system in some DMAPI implementations. Therefore, routines that use handles in their interfaces specify two parameters; a `void *` that provides access to the actual handle, and a `size_t` that specifies the length of the handle. The DMAPI implementation allocates space for handles via the `dm_path_to_handle()`, `dm_fd_to_handle()`, and `dm_path_to_fshandle()` routines. When a DM application is finished with the handle, it should free the space via the `dm_handle_free()` call.

Most DMAPI functions take a handle as part of their interface; many events also provide a handle as part of their event-specific data. To convert from pathnames and file descriptors to handles, a number of routines are provided, as described in the man pages. They are outlined below:

```
int
dm_path_to_handle(
   char            *path,
   void           **hanpp,
   size_t          *hlenp)
```

Return an object handle given a path name.

```
int
dm_fd_to_handle(
   int              fd,
   void           **hanpp,
   size_t          *hlenp)
```

Return a object handle given a file descriptor.

```
int
dm_path_to_fshandle(
   char            *path,
   void           **hanpp,
   size_t          *hlenp)
```

Return the file system handle given a path name.

```
int
dm_handle_cmp(
   void     *hanp1,
   size_t    hlen1,
   void     *hanp2,
   size_t    hlen2)
```

Compare two handles for equality.

```
void
dm_handle_free(
   void     *hanp,
   size_t    hlen)
```

Free the space consumed by the indicated handle.

```
u_int
dm_handle_hash(
   void     *hanp,
   size_t    hlen)
```

Hashes the contents of a handle.

```
int
dm_handle_to_path(
   void     *dirhanp,
   size_t    dirhlen,
   void     *targhanp,
   size_t    targhlen,
   size_t    buflen,
   char     *pathbufp,
   size_t   *rlenp)
```

Take two object handles, a directory handle and a target object handle, and return an absolute pathname for the target object handle, provided one exists.

3.3 Sessions

A session ID is of type `dm_sessid_t`. The session ID is used to identify the recipient of an event message. Session IDs are opaque to DM applications.

Sessions can be thought of as message queues. The implementation of the DMAPI enqueues messages on a session to make them available to a DM application. Sessions provide a mechanism for a DM application to receive events. Sessions are also the cornerstone of the recovery mechanism.

Sessions are governed by the following restrictions:

- sessions are not persistent across reboots.
- sessions are unique for as long as the system is up; they are not unique across reboots.

- sessions must be explicitly destroyed. If a process exits or otherwise aborts without doing a `dm_destroy_session()`, then the behavior of the session is dictated by the constraints in the *Sessions and Event Messages* section.
- A process that has successfully executed `dm_init_service()` can use any session id that the process can access.
- session IDs should not be interpreted as file descriptors. The underlying implementation may use file descriptors, but the DM application should make no assumptions about the implementation.

3.3.1 Session Instantiation

A session must be created via `dm_create_session()` before a DM application can communicate with the DMAPI. When a session is created, it is possible to specify a previous instantiation of a session that will then be *assumed* (taken over), which is useful for recovery purposes. The `dm_create_session()` call is atomic; if the call succeeds, the DMAPI guarantees that all old messages that were enqueued on the old session are now part of the new session. When assuming an existing session the old session is invalid when the call returns.

To shut down and destroy a session, a DM application may have to perform a number of operations, such as ensuring that no more events will be generated, responding to outstanding messages, and so forth. If a DM application attempts to destroy a session that has outstanding event messages still enqueued, an error will be returned. It is assumed that `dm_destroy_session()` will only be called after the application has ensured that no more events will be generated on the session.

Examples of creating and destroying sessions are given in the *Using the Data Management Interfaces* section.

The following routines are provided for manipulating an instantiation of a session:

```
int
dm_create_session(
   dm_sessid_t     oldsid,
   char           *sessinfop,
   dm_sessid_t    *newsidp)
```

Create a new session; an old session may also be assumed.

```
int
dm_destroy_session(
   dm_sessid_t    sid)
```

Destroy the specified session.

3.3.2 Sessions and Event Messages

At any time, a session may have *synchronous* event messages that are in one of two states:

- enqueued, undelivered.
- delivered and awaiting a response from the DM application.

From the standpoint of the DMAPI implementation, synchronous event messages that are in the second state (delivered and awaiting a response) are *outstanding*. *Asynchronous* messages do not require a response from the DM application, and therefore will never be in the outstanding state.

In the following figure, there are three event messages on the session. Each event message is identified by a unique token. One synchronous event message has been delivered to a DM application, and therefore the session has an outstanding message. The event message will continue to exist until some DM application responds to it. The two other event messages on the session are just enqueued and have not yet been delivered to a DM application.

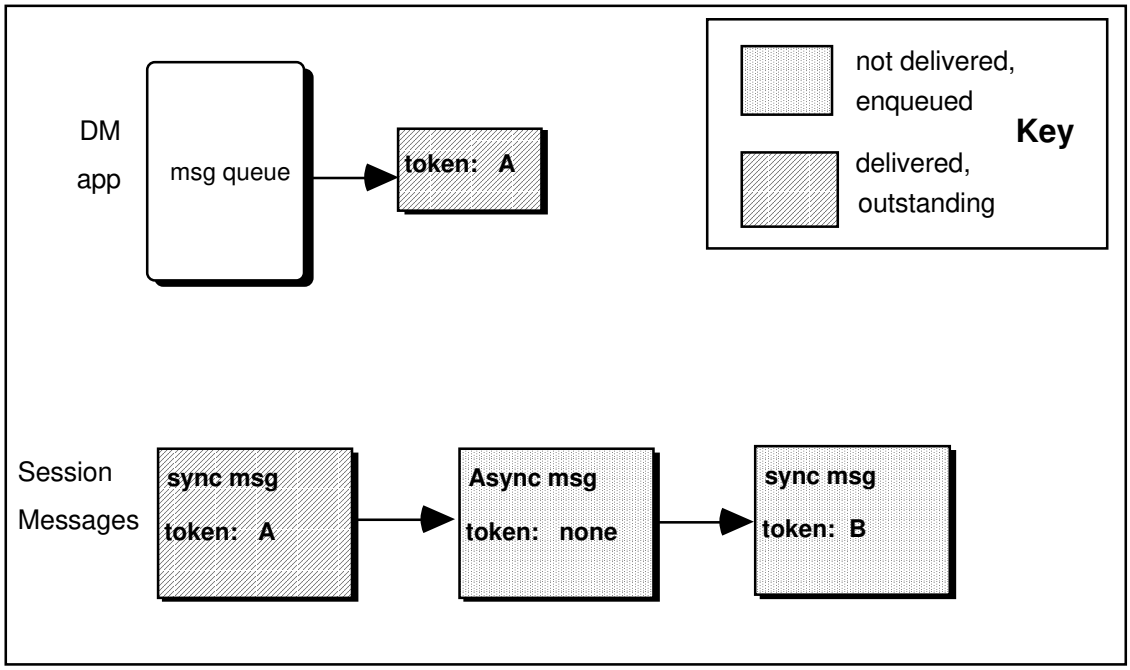

Figure 2: Message states

As part of sending a synchronous event message, the implementation of the DMAPI may convey access rights to one or more objects in the message. If a DM application fails (dies, hangs, or otherwise malfunctions), a recovery process must determine the outstanding event messages and take care of the associated events to prevent the system from hanging. Since tokens are tied to a session, and are always associated with a synchronous event message, it is possible to obtain all outstanding event messages simply by knowing all the tokens.

An active session is not needed to obtain the list of all valid sessions in the system. This allows a recovery application to interrogate all sessions even in the unlikely event the system runs out of sessions.

Recovering after a DM application failure is very different from recovering from a system crash. The requirements of each individual DM application will be unique with respect to recovery from a system crash; it is beyond the scope of the DMAPI to provide all the tools for a DM application to recover itself in this instance.

The following interfaces exist to manage session and event message recovery.

```
dm_getall_sessions(
   u_int           nelem,
   dm_sessid_t    *sidbufp,
   u_int          *nelemp)
```

Obtain all valid sessions in the system.

```
dm_query_session(
   dm_sessid_t     sid,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Retrieve the information associated with the session when it was created.

```
dm_getall_tokens(
   dm_sessid_t     sid,
   u_int           nelem,
   dm_token_t     *tokenbufp,
   u_int          *rlenp)
```

Get all tokens for all outstanding event messages for a session.

```
dm_find_eventmsg(
   dm_sessid_t     sid,
   dm_token_t      token,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Get the message for an event.

3.4 Tokens

Tokens are a reference to state associated with a synchronous event message. They are always associated with one and only one synchronous event message. When responding to an event message, the same token that was delivered with the message must be supplied. Tokens can be thought of as the identifier that a DM application must use to reference a synchronous event message; the DM application presents the token to the DMAPI and in return, is provided with the state associated with the event message.

Like session IDs, tokens are opaque to DM applications. There is no special security expressed or implied by the possession or use of a token. If a DM application can 'guess' the value of a token, then it can use it (assuming that it can supply the appropriate session ID and has other system-dependent privileges).

Tokens are governed by the following restrictions:

- Tokens are not *owned* by any particular process.
- The DMAPI does not mandate authentication or authorization of the process using the token; if a process knows a token's value, it can use it.
- Tokens are meaningful only within the session that they were created under.

3.5 Access Rights

There are two primary rights; `DM_RIGHT_SHARED` and `DM_RIGHT_EXCL`. The third access right, `DM_RIGHT_NULL`, is not considered a primary access right, since it conveys no rights to an object.

Synchronous event messages may convey access rights to one or more object handles. Some event messages contain multiple file handles. The event message **may** convey access rights to all the files in the event message; the DM application must use `dm_query_right()` to determine what rights for the given file handles, if any, are present in the message.

If a DM application needs to obtain access rights for more than one handle, it can use the same token in repeated calls to `dm_request_right()` and `dm_release_right()`. It is not necessary for a new message (and its corresponding token) to be created via `dm_create_userevent()` for each handle the DM app needs to acquire access rights to.

As already noted, tokens do not belong to any particular process. An application presents a token to the DMAPI to reference and identify a specific access right. When a DM application is informally described as "holding a token" or "obtaining an access right", a more precise description would be that an outstanding access right exists, is encapsulated within a synchronous event message, and is identified by a specific token.

Many DMAPI routines require a token that references a specific access right to an object. In some cases, it may be advantageous for a DM application to not have to go through the steps of explicitly creating a token and acquiring the necessary access rights just to call a DMAPI routine. Therefore, many routines accept either a token that references the required rights, or the special value `DM_NO_TOKEN`, that indicates no token is present in the interface.

If a DM application does not pass a token to a DMAPI routine that normally requires a token as one of its parameters, then the routine will acquire the appropriate rights automatically on behalf of the DM application. In this case, the DM application must be willing to be blocked. The DM application may or may not be blocked interruptibly, depending on the implementation of the DMAPI; see `dm_request_right()` for more information.

The DM application must use caution when availing itself of this optimization. If a DM application holds a token that references a right to an object, but fails to present it when calling a DMAPI routine, then the application is in danger of deadlocking with itself. This is because the DMAPI routine will not be able to acquire the necessary rights on behalf of the DM application since the application already holds a token referencing those rights. The DM application should also not use this method of acquiring access rights if it is receiving synchronous events via `dm_get_events()`. Since one of the synchronous event messages may contain a token that references an access right the DM application may be trying to obtain, the application will again deadlock with itself.

The existence of any outstanding `DM_RIGHT_SHARED` access rights for a file system object will block **all** attempts from all processes performing the following operations:

- all data modification, such as via `write(2)`.
- object destruction.

The existence of the `DM_RIGHT_EXCL` access right will block all attempts to perform any operation on the file system object, with the sole exception of the `stat(2)` family (`stat`, `lstat`, `fstat`, etc.).

The locking properties of access rights are summarized in Table 1.

Access Right	Blocked Operations
DM_RIGHT_SHARED	data write, object destruction
DM_RIGHT_EXCL	all but stat(2)

Table 1: Access right properties for files

Notice that the above descriptions do not say that ***other*** processes are blocked; they say that ***all*** processes are blocked. This is where the distinction that DM applications do not really "own" access rights comes into play.

The only way a DM application can distinguish itself from other processes that should be blocked is by knowing the dm_token_t value identifying the appropriate token, and passing it in with any operations that are to be performed on the file. It follows from this that *once a DM application has "obtained" a DM_RIGHT_SHARED or DM_RIGHT_EXCL access right***,** either directly via a dm_request_right() call or indirectly via an event message, *the DM application must be extremely cautious when performing operations on file system objects. Generally, it must restrict itself to using interfaces containing dm_token_t parameters.*

For example, calling dm_request_right() and requesting DM_RIGHT_EXCL does not make a DM application the owner of the right; dm_request_right() merely creates the right and encapsulates it in the synchronous event message referenced by the token. Once that happens, all operations against the file system object will be blocked as described above, even if they come from the same process that called dm_request_right(). Only operations that are part of the DMAPI and contain dm_token_t arguments are safe for DM application to call at this point, because those interfaces are the only way DM application can distinguish themselves as "owning" the DM_RIGHT_EXCL right.

Upgrading Access Rights

When requesting access rights to an object via dm_request_right(), the requested right may not be immediately available. If the DM application has specified that it wants to block until the right becomes available, the DM app may or may not be blocked interruptibly. The implementation of the DMAPI will specify the semantics for interrupting blocked processes.

If a DM application holds a DM_RIGHT_SHARED access right, it can attempt to upgrade the right to a DM_RIGHT_EXCL in a non-blocking manner via dm_request_right(). If the DMAPI implementation cannot grant the request, however, the DM application will most likely have to release the DM_RIGHT_SHARED right, and request DM_RIGHT_EXCL access to the object via dm_request_right() in a blocking fashion.

While the DM application is waiting for the requested exclusive right, it is quite possible that the file may change state. To provide some indication that the file changed while the application was blocked, the DMAPI provides the notion of a *change indicator* that can be interrogated via dm_get_fileattr(). This change indicator is modified by any operation that modifies file data or metadata. The change indicator is not persistent and has no meaning across reboots. Its only purpose is to indicate to the DM application that the file **may** have changed since the last time the change indicator was interrogated.

The normal sequence of events for attempting a lock upgrade where the current shared lock must be dropped would be as follows:

1. Obtain current change indicator.
2. Release shared right.
3. Request exclusive right (blocking operation).
4. Obtain new change indicator to see if the file has changed.

The following routines for manipulating access rights are provided:

```
int
dm_request_right(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   int             nowait
   dm_right_t      right)
```

Request a specific access right to an object.

```
int
dm_release_right(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token)
```

Relinquish the access rights to an object.

```
int
dm_query_right(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_right_t     *rightp)
```

Determine the set of access rights to an object.

3.6 Finding Extents and Punching Holes

Data Management applications often need to release the on-disk blocks of a file to free up space on a file system. Likewise, if a large, but sparsely populated file is to be backed up efficiently, a DM application needs to know where the file has non-null data and where the file has holes. These operations may not be supported on all file system types; `dm_get_config()` can be used to determine if the underlying file system supports punching holes.

The DM application will be responsible for maintaining accurate information about the location of any holes in the original file when a sparse file is made non-resident. It is assumed that the DM application will call `dm_get_allocinfo()` to determine where actual storage is located, and only perform `dm_read_invis()` operations on the portions of the file that contain data.

The following routines return information about a file in terms of a `dm_extent` structure, as defined in the *Data Structures* section. These routines, which do not affect any of a file's time stamps, are provided for managing the storage space for a file:

```
int
dm_get_allocinfo(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_off_t       *offp,
   u_int           nelem,
   dm_extent_t    *extentp,
   u_int          *nelemp)
```

Return the residency information for the file specified by the handle.

```
int
dm_probe_hole(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_off_t        off,
   dm_size_t       len,
   dm_off_t       *roffp,
   dm_size_t      *rlenp)
```

Interrogate the DMAPI implementation for size and offset around the area that the DM applications want to punch a hole.

```
int
dm_punch_hole(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_off_t        off,
   dm_size_t       len)
```

Logically write zeroes in the indicated region of the file identified by the handle, thereby allowing the DMAPI implementation to release media resources associated with that region. None of the file's timestamps are updated, but the file's DMAPI change counter will be updated.

3.7 Invisible Read and Write

Many data management applications must be able to access file data without altering the file's access, modification, and change times, and without generating any events. The operations in this section do not trigger events; they bypass the normal event delivery mechanism to prevent a DM application from receiving events generated by itself.

The following routines, which do not affect any of a file's time stamps, are provided:

```
dm_ssize_t
dm_read_invis(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_off_t        off,
   dm_size_t       len,
   void           *bufp)
```

Do a read without updating any of the file's time stamps.

```
dm_ssize_t
dm_write_invis(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_off_t        off,
   dm_size_t       len,
   void           *bufp);
```

Do a write without updating any of the file's time stamps. The DMAPI change indicator will be updated.

3.8 Managed Regions

Managed regions provide a mechanism for a data management application to control a specific region of a file. Managed regions provide granularity finer than the entire file for data events such as read and write. Their use is particularly important for very large files that may be larger than the actual amount of available disk space.

A single managed region is represented by a `dm_region` structure. The set of managed regions for a file will be a collection of these structures. See the *Data Structures* section for a definition of this structure.

The generation of events for a managed region is controlled by a flags field in the `dm_region` structure. The possible values for this field are a bitwise `OR` of one or more of the following:

`DM_REGION_READ`	Generate a synchronous event for a *read* operation that overlaps this managed region.
`DM_REGION_WRITE`	Generate a synchronous event for a *write* operation that overlaps this managed region.
`DM_REGION_TRUNCATE`	Generate a synchronous event for a *truncate* operation that overlaps this managed region.

or the following value:

`DM_REGION_NOEVENT`	Do not generate any events for this managed region.

The events defined above are the only synchronous data events that are defined for a managed region. Only one of the above events will be produced for a particular read/write/truncate operation no matter how many managed regions the operation may overlap.

The example below shows a read operation that overlaps two managed regions that have read events set.

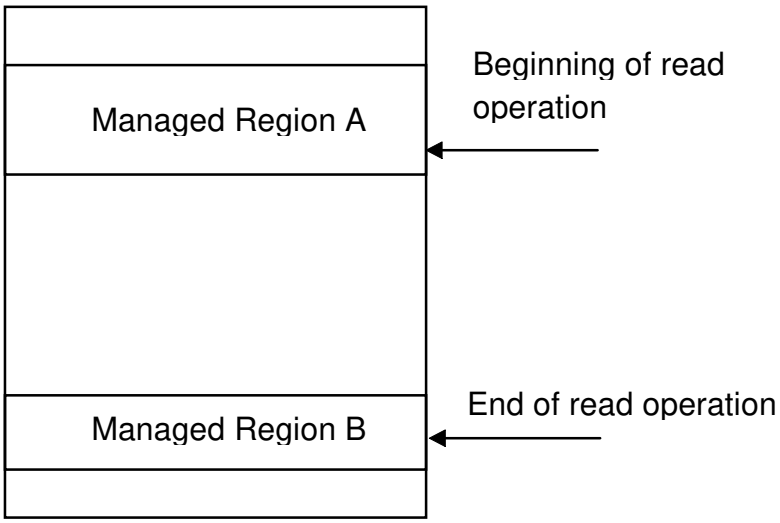

Figure 3: Overlapping of events across managed regions

In figure three, a *read* event will be produced for managed region A. The arguments passed to the DM application in the event message will have the offset and length of the read operation; it will be up to the DM application to determine which managed regions the operation will overlap. Once the DM application responds to the event message, the DMAPI implementation will allow the read to continue.

As an example, if a DM application fills the managed region A above, but not B, and continues the operation, the behavior of the entire read operation is undefined.

Rationale:

Triggering one event per file operation eliminates the necessity of having the DMAPI implementation re-evaluate all managed regions involved in a given operation. Otherwise, the DMAPI implementation could be forced to generate multiple events per managed region for a single I/O operation.

To change the set of managed regions, the DM application must obtain `DM_RIGHT_EXCL` rights to the object. Since managed regions may or may not be persistent, the DM application must be prepared to expect a *debut* event and to use `dm_set_region()` to download the set of managed regions for a file.

To keep the implementation effort in line with the time frame discussed in the Requirements Document, managed regions may be constrained by the following restrictions in the first phase of the DMAPI:

- Implementations may choose to support only one managed region per file, which may always be the entire file.
- Managed regions may not overlap. Each region is a distinct subset of the file.
- Only regular files may be partitioned into multiple managed regions.

A DM application can determine the properties of the DMAPI managed region implementation by consulting the `dm_get_config()` interface.

The following routines, which do not affect any of a file's time stamps, are provided for manipulating the managed regions of a file:

```
int
dm_get_region(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           nelem,
   dm_region_t    *regbufp,
   u_int          *nelemp)
```

Return the set of managed regions for a file.

```
int
dm_set_region(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           nelem
   dm_region_t    *regbufp,
   u_int          *exactflagp);
```

Set the managed regions for a file. The DMAPI change indicator will be updated.

3.9 File Attributes and Bulk Retrieval

Attributes need to be retrieved for a single file, a directory or a whole file system. The attributes returned are defined by the `dm_stat` structure. There are a number of methods for obtaining these file attributes:

- The `dm_get_fileattr()` function obtains the attributes for a single file specified by the file's handle.
- The attributes and names for all files in a directory can be obtained through use of the `dm_get_dirattrs()` function.
- The attributes for all files in a file system can be obtained through use of the `dm_get_bulkattr()` function.

For the second and third methods, the application either provides a buffer large enough to contain all retrieved attributes or more commonly (particularly for the last option) the application makes iterative calls through the interface. A file system must be mounted to have its attributes retrieved via any of the above methods.

DM applications often need to **set** a file's metadata to specific values transparently. For example, a backup application might want to set a file's timestamps to their original value when the file is restored. Specific fields from the `dm_stat` structure are encapsulated in the `dm_fileattr` struct; this structure is used to set various metadata fields to specific values via `dm_set_fileattr()`.

Before calling `dm_get_bulkattr()` and `dm_get_dirattrs()`, the DM application must initialize an opaque *cookie* that provides location information to the DMAPI. Each call of `dm_get_bulkattr()` or `dm_get_dirattrs()` can use this cookie to determine location information from one call to the next.

The file's change indicator can also be retrieved using `dm_get_fileattr()`. This change indicator is modified by any operation that modifies file data or metadata. DM applications can use the change indicator to determine if a file may have changed state; if the indicator is the same between two calls, the file is guaranteed not to have changed. If the indicator is different, the file

may (but not necessarily) have changed. This is especially useful for attempting lock upgrades, as described in the *Upgrading Access Rights* section.

The following calls, which do not update any of an object's time stamps, are provided for obtaining bulk attributes:

```
int
dm_get_bulkattr(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           mask,
   dm_attrloc_t   *locp,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Get the specified attributes in bulk for the given file system.

```
int
dm_get_dirattrs(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           mask,
   dm_attrloc_t   *locp,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Get the specified file attributes and names in bulk for the given directory.

```
int
dm_init_attrloc(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_attrloc_t   *locp)
```

Initialize the location cookie for successive dm_get_bulkattr() calls.

The following routine, which does not update any of a file's timestamps, is provided for obtaining the attributes of a single file:

```
int
dm_get_fileattr(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           mask,
   dm_stat_t      *statp)
```

Get the specified attributes for the given object.

The following routine, which does not update any of the file's timestamps (other than those specified) as a side effect, is provided for metadata modification:

```
int
dm_set_fileattr(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           mask,
   dm_fileattr_t  *attrp)
```

Set a specified attribute to a given value.

3.10 Data Management Attributes

Support for *persistent* data management attributes is a DMAPI implementation option. Some DMAPI implementations may not support persistent opaque data management attributes, while others may not provide support for persistent non-opaque attributes such as event lists. DM applications should use the `dm_get_config()` call to determine what the implementation provides.

3.10.1 Non-Opaque Data Management Attributes

There are two types of non-opaque attributes:

Managed Regions	The DMAPI implementation may support persistence of managed regions. The dm_get_config() call returns the number of persistent managed regions supported.
Event Bit Masks	Event bit masks encode which events are enabled for a particular file within a finite number of persistent bits.

3.10.2 Opaque Data Management Attributes

The DMAPI persistent opaque attribute mechanism provides a set of (name, value) pairs associated with a file system object. The name is a fixed length (64 bits) opaque value determined by the DM application. The value is variable length, and also opaque. These attributes are persistent across reboots. Any operation which manipulates attributes does not modify the file's traditional time stamps (`atime`,`mtime`,`ctime`).

For opaque attributes, a finite number of attributes may be stored persistently with each file. Each attribute is guaranteed a minimum size of 32 bytes for storage of its data; however, the maximum space available for storage of a single attribute's data for an individual file is limited to `DM_CONFIG_MAX_ATTRIBUTE_SIZE`, as returned by `dm_get_config()`. The DMAPI provides a lower bound of 256 bytes for `DM_CONFIG_MAX_ATTRIBUTE_SIZE`. The total amount of space available for storage of **all** persistent attributes on a file system is bounded by `DM_CONFIG_TOTAL_ATTRIBUTE_SPACE`.

Associated with the file attributes is a per-file time stamp that is changed when attributes are created, modified, deleted. The time stamp is also changed when a new file is created that inherits attributes from its parent directory. For those operations that return file attributes, a value of zero will be returned if a file has no associated attributes.

Data Management applications can mark attributes as inheritable. If a directory has an attribute (such as *lock_on_magnetic*) which has been marked inheritable and a file is created in the directory, then the file would inherit the attribute. Attributes that are not marked inheritable are not copied.

DM applications mark an attribute inheritable on a per file system basis and for specified file types. For example, a DM application could mark the above attribute (*lock_on_magnetic*) inheritable for newly created regular files only. Newly created directories would not inherit the attribute.

Attribute inheritance is not persistent across reboots. If a DM application marked the *lock_on_magnetic* attribute as inheritable and the system were then brought down, the attribute would no longer be inheritable when the system came back up.

The following routines are provided for attribute management:

```
int
dm_set_dmattr(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_attrname_t  *attrnamep,
   int             setdmtime,
   size_t          buflen,
   void           *bufp)
```

Create a persistent attribute or override the contents of an existing attribute.

```
int
dm_getall_dmattr(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Retrieve all of the DM attributes and their values for an object.

```
int
dm_get_dmattr(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_attrname_t  *attrnamep,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Retrieve the value of a specific attribute.

```
int
dm_remove_dmattr(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
```

```
    dm_token_t      token,
    int             setdmtime,
    dm_attrname_t  *attrnamep)
```

Remove a specific attribute from an object.

The following routines are provided for managing inheritable attributes:

```
int
dm_set_inherit(
    dm_sessid_t     sid,
    void           *hanp,
    size_t          hlen,
    dm_token_t      token,
    dm_attrname_t  *attrnamep,
    mode_t          mode)
```

Mark an attribute as inheritable on a file system.

```
int
dm_clear_inherit(
    dm_sessid_t     sid,
    void           *hanp,
    size_t          hlen,
    dm_token_t      token,
    dm_attrname_t  *attrnamep)
```

Mark an attribute as no longer being inheritable.

```
int
dm_getall_inherit(
    dm_sessid_t     sid,
    void           *hanp,
    size_t          hlen,
    dm_token_t      token,
    u_int           nelem,
    dm_inherit_t   *inheritbufp,
    u_int          *nelemp)
```

Get all the attributes that have been marked inheritable on a file system. This is especially useful for application restart after a fail.

3.11 Events

The DMAPI provides DM applications with the ability to monitor and manage the data in a file system without having to export all the file system semantics from kernel space to user space via the event interface. Events are generated by a file system, and then the messages are enqueued on a session for delivery to a DM application.

The intent of the DMAPI is to support a single product on any single file system. The DMAPI does not preclude different products from different vendors operating on the same file system, but it is not recommended. Different products on different file systems are fully supported by the DMAPI with regard to event delivery.

Therefore, the following event restrictions exist:

- Multiple sessions can not receive the same event on the same file, even if the event is asynchronous. This is primarily to avoid problems with the paired namespace events (sync before / async after) from being delivered to two *different* sessions.
- Event messages are targeted to and enqueued on sessions; there is no *explicit* targeting of an event to a specific process.
- The behavior of event delivery when no session has requested to receive a particular event (i.e., `dm_set_disp()` with the given event has not been executed) is event specific. Each event will specify the disposition of undeliverable events (see the discussion in the *Event Types* section of the specific event for additional information).

3.11.1 Setting Event Disposition

After creating a session, DM applications must register with the DMAPI to establish the disposition of events for the file system (i.e., what session the events will be sent to). The event list is the complete set of all events, including managed region events, that the DM application will be monitoring during the life of the session. Since registration is on a per-session basis, this event list is not persistent across reboots. It is not possible to register to receive events on anything other than the file system object.

Once a DM application has registered its event list and session with the DMAPI, it can begin receiving event messages on a file system. Registration can be thought of as establishing the association between a file system and a session, as it lets the DMAPI implementation know which session to send specific event messages to.

The example shown in figure four illustrates the case where a DM application has registered with the file system represented by `foo` for the *`read`* and *`write`* events. The event messages will be delivered to the application via session 42. The file `bar` has an event list of *`read`*, *`write`*, and *`truncate`* that was previously set via `dm_set_eventlist()`.

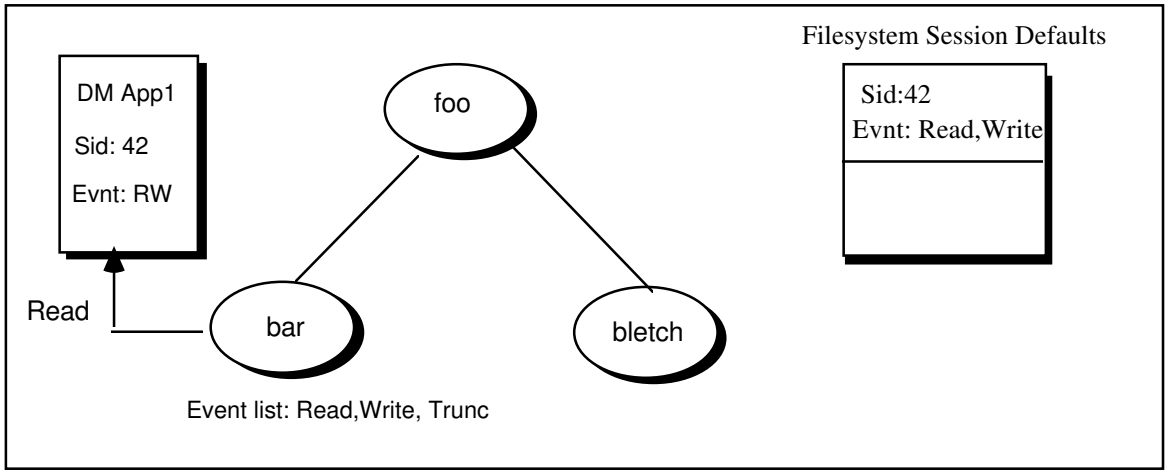

Figure 4: Disposition of event delivery

In figure four, the *`read`* event is delivered to DM app one, since that is the session for that specific event.

Multiple applications can register their session and event list for a file system. If two applications attempt to register to receive the same event, the last application to register for the event will receive it; prior registrations for the event are replaced.

Rationale:

If this were not the case, and replacement were done on an entire event list, not a per-event basis, then it would not be possible to have more than one active session registered for a file system. Having each event in the event list handled individually allows multiple applications to be active on the same file system simultaneously, all handling different events.

The following diagram illustrates how figure four would change if a second DM application registered for just the *read* events.

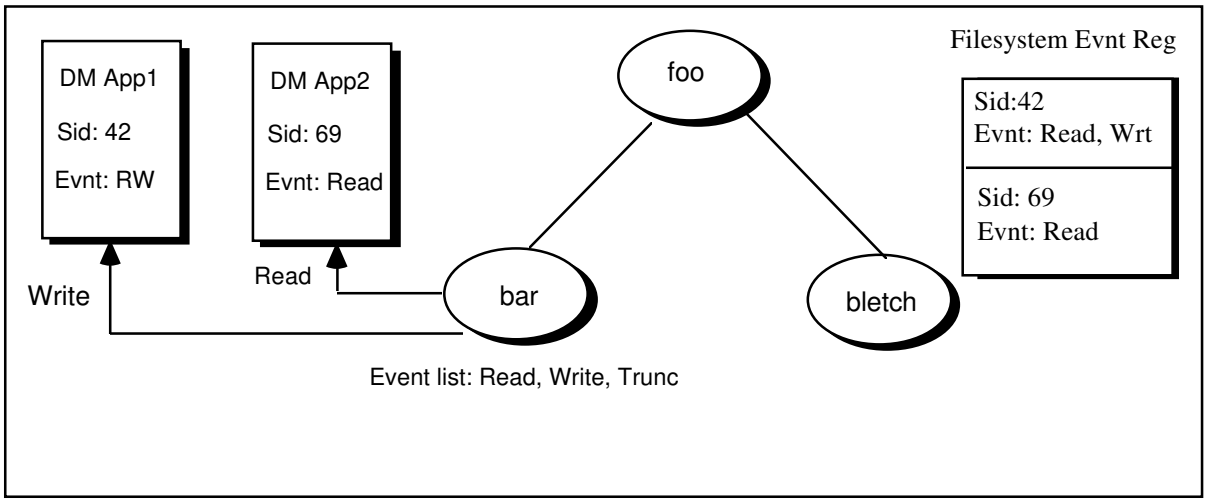

Figure 5: Duplicate event registrations on a file system

In figure five, *read* events will now be sent to DM app two, via session 69, not DM app one. *write* events will still be delivered to DM app one.

Rationale:

The burden is on the system administrator to ensure that two different DM applications do not attempt to control the same files on the same file system. In figure five, an alternative implementation of `dm_set_disp()` would be to return an error saying that an `<event, file system, session>` binding already exists. Another option would be to send a special event to DM app one, informing it that it no longer will be receiving *read* events. While these options could be implemented, it is believed that the level of complexity is not warranted for this version of the DMAPI.

The examples given above assume that the file system the DM application is monitoring is already mounted. However, it is quite possible that a DM application wants to set itself up to monitor a file system that is not yet mounted.

3.11.2 The *mount* Event

The restriction of only sending synchronous events to one session has special ramifications with regard to the *mount* event. It is not the intent of the DMAPI to force a model of one "super-daemon" that listens for *mount* events, and then forwards the event to the appropriate recipient. However,

there is a special bootstrap problem with regard to receiving the *mount* event before a file system handle is available. To receive *mount* events, a DM application must use the global handle in the dm_set_disp() call. The *mount* event will be sent serially to each session that has executed dm_set_disp(). The event is not broadcast to all sessions concurrently. The order in which the DMAPI implementation sends the event to the session is not defined.

The *mount* event will be sent for all file systems that support the DMAPI. Specifying the event in the dm_set_eventlist() call is not allowed, since the event is not persistent. When the *mount* event is received, the DM application can determine if it is interested in the file system that is specified in the event message. If a DM application is not interested in the file system, then it must respond to the event via dm_respond_event() with a code of DM_RESP_DONTCARE. The first DM application that responds to the event with DM_RESP_CONTINUE and an error code of zero will prevent the event from being sent to any of the remaining sessions. If any DM application returns an error (DM_RESP_ABORT code), then the *mount* event will not be sent to any other session.

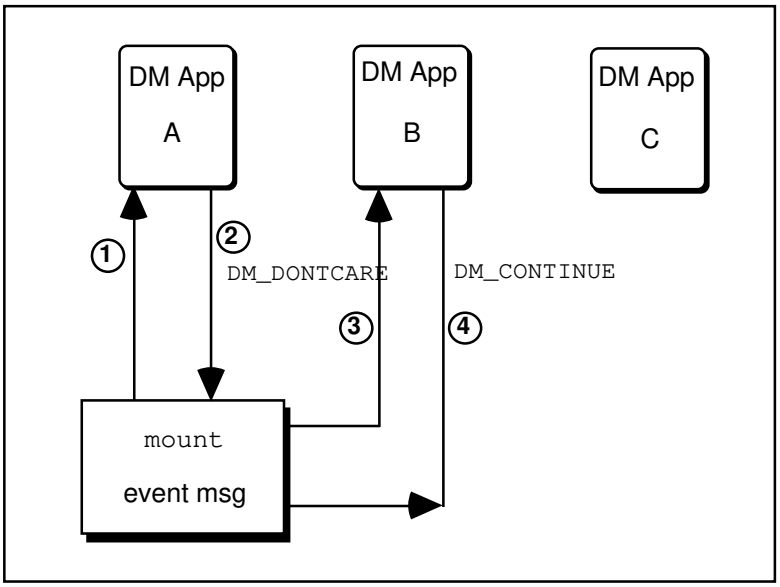

Figure 6: Mount event propagation

In figure six, three DM applications have specified via dm_set_disp() that they want to receive the mount event. The DMAPI implementation sends the mount event message to DM application A in step **1**, which is not interested in the event, so it responds to the event message with DM_RESP_DONTCARE in step **2**. The DMAPI implementation then sends the mount event message to DM app B in step **3** which determines that it wants to monitor the file system. It responds to the event message with a DM_RESP_CONTINUE in step **4**, so the mount event is not sent to the remaining DM app C.

For recovery processing, many DM applications will need the name of the file system device and the directory that it was mounted at. This information is made available via the *mount* event; during application restart, an application can get the same information via dm_get_mountinfo(). A DM app would determine all the file systems that were being monitored via dm_getall_disp(), and then use dm_get_mountinfo() to obtain more information about the file systems.

The following routines are provided for manipulating the disposition of a session's events for a file system:

```
int
dm_set_disp(
   dm_sessid_t     sid
   void           *hanp,
   size_t          hlen,
   dm_token_t      token
   dm_eventset_t  *eventsetp,
   u_int           maxevent)
```

Set the disposition of a session's events on a file system.

```
int
dm_getall_disp(
   dm_sessid_t     sid,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Get the disposition of events for all file systems for a session.

```
int
dm_get_mountinfo(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Get the information that was delivered on a mount event for the indicated file system.

3.11.3 Setting Event Notification

DM applications can specify that they need to receive certain events on an object. Events will only be generated for these objects, not all objects in the file system (except for the *debut* event, discussed later). To set event notification on a object, the DM application must specify an event list for the object. This object is specified via a handle; the handle can be either the file system handle when setting events on a per file system basis, or a handle to a specific file system object. Executing `dm_set_eventlist()` may or may not persistently store the eventlist with the object; it is dependent on the particular implementation of the DMAPI. The persistence characteristic can be determined via the `dm_get_config()` call.

The DM application must specify the entire list of events that is to be generated for the object. If an event list already exists for the object, it is replaced by the new one specified in the `dm_set_eventlist()` call. All events, with the exception of the managed region events, can be specified in the `dm_set_eventlist()` call. If the object has multiple managed regions, then `dm_get_eventlist()` will return the union of all managed region events, in addition to the other events.

When an event is generated by the file system, the DMAPI implementation uses the session to determine the recipient. Since DM applications must register with the DMAPI via the `dm_set_disp()` to specify the event list and the session, the DMAPI can easily determine the target session for any given event.

Some implementations of the DMAPI may not provide any persistent storage, even for event notification. For these "zero bit" implementations, the DMAPI provides a *debut* event before any access is granted to the object. This *debut* event should be specified in the event list when the DM application sets its event disposition. The *debut* event gives the DM application the ability to download information (such as event lists and managed region information) that may be needed by the DMAPI implementation. Most likely, when downloading a new event list for an object, the list will not include the *debut* event, but only include events that require some action to be performed by the DM application.

The *debut* event is the first indication given to a DM application that a primitive DMAPI implementation is going to perform an operation on a file. The DM application can take this opportunity to download all the necessary information for that particular file, or for other files as well. Alternately, some DM applications may want to intercept the *Mount* event to prime primitive DMAPI implementations, rather than having to receive many *debut* events.

The following routines for managing event lists on a file system objects are provided:

```
int
dm_set_eventlist(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_eventset_t  *eventsetp,
   u_int           maxevent)
```

Specify the events, with the exception of the managed region events, to be generated for an object.

```
int
dm_get_eventlist(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           nelem,
   dm_eventset_t  *eventsetp,
   u_int          *nelemp)
```

Get the list of events to be generated for an object.

3.11.4 Receiving and Responding to Events

Pending events can received one at a time or in bulk. For synchronous events, a response to each event message is required. For all events the only valid response is an indication of whether the operation should be continued or aborted. If the operation is to be aborted, an error can also be specified that will be returned to the user.

Event messages are variable length. This is because two of the primary fields of most event messages, file handles and pathnames, are variable length. DM applications should use `dm_get_config()` to determine the largest message size to size their buffers for calls to `dm_get_events()`. For more information on accessing and manipulating variable length message buffers, see the *Data Structures* section.

The process that generated the event is blocked until the response is received by the DMAPI implementation. The sleep may or may not be interruptible; the implementation of the DMAPI will need to define the behavior for each synchronous event.

When a synchronous event message is generated, a token is part of the message. The token identifies the event message, and may reference access rights that are conveyed as part of the event message. No tokens are passed in an asynchronous messages.

When a DM application responds to an event message, the token may reference access rights. If a DM application allows the operation to continue with the `DM_RESP_CONTINUE` return code, then special care must be taken by the implementation of the DMAPI to allow the operation that caused the event generation to continue without another DM application changing the state of the file.

Consider the following example:

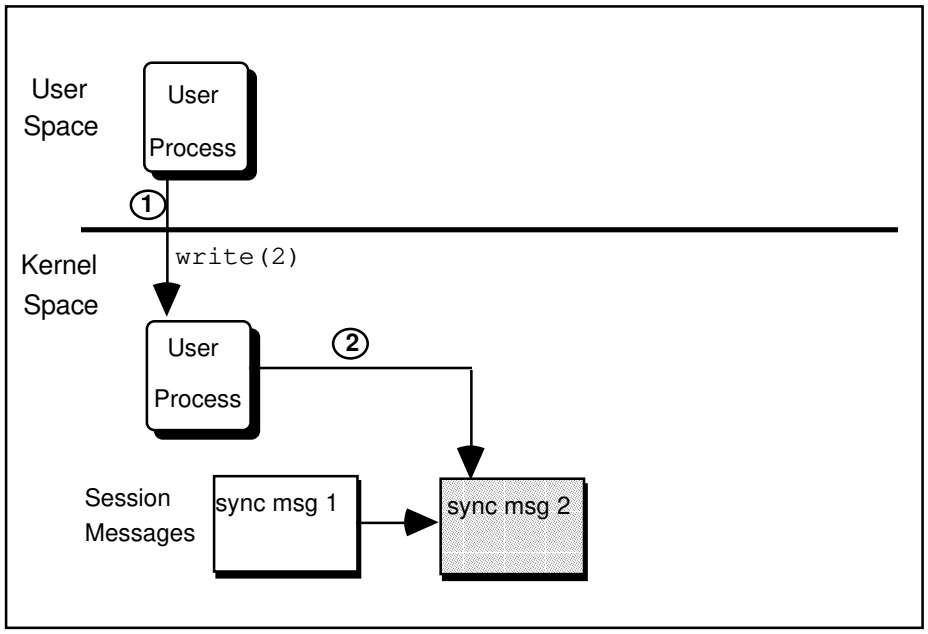

Figure 7: Event generation with no rights

In figure seven, the user process has initiated a `write(2)` operation in user space, shown as step **1**. When the application begins executing the operating system code that performs the operation in the kernel, it detects that it must generate a synchronous managed region write event. The event message is enqueued on the session in step **2**, and the user process is then pended.

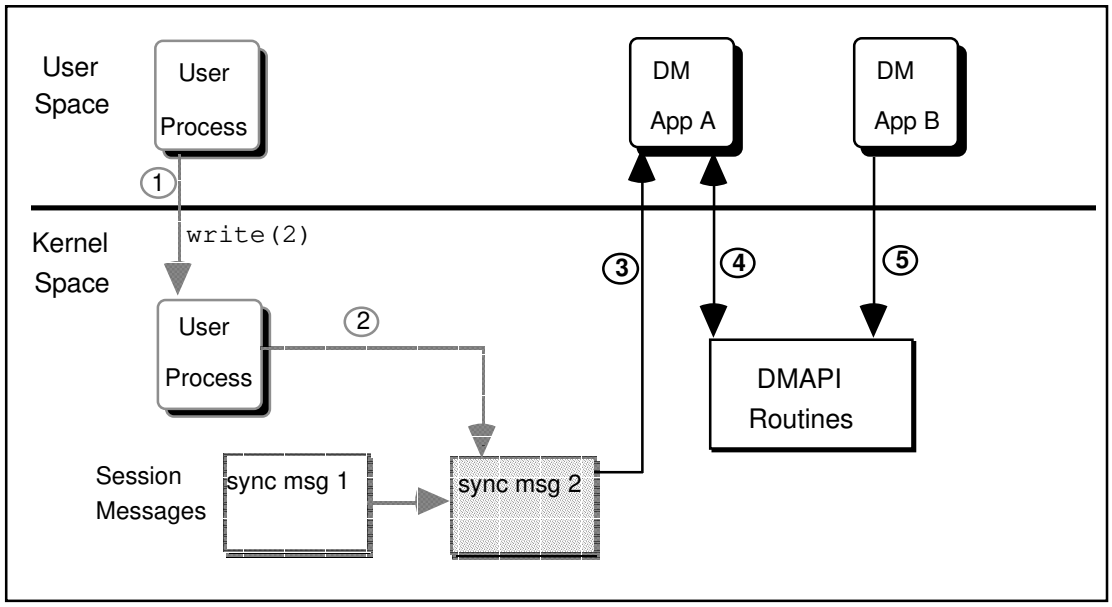

Figure 8: Requesting access rights after event generation

In figure eight, the event message has been enqueued on the session in step **2**, and is delivered to DM app A via `dm_get_events()` in step **3**. Since the event message conveys no rights, DM app A must obtain access rights to the object. In this example, it requires the `DM_RIGHT_EXCL` right, which it obtains in step **4**. At the same time, DM app B attempts to get exclusive access to the file in step **5**. Since the access right is not available, DM app B will pend.

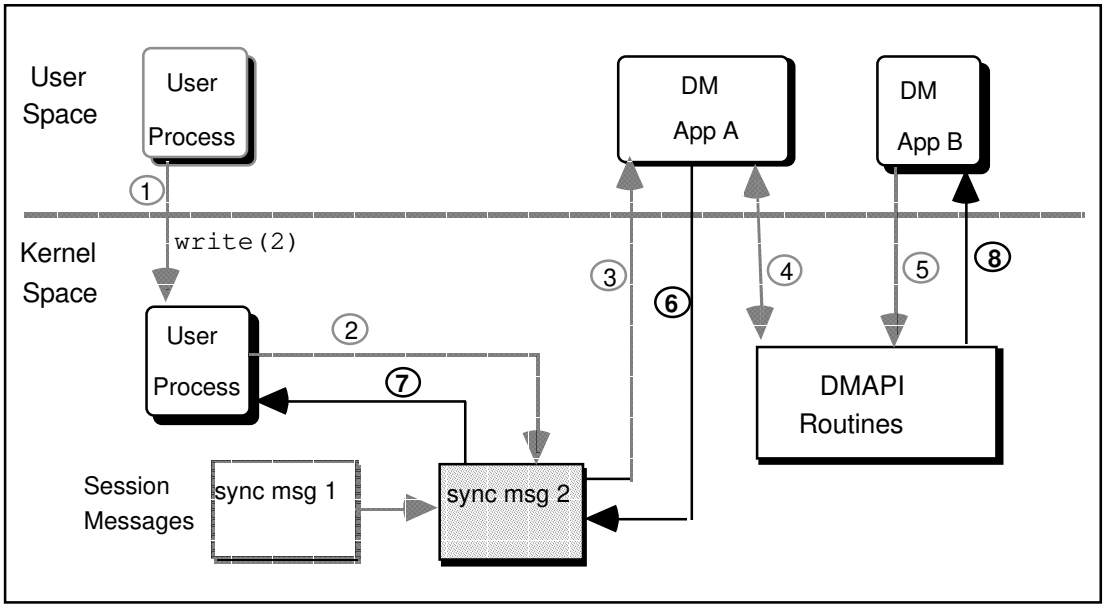

Figure 9: Continuing an event with access rights

In figure nine, DM app A has completed its processing in step **6** and continues the operation via a `dm_respond_event()` with the `DM_RESP_CONTINUE` response code. At the point when the call returns to the DM app (not explicitly shown, but it can be assumed to be a step **6a**), the token that referenced the access rights to the object is invalid. However, the DMAPI implementation can not immediately release the rights referenced by the token and grant them to someone else.

In step **7**, the user process that caused the event to be generated is resumed by the operating system, and continues operation at the point at which the event was generated. Once the DMAPI implementation has completed whatever event processing it deems necessary, and once it has acquired whatever locks it needs to complete the rest of the `write(2)` operation, the access rights can be released. At this point, DM app B can be allowed to obtain the `DM_RIGHT_EXCL` access right, in step **8**.

Rationale:

DM applications are logical extensions of the file system. When a DM application has completed the servicing of an event, it should appear as though the conditions that caused the event to be generated no longer exist. From the standpoint of the operating system, it is as though the event never occurred; whatever state that required the event to be generated has been taken care by the DM application.

In the example above, if DM app B were allowed to gain exclusive access to the file, it could possibly change the state of the file; all the recently-completed work of DM app A would be for naught. More importantly, the implementation of the DMAPI would have no way to tell what state the file is in, unless it monitored all the actions of DM app B. It is also important to prevent the user process from starvation. Therefore, the user process should be allowed to continue its processing after DM app A has completed the event servicing.

The following routines for receiving event messages and responding to synchronous messages are provided:

```
int
dm_get_events(
   dm_sessid_t     sid,
   u_int           maxmsgs,
   int             nowait,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Get the next available event or events.

```
int
dm_respond_event(
   dm_sessid_t     sid,
   dm_token_t      token,
   dm_response_t   response,
   int             errno,
   size_t          buflen,
   void           *respbufp)
```

Respond to a synchronous event.

Some DM applications may be multi-threaded (or made up of multiple processes). To facilitate the processing of events between related processes, the DMAPI provides a method to move an outstanding event message from one session to another. The event message remains in the *outstanding* state, even though it is now enqueued on a different session.

The following routine is provided:

```
int
dm_move_event(
    dm_sessid_t      srcsid,
    dm_token_t       token,
    dm_sessid_t      targetsid,
    dm_token_t      *rtokenp)
```

Move an event from one session to another.

3.11.5 Pseudo Events

Pseudo events do not correspond to an event generated as a result of an operation in the operating system, such as a `write(2)`. They are created by the DM application for purposes of generating a token or sending a message to a session. The actual message data is opaque to the DMAPI implementation. For the format of the pseudo-event, see the *Pseudo Events* section. There is currently only one type of pseudo event; the *`user`* event.

As described in the *Token Management* section, tokens are always associated with a synchronous event message. To gain access to an object, a DM application must first create a message that contains the context for a token. The required access right can then be obtained via `dm_request_right()`. `dm_create_userevent()` will create a synchronous event message of type *`user`* and enqueue it on the indicated session. The message and its corresponding token are *outstanding*. From the standpoint of the DMAPI, the message appears to have been delivered to a DM app via `dm_get_events()`, but has not yet been responded to via `dm_respond_event()`. The message will continue to exist until the DM app does a `dm_respond_event()` with the token.

For purposes of recovery processing, intelligent DM applications can use the user-generated event message mechanism to log their state during long and complicated operations. For example, if a DM application requires exclusive access to a file, it will first need to create a synchronous message. It will put together a user-level event message describing the operation, and then request that a token be generated and associated with this pseudo-event message. If the DM application aborts (via a bus error, kill signal, etc.) before responding to the event, when it restarts, it can obtain the message and any corresponding state. This can provide the application with valuable information about its state when it died.

User-created messages can also be used as a test mechanism, to ensure that communications between the DMAPI implementation and a DM application are working correctly. Applications can use `dm_send_msg()` to craft a synchronous or asynchronous message and have it enqueued on any specified session. The created message is also of type *`user`*, and contains the data specified by the user. For synchronous messages, the call will not return until the message has been responded to. Obviously, the process initiating the message via `dm_send_msg()` must not also be responsible for consuming the message via `dm_get_events()` or it will hang.

The following routines for creating a user level event message exists:

```
int
dm_create_userevent(
   dm_sessid_t     sid,
   size_t          msglen,
   void           *msgdatap,
   dm_token_t     *tokenp)
```

Generate a *user* pseudo-event message and return the token.

```
int
dm_send_msg(
   dm_sessid_t     targetsid,
   dm_msgtype_t    msgtype;
   size_t          buflen,
   void           *bufp)
```

Generate a *user* pseudo-event message and send it to the indicated session.

3.12 Configuration Information

In order for a DM application to determine information about the underlying implementation of the DMAPI, an interface exists to interrogate various implementation specific details. The function, `dm_get_config`, is called on a per file system basis. Based on selected options, the following information can be returned:

Maximum Attribute Size

The maximum size of a single persistent attribute per file is returned. Attributes are guaranteed a minimum amount of space, as described in the *Data Management Attributes* section.

Maximum Size of User Message Data

The number of bytes of data in the largest user created event message. DM applications can use this value to determine the largest buffer size allowed in calls to `dm_create_userevent()` and `dm_send_msg()`.

Number of Managed Regions Supported

The return value for this option is the maximum number of managed regions supported per file on the specified file system.

Operation Retry Supported

A boolean value indicating whether the DMAPI implementation will / will not retry the operation that caused a `nospace` event to be generated if a `DM_RESP_CONTINUE` response is returned.

Persistent Attribute Support

A boolean value indicating whether the DMAPI implementation supports / does not support persistent, opaque data management attributes. (see the *Data Management Attributes* section).

Persistent Event Masks Supported

A boolean value indicating whether the DMAPI implementation supports / does not support persistent event masks.

Persistent Managed Regions Supported

A boolean value indicating whether the DMAPI implementation supports / does not support persistent managed regions.

Punch Hole Supported

A boolean value indicating if the file system supports punching holes in files (see `dm_punch_hole()` and `dm_get_allocinfo()`).

Size of Persistent Attribute Space

The total size for persistent attributes per file system is returned. If persistent data management attributes are supported then there is a guaranteed minimum amount of space available which can be increased on a per file system basis.

The following routine exists for determining configuration information:

```
long
dm_get_config(
   void           *hanp,
   size_t          hlen,
   dm_config_t     flagname,
   dm_size_t      *retvalp)
```

Return configuration information for the specified file system.

4.0 Event Types

This section enumerates the types of events that can be generated, explains the data sent with the event message, and indicates what rights if any will be conveyed to the receiver of the event message. Each event message consists of a common portion and an event specific portion. The common portion consists of the event type, a token, and the length of the data.

The common portion will not be shown in the following descriptions of events. For specific information on the layout of event messages, see the section entitled *Event message types*.

There are five classes of events:

- file system administration.
- namespace.
- data.
- metadata.
- pseudo events.

Events can be either synchronous or asynchronous, as determined by the event type. Synchronous events allow a DM application to take action before an operation is continued by the operating system. For all events, it is not possible for a DM application to implement the operation in user space as though it were part of the operating system.

The DM application can respond to synchronous events in a number of ways. The operation can be continued, or it can be aborted with a specified error code returned back to the originating user process. For the `mount` event, the DM application can instruct the DMAPI implementation that it is not interested in the event, and the event should be sent to another session (see the *Setting Event Disposition* section for more information). Some DMAPI implementations may also support a *retry* response to a `nospace` event, in which case the operation will be retried by the operating system. Support of a retry response is indicated by the `DM_CONFIG_WILL_RETRY` boolean returned via `dm_get_config()`

Asynchronous messages inform the DM application of the success or failure of an operation in the operating system. They are delivered asynchronously with respect to the process performing the operation and offer no opportunity for a DM application to affect the requesting process; thus they are for notification purposes rather than for control.

Due to their asynchronous nature there is no inherent upper bound on the number of undelivered asynchronous messages that may be queued on a session. Therefore implementations must choose either a *reliable* or an *unreliable* model for asynchronous messages.

In the reliable model, no asynchronous messages are ever lost. However, this requires that once the number of undelivered asynchronous messages reaches an implementation-defined maximum that processes attempting to generate additional messages are blocked until previously queued messages are delivered.

In the unreliable model, the implementation may simply drop messages when the undelivered message count exceeds an implementation limit.

A conforming DMAPI implementation shall document which model it provides for asynchronous messages. It is recommended that implementations adopt a reliable model, or failing that, an unreliable model that provides reliable semantics in all but the most unusual cases.

For synchronous messages, tokens identify the message. Each synchronous event message **may** convey the access rights that are required by the DM application; however, it is not guaranteed. The same message may even convey different access rights, depending on the code path taken in the operating system. DM applications must use the `dm_query_right()` and `dm_request_right()` interfaces to obtain the proper access rights for their operations. For asynchronous messages, there is no token.

Implementation Responsibilities

The DMAPI implementation must specify for each event the behavior in the case that an event needs to be generated, but no session has been established to receive the event message. This is different from the case where a session exists, but there is no process to receive the event message (for example, the DM application has died). Some implementation alternatives are discussed in the *Undeliverable Events* section.

The implementation of the DMAPI will also need to define whether each individual synchronous event is interruptible. Since placement of events in the operating system is not defined by the DMAPI, interruptibility can not be mandated by this interface specification.

Interruptible Events

If an implementation provides interruptible events, it must guarantee that event messages that convey access rights can not vanish out from underneath a DM application.

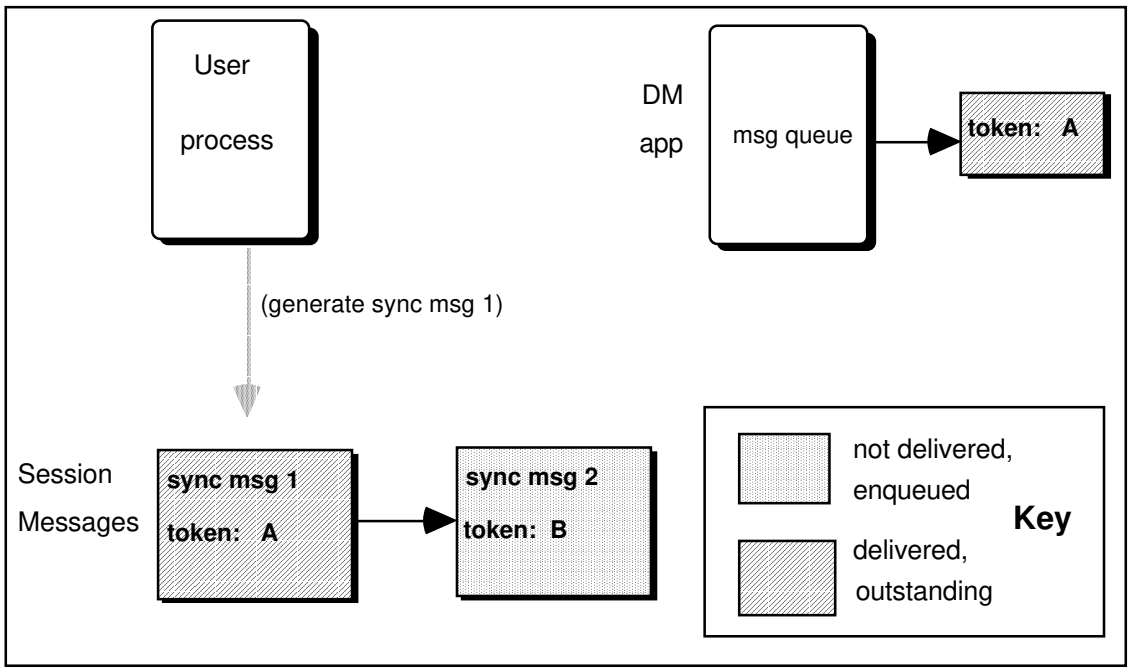

Figure 10: Interrupting a synchronous message

In figure ten, the user has performed an operation that caused a synchronous event message, sync msg 1, to be generated. The message has been received by the DM app, and is being processed. Assume the message conveyed rights to the DM app. If the implementation allows the user to interrupt the operation that caused the event message, the DMAPI implementation can not remove

the message from the queue because it conveyed rights to the recipient. If the message (and its corresponding token) were removed, the DM application would be suddenly be left holding an invalid token.

The following guidelines exist for handling interrupted messages:

- Event messages cannot be removed from a session if they convey any access rights to the recipient.
- Undelivered event messages can be removed from a session without difficulty.

It is important to note that a delivered event message containing a token cannot be removed from a session, even if the message conveys no rights. DM applications that are restartable depend on the existence of event messages to determine the state of the application before it died. If the message and its token are removed from the session, the restarted DM application has no way to pick up where it left off.

Asynchronous Namespace Event Delivery

Asynchronous namespace event messages are always generated for operations that succeed. An asynchronous event message will be generated for a failed event if the failure occurred at a point in the code sequence after which the corresponding synchronous event message would be generated.

4.1 File system Administration Events

4.1.1 mount

This event is generated at the point of a mount operation where the operation will succeed, but access to the file system has not yet been granted. This event allows DM applications to initialize, and provides them with the filesystem handle. This event is synchronous.

The *mount* event is unlike other events in that it may be sent to multiple sessions; see the section entitled *mount event* for more information.

The event message contains the following specific information:

fs handle	filesystem handle.
mountpoint handle	handle for the directory to be mounted over.
mountpoint path	pathname of the directory to be mounted over.
media designator	file system media designator.

4.1.2 preunmount

This event is generated at the point of an unmount operation where the operation is about to check for open files. Events for this file system may still be generated after the *preunmount* event. This allows the DM application to close any active accesses and "cd" out of the file system so that a DM application itself does not cause an unmount to fail. This event is synchronous.

The event message contains the following specific information:

fs handle	filesystem handle.
rootdir handle	handle to the root directory of the file system.
unmount mode	0 or DM_UNMOUNT_FORCE.

4.1.3 unmount

This event is generated after the operating system has attempted to unmount a file system. A return code of zero indicates the unmount was successful, and the file system is no longer mounted. Once unmounted, the DMAPI will not generate any other events for the file system until it is remounted. A non-zero return code indicates the ummount attempt failed. The reason for failure is given by the value of the return code. The DMAPI will continue to generate events for the file system after a failed unmount. The DMAPI implementation will always continue the operation, ignoring the response code from the DM application. This event is synchronous.

The event message contains the following specific information:

fs handle	filesystem handle.
unmount mode	0 or DM_UNMOUNT_FORCE.
retcode	return code of the operation (0 or errno).

4.1.4 nospace

This event is generated when an operation fails with ENOSPC. This event is synchronous. If the DMAPI implementation supports operation retry (DM_CONFIG_WILL_RETRY in the dm_get_config() call), a DM_RESP_CONTINUE response will cause the operation that produced the ENOSPC error to be generated to be retried.

The event message contains the following specific information:

fs handle filesystem handle for file system where the `ENOSPC` error occurred.

4.1.5 debut

If the debut event is specified when a DM application registers via `dm_set_disp()`, the DMAPI implementation will generate this event *at least once* before any of the set of other possible events are generated for the object. If the DMAPI implementation supports persistent managed regions and persistent event lists (`DM_CONFIG_PERS_MANAGED_REGIONS` and `DM_CONFIG_PERS_EVENTS` in the `dm_get_config()` call), this event may not be generated. For DMAPI implementations that do not provide persistent storage of managed regions or event lists, the DM application should download the managed region list and a specific event list for the object. Most likely, the event list will not include the debut event, but instead will only include events that require some action to be performed by the DM application. This event is synchronous.

The event message contains the following specific information:

object handle object that is being accessed.

4.2 Namespace Events

Namespace events allow a DM application to permit or deny an operation from continuing in the operating system. The synchronous events are generated before the operation has completed; there is no guarantee that the operation will be successful, however. To determine if the operation succeeded or failed, the DM application must also intercept the asynchronous *after* event. Asynchronous events are **always** sent if the operation succeeds and undefined whether or not they are sent if the operation fails.

For example, if a DM application is monitoring the *create* event, it should also monitor the *postcreate* event to determine if the operation was successful. The operating system will first send the *create* event followed by the *postcreate* event to the DM application.

The DMAPI guarantees the ordering of paired namespace events. Asynchronous *post* event messages will always be generated after synchronous *before* event messages. If there are multiple synchronous events on a directory entry, it is not guaranteed that the associated asynchronous events are properly interleaved.

Namespace event messages are generated based on the directory entry that is being affected. If a DM application wants to receive namespace events for every file in the file system, it should do a `dm_set_eventlist()` using the filesystem handle. Setting a namespace event on the filesystem handle is not persistent.

4.2.1 create

This event is generated before the operating system adds a new entry to a directory. This event is synchronous. There is no guarantee that the operation will be successful.

The event message contains the following specific information:

parent handle	parent directory where *entry name* will be added.
entry name	name of the new entry.
mode	type of the object to be created and is the same value as `st_mode` in `struct stat` and `dt_mode` in `struct dm_stat`.

The DM application can either allow the operation to continue (`DM_RESP_CONTINUE`) or return an error code.

4.2.2 postcreate

This event is generated after the operating system has attempted to add an entry to a directory. This event is asynchronous.

The event message contains the following specific information:

parent handle	parent directory where *entry name* was added.
entry handle	handle for the new entry.
entry name	entry name that was added to the directory.
mode	type of the object created and is the same value as `st_mode` in `struct stat` and `dt_mode` in `struct dm_stat`.
retcode	return code of the operation (0 or `errno`).

4.2.3 remove

This event is generated before the operating system removes an entry from a directory. This event is synchronous. There is no guarantee that the operation will be successful.

The event message contains the following specific information:

parent handle	parent directory where *entry name* will be removed.
entry name	entry name to be removed from the directory.
mode	type of the object to be removed and is the same value as `st_mode` in `struct stat` and `dt_mode` in `struct dm_stat`.

The DM application can either allow the operation to continue (`DM_RESP_CONTINUE`) or return an error code.

4.2.4 postremove

This event is generated after the operating system has attempted to remove an entry from a directory. This event is asynchronous.

The event message contains the following specific information:

parent handle	parent directory where *entry name* was removed.
entry name	entry name removed from the directory.
mode	type of the object removed and is the same value as `st_mode` in `struct stat` and `dt_mode` in `struct dm_stat`.
retcode	return code of the operation (0 or `errno`).

4.2.5 rename

This event is generated before the operating system renames a directory entry in a file system. This event is synchronous. There is no guarantee that the operation will be successful.

The event message contains the following specific information:

old parent handle	original parent directory where *old entry name* resides.
new parent handle	new parent directory where *new entry name* is to be added.
old entry name	directory entry to be renamed.
new entry name	new name of directory entry.

The DM application can either allow the operation to continue (`DM_RESP_CONTINUE`) or return an error code.

4.2.6 postrename

This event is generated after the operating system has attempted to rename an entry in a file system. This event is asynchronous.

The event message contains the following specific information:

old parent handle	original parent directory where *old entry name* resided.
new parent handle	new parent directory where *new entry name* was added.
old entry name	original directory entry that was renamed.
new entry name	new name of directory entry.
retcode	return code of the operation (0 or `errno`).

4.2.7 symlink

This event is generated before the operating system adds a new symbolic link entry in a directory. This event is synchronous. There is no guarantee that the operation will be successful.

The event message contains the following specific information:

parent handle	parent directory where *symlink entry name* will be added.
symlink entry name	name of symbolic link entry.
symlink contents	contents of symbolic link.

The DM application can either allow the operation to continue (`DM_RESP_CONTINUE`) or return an error code.

4.2.8 postsymlink

This event is generated after the operating system has attempted to add a symbolic link entry in a directory. This event is asynchronous.

The event message contains the following specific information:

parent handle	parent directory where *symlink entry name* was added.
entry handle	handle for the new symlink entry.
symlink entry name	entry name of newly added symbolic link.
symlink contents	contents of symbolic link.
retcode	return code of the operation (0 or `errno`).

4.2.9 link

This event is generated before the operating system adds a new hard link entry in a directory. This event is synchronous. There is no guarantee that the operation will be successful.

The event message contains the following specific information:

parent handle	parent directory where *target entry name* will be added.
source link handle	handle of entry to be linked to.
target entry name	new link name to be added to the directory.

The DM application can either allow the operation to continue (`DM_RESP_CONTINUE`) or return an error code.

4.2.10 postlink

This event is generated after the operating system has attempted to add a new hard link entry in a directory. This event is asynchronous.

The event message contains the following specific information:

parent handle	parent directory where *target entry name* was added.
source link handle	handle of entry that was linked to.
target entry name	new link name added to the directory.
retcode	return code of the operation (0 or `errno`).

4.3 Data Events

Data events are triggered by interactions with managed regions. They are synchronous, and allow the DM application to take some action before the operation is continued in the operating system. Data events only occur on regular files.

4.3.1 read

This event is generated when a read of a file overlaps one or more managed regions that have the event set in the managed region flags field. This event is synchronous.

The event message contains the following specific information:

file handle	handle for file being read.
offset	starting offset of read operation.
length	length in bytes of read request.

4.3.2 write

This event is generated when a write to a file overlaps one or more managed regions that have the event set in the managed region flags field. This event is synchronous.

The event message contains the following specific information:

file handle	handle for file being written.
offset	starting offset for write operation.
length	length in bytes of write request.

4.3.3 truncate

This event is generated when the operating system attempts a truncation of a file that overlaps one or more managed regions that has the event set in the managed region flags field. This event can be generated directly in response to a truncate request by the user, as from `truncate(2)`, or indirectly as when an existing file is truncated via `open(2)` with the appropriate flags. This event is synchronous.

The event message contains the following specific information:

file handle	handle for file being truncated.
offset	starting offset for truncation operation.

4.4 Metadata Events

4.4.1 attribute

This event is generated when an object is changed in a way that affects the change time (`dt_ctime`) value returned by `dm_get_bulkattr()`, `dm_get_dirattrs()` or `dm_get_fileattr()`. This event is asynchronous.

The event message contains the following specific information:

object handle object that was affected.

4.4.2 destroy

This event is generated when the operating system has destroyed an object. This event is asynchronous.

The event message contains the following specific information:

object handle object that was destroyed.

4.5 Pseudo Events

Pseudo events are events that do not correspond to any operation in the operating system. Currently, the only defined pseudo event is the *user* event that is generated by both `dm_create_userevent()` and `dm_send_msg()`.

4.5.1 user event format

This event structure is used in the `dm_create_userevent()` and `dm_send_msg()` calls for both synchronous and asynchronous events. All `dm_create_userevent()` calls are synchronous. `dm_send_msg()` calls can be either synchronous or asynchronous. Only synchronous events will associate a token with the event message. Asynchronous events can be indicated by a value of `DM_INVALID_TOKEN` in the token field. DM applications can use this event to store information that is associated with the message that is to be created. The message data is not interpreted by the DMAPI implementation.

For more information see the *Token Management* and *Events* sections.

Since this is not a generated event, the entire event message will be shown, and is as follows:

event type	set to `DM_EVENT_USER`.
token	token that will be returned or `DM_INVALID_TOKEN`.
message length	length of the data to follow.
data	private data that the DM application wants to have associated with a message.

4.6 Event Summary

A summary of events, the type of objects they are delivered on, and how they are enabled is given in the following table:

Event	Sync/Async	Enabled By	Handle Applies To
mount	sync	always enabled	global
preunmount	sync	dm_set_eventlist	fs
unmount	sync	dm_set_eventlist	fs
nospace	sync	dm_set_eventlist	fs
debut	sync	always enabled[1]	file, dir or fs
create	sync	dm_set_eventlist	dir or fs
postcreate	async	dm_set_eventlist	dir or fs
remove	sync	dm_set_eventlist	dir or fs
postremove	async	dm_set_eventlist	dir or fs
rename	sync	dm_set_eventlist	dir or fs
postrename	async	dm_set_eventlist	dir or fs
symlink	sync	dm_set_eventlist	dir or fs
postsymlink	async	dm_set_eventlist	dir or fs
link	sync	dm_set_eventlist	dir or fs
postlink	async	dm_set_eventlist	dir or fs
read	sync	dm_set_region	managed region
write	sync	dm_set_region	managed region
truncate	sync	dm_set_region	managed region
attribute	async	dm_set_eventlist	file, dir or fs
destroy	async	dm_set_eventlist	file, dir or fs
user	either	created by dm_create_userevent and dm_send_msg	not applicable

Table 2: Event summary

[1] Applies only for zero bit DMAPI implementations. See the *Setting Event Notification* section for more information.

The following table lists the objects that the DMAPI implementation will consult to determine if an event should be generated. For example, the *rename* event will be generated if a `rename()` occurs where one of the following objects has that event enabled:

- old parent directory.
- new parent directory.
- filesystem handle for the file system containing the object to be renamed.

Event	Controlling Object Handle
mount	global
preunmount	fs
unmount	fs
nospace	fs
debut	file, dir or fs
create	parent dir or fs
postcreate	parent dir or fs
remove	parent dir or fs
postremove	parent dir or fs
rename	old parent, new parent, or fs
postrename	old parent, new parent, or fs
symlink	parent dir or fs
postsymlink	parent dir or fs
link	parent dir or fs
postlink	parent dir or fs
read	file
write	file
truncate	file
attribute	file, dir or fs
destroy	file, dir or fs
user	n/a

Table 3: Event generation objects

Events are defined via a number of different data structures. The following table summarizes the event structures that are used to deliver each event.

Event	Data structure
mount	dm_namesp_event
preunmount	dm_namesp_event
unmount	dm_namesp_event
nospace	dm_namesp_event
debut	dm_namesp_event
create	dm_namesp_event
postcreate	dm_namesp_event
remove	dm_namesp_event
postremove	dm_namesp_event
rename	dm_namesp_event
postrename	dm_namesp_event
symlink	dm_namesp_event
postsymlink	dm_namesp_event
link	dm_namesp_event
postlink	dm_namesp_event
read	dm_data_event
write	dm_data_event
truncate	dm_data_event
attribute	dm_namesp_event
destroy	dm_namesp_event
user	opaque to DMAPI implementation

Table 4: Event data structures

A number of different events use the `dm_namesp_event` structure. The following table lists which fields are valid, and their contents, for each event message that uses this structures.

- == not applicable
r/c == return code

Event	handle1	handle2	name1	name2	mode	retcode
mount	fs	mountpoint	mountpoint	media designator	-	-
preunmount	fs	rootdir	-	-	unmount mode	-
unmount	fs	-	-	-	unmount mode	r/c
nospace	fs	-	-	-	-	-
debut	object	-	-	-	-	-
create	parent	-	name	-	st_mode	-
postcreate	parent	new	name	-	st_mode	r/c
remove	parent	-	name	-	st_mode	-
postremove	parent	-	name	-	st_mode	r/c
rename	old parent	new parent	old name	new name	-	-
postrename	old parent	new parent	old name	new name	-	r/c
symlink	parent	-	name	contents	-	-
postsymlink	parent	new	name	contents	-	r/c
link	parent	source	name	-	-	-
postlink	parent	source	name	-	-	r/c
attribute	object	-	-	-	-	-
destroy	object	-	-	-	-	-

Table 5: Field use in the dm_namesp_event structure

5.0 Data Structures

The DMAPI references a number of data structures that are presented here in alphabetical order. Many of the interfaces deal with variable length data, such as handles and pathnames. This data is encapsulated in the `dm_vardata_t` structure. To access the data, the following routines are provided:

```
#define DM_GET_VALUE(p, field, type)  \
         ((type)((char *)(p) + (p)->field.vd_offset))

#define DM_GET_LEN(p, field)  ((p)->field.vd_length)
```

Rationale:

The philosophy behind all the interfaces that deal with variable length data is to allow for such data to be embedded in the same buffer as the structure that contains a reference to it. The references are all relative to the start of the structure containing them, so that buffers containing such structures can be copied without destroying the embedded references. The expectation is that the variable length data will most commonly be appended directly after the structure that references it, but the `dm_vardata_t` interface doesn't demand this spatial relationship. However, the interfaces that return variable length data are designed (and implicitly specified) so that this spatial relationship **does** hold for the information they return.

To access fields within a variable length structure, the DM application must use the macros defined above. For example, to get at the handle in an event message, the following code could be used:

```
dm_eventmsg_t     *msg = malloc(bufsize);
dm_data_event_t   *event;

dm_get_events(sid, ....msg,...);
event = DM_GET_VALUE(msg, data, dm_data_event_t *);
hanp  = DM_GET_VALUE(event, handle, void *);
hlen  = DM_GET_LEN(event, handle, len *);
```

A number of DMAPI routines return lists of variable length structures. To move from one structure to the next in a list, the following macro should be used:

```
#define DM_STEP_TO_NEXT(struct_ptr, struct_type)
```

A suggested implementation of the `DM_STEP_TO_NEXT()` macro is given under *Structure Lists* in the *Implementation Notes* section.

The following table shows the affected structures:

Structure Name
`dm_attrlist_t`
`dm_dispinfo_t`
`dm_eventmsg_t`
`dm_stat_t`

Table 6: Structures used in lists

The `DM_STEP_TO_NEXT()` macro can be used as shown in the following example:

```
dm_stat_t   *statbuf = malloc(bufsize);
dm_stat_t   *sbufp;

dm_init_attrloc(...)
dm_get_bulkattr(sid, ...statbuf);
```

```
sbufp = statbuf;
while (sbufp != NULL) {
     ......
     sbufp = DM_STEP_TO_NEXT(sbufp, dm_stat_t *);
}
```

5.1 dm_attrname_t

```
struct dm_attrname {
   u_char an_chars[8];
};
typedef struct dm_attrname    dm_attrname_t;
```

Description

This is used by the data management attributes routines to identify an attribute. The name is treated as an eight character opaque identifier; it is not a NULL-terminated string.

It is recommended that applications use printable eight character names (or pad smaller names with the NULL character) to simplify display of attributes names. However, applications are free to use any name encoding they wish. An attribute name constructed as:

```
dm_attrname_t x;

x.an_chars[0] = 0xff;
x.an_chars[1] = 0xff;
     ....
x.an_chars[7] = 0xff;
```

is a legal attribute name.

5.2 dm_attrlist_t

```
struct dm_attrlist {
   dm_attrname_t  al_name;
   dm_vardata_t   al_data;
};
typedef struct dm_attrlist    dm_attrlist_t;
```

Description

A `dm_attrlist_t` pointer is passed to `dm_getall_dmattr()` to retrieve all persistent data management attributes associated with the specified file. The data structure includes, but is not limited to, the fields described above.

Since attributes may have different size data fields, the `al_datalen` field indicates the length the data. All attribute names are 64 bit opaque (to the DMAPI) values while the length of data is variable. DM applications should use the `DM_GET_VALUE()` macro to obtain the data.

Attributes are guaranteed a minimum of 32 bytes for storage; however, the maximum space available for storage of a single attribute's data for an individual file is limited to `DM_CONFIG_MAX_ATTRIBUTE_SIZE`, as returned by `dm_get_config()`. The DMAPI specifies a lower bound of 256 bytes for `DM_CONFIG_MAX_ATTRIBUTE_SIZE`. The total amount of space available for storage of **all** persistent attributes on a file system is bounded by `DM_CONFIG_TOTAL_ATTRIBUTE_SPACE`.

5.3 dm_attrloc_t

Description

An opaque scalar that is used in the `dm_get_bulkattr()` and `dm_init_attrloc()` calls.

5.4 dm_config_t

```
typedef enum {
   DM_CONFIG_INVALID,
   DM_CONFIG_MAX_ATTRIBUTE_SIZE,
   DM_CONFIG_MAX_MANAGED_REGIONS,
   DM_CONFIG_MAX_MESSAGE_DATA,
   DM_CONFIG_PERS_ATTRIBUTES,
   DM_CONFIG_PERS_EVENTS,
   DM_CONFIG_PERS_MANAGED_REGIONS,
   DM_CONFIG_PUNCH_HOLE,
   DM_CONFIG_TOTAL_ATTRIBUTE_SPACE,
   DM_CONFIG_WILL_RETRY
} dm_config_t;
```

Description

Enums from `dm_config_t` are used on `dm_get_config()` call to query various options pertaining to the implementation of the DMAPI.

5.5 dm_dispinfo_t

```
struct dm_dispinfo {
   dm_vardata_t   di_fshandle;
   dm_eventset_t  di_eventset;
};
typedef struct dm_dispinfo    dm_dispinfo_t;
```

Description

The `dm_dispinfo_t` structure is used on calls to `dm_getall_disp()` to retrieve the list of all event dispositions for all file systems for a given session. DM applications should use the `DM_GET_VALUE()` macro to retrieve the value of the filesystem handle and `DM_STEP_TO_NEXT()` to move between successive structures.

5.6 Event message types

5.6.1 dm_eventmsg_t

```
struct dm_eventmsg {
   dm_eventtype_t ev_type;
   dm_token_t     ev_token;
   dm_vardata_t   ev_data;
};
typedef struct dm_eventmsg    dm_eventmsg_t;
```

Description

Every event message is of type `dm_eventmsg_t`. Since multiple messages can be returned in a single call, DM applications must use the `DM_STEP_TO_NEXT()` macro to move from one message to the next.

All event messages contain an event type. Synchronous messages also contain a token. The `dm_eventmsg_t` structure represents this common data via the `ev_type` and `ev_token` fields. Asynchronous messages never have a token associated with them; therefore, `ev_token` will always be `DM_INVALID_TOKEN` for asynchronous messages.

The rest of the contents of the message for a given event is distinct for an event's type, and will be contained in a `dm_`***`xxx`***`_event` buffer, where ***`xxx`*** depends on the event type. As with other data whose size or type isn't known in advance, it is referred to through the `DM_GET_VALUE()` macro applied to a descriptor field that lives in the common prelude part of the event message.

`ev_type`	The event type that occurred.
`ev_token`	The token may reference and identify a specific access right for a handle in the event-specific portion of the message. When responding to the message, the token is used to identify the message and is no longer valid when the response call (`dm_respond_event()`) has completed. The token field is valid for synchronous messages only; the field will be `DM_INVALID_TOKEN` for asynchronous event messages.
`ev_data`	The event-specific data.

5.6.2 dm_data_event_t

```
struct dm_data_event {
   dm_vardata_t   de_handle;
   dm_off_t       de_offset;
   dm_size_t      de_length;
};
typedef struct dm_data_event  dm_data_event_t;
```

Description

The `dm_data_event` structure is used for all data events, both synchronous and asynchronous.

5.6.3 dm_namesp_event_t

```
struct dm_namesp_event {
   mode_t               ne_mode;
   dm_vardata_t         ne_handle1;
   dm_vardata_t         ne_handle2;
   dm_vardata_t         ne_name1;
   dm_vardata_t         ne_name2;
   int                  ne_retcode;
};
typedef struct dm_namesp_event      dm_namesp_event_t;
```

Description

The `dm_namesp_event` structure is used for all namespace events.

5.7 dm_eventset_t

Description

To specify events, the `dm_eventset_t` type must be used. This is an opaque type (to the DM application) that is manipulated via the `DMEV_ZERO()`, `DMEV_SET()` and `DMEV_CLR()` macros in a manner analogous to the `FD_SET()` type that is used in calls to `select(2)`.

5.8 dm_eventtype_t

```
typedef enum {
   DM_EVENT_INVALID,
   DM_EVENT_MOUNT,
   DM_EVENT_PREUNMOUNT,   DM_EVENT_UNMOUNT,
   DM_EVENT_NOSPACE,
   DM_EVENT_DEBUT,
   DM_EVENT_CREATE,       DM_EVENT_POSTCREATE,
   DM_EVENT_REMOVE,       DM_EVENT_POSTREMOVE,
   DM_EVENT_RENAME,       DM_EVENT_POSTRENAME,
   DM_EVENT_SYMLINK,      DM_EVENT_POSTSYMLINK,
   DM_EVENT_LINK,         DM_EVENT_POSTLINK,
   DM_EVENT_READ,
   DM_EVENT_WRITE,
   DM_EVENT_TRUNCATE,
   DM_EVENT_ATTRIBUTE,
   DM_EVENT_DESTROY,
   DM_EVENT_USER,
   DM_EVENT_MAX
} dm_eventtype_t;
```

Description

Event types are encoded in message structures and `dm_eventset_t` types as `dm_eventtype_t` enums. `DM_EVENT_MAX` is guaranteed to be the number of the largest event type that can be represented in the `dm_eventset_t` type.

5.9 dm_extent_t

```
struct dm_extent {
   dm_extenttype_t      ex_type;
   dm_off_t             ex_offset;
   dm_size_t            ex_length;
};
typedef struct dm_extent      dm_extent_t;
```

Description

The extent structures returned by the call to `dm_get_allocinfo()` contain the fields `ex_type`, whose value is one of two indicators for the type of extent. `DM_EXTENT_RES` indicates that disk blocks are allocated for the portion of the file defined by this extent. `DM_EXTENT_HOLE` indicate that no disk blocks are allocated for the extent. `ex_offset` is the byte offset into the file where the extent begins, and `ex_length` is the byte count of the extent.

5.10 dm_extenttype_t

```
typedef enum {
   DM_EXTENT_INVALID,
   DM_EXTENT_RES,
   DM_EXTENT_HOLE
} dm_extenttype_t;
```

Description

The dm_extenttype_t enum is used to specify the type of extent in the dm_extent_t structure.

5.11 dm_fileattr_t

```
struct dm_fileattr {
   mode_t    fa_mode;
   uid_t     fa_uid;
   gid_t     fa_gid;
   time_t    fa_atime;
   time_t    fa_mtime;
   time_t    fa_ctime;
   time_t    fa_dtime;
} ;
typedef struct dm_fileattr dm_fileattr_t;
```

Description

These are the fields that can be modified by dm_set_fileattr().

5.12 DM Handles

```
void        *hanp;
size_t       hlen;
```

Description

Handles are opaque to DM applications. The length of a handle is implementation defined. When a handle must be used in a function, it is specified using two parameters. The hlen parameter specifies the length of the opaque data that is pointed to by hanp.

When a handle is embedded in a structure, the dm_vardata_t structure is used. The DM application should use the DM_GET_VALUE() macro to access the handle data and DM_GET_LEN() to determine the length of the opaque handle data.

5.13 dm_inherit_t

```
struct dm_inherit {
   dm_attrname_t  ih_name ;
   mode_t         ih_filetype;
};
typedef struct dm_inherit     dm_inherit_t;
```

Description

A dm_inherit_t pointer is passed to dm_getall_inherit() to retrieve all data management attributes which have been made inheritable through the dm_set_inherit() call. The call is made on a per file system basis. The ih_filetype field is the type of the file as defined in Spec 1170.

5.14 dm_msgtype_t

```
typedef enum {
   DM_MSGTYPE_INVALID,
   DM_MSGTYPE_SYNC,
   DM_MSGTYPE_ASYNC
} dm_msgtype_t;
```

Description

This is used when creating a user event via `dm_send_msg()` to specify the type of event message that should be created, synchronous or asynchronous.

5.15 dm_off_t

Description

This is a signed scalar used to represent file offsets.

Rationale:

Vendors have taken different approaches for extending file addressibility beyond the `off_t` type, which is usually 32 bits. For example, some vendors have an `offset_t` type or an `off64_t` type for working with files larger than 2GB.

The intent is that `dm_off_t` be a suitable type for addressing all files supported by the system.

5.16 dm_region_t

```
struct dm_region {
   dm_off_t        rg_offset;
   dm_size_t       rg_size;
   u_int           rg_flags;
};
typedef struct dm_region      dm_region_t;
```

Description

The `dm_region` structure defines the range of bytes that are managed in the file. The `rg_flags` field can be set to generate synchronous read, write, and truncate events whenever the associated operation is performed within the managed region. A region may extend outside the current valid portions of a file. A region may extend to the end of a file and allow control on files that are appended to by setting `rg_size` to zero. This means that the `rg_flags` field must always be set to a valid state. Regions may also be allocated that have no flags set. This may be used to allocate space for future control of a region of the file in DMAPI implementations that provide persistent managed regions.

Valid values for the `rg_flags` field are as follows:

`DM_REGION_READ`	Generate synchronous event for read operations that overlap this managed region
`DM_REGION_WRITE`	Generate synchronous event for write operations that overlap this managed region
`DM_REGION_TRUNCATE`	Generate synchronous event for truncate operations that overlap this managed region
`DM_REGION_NOEVENT`	Do not generate any events for this managed region.

5.17 Region Flags

```
#define DM_REGION_NOEVENT        0x0
#define DM_REGION_READ           0x1
#define DM_REGION_WRITE          0x2
#define DM_REGION_TRUNCATE       0x4
```

Description

The defines above are used to specify the generation of events in the `rg_flags` field of the managed region structure. `DM_REGION_READ`, `DM_REGION_WRITE` and `DM_REGION_TRUNCATE` may be OR'ed together.

5.18 dm_response_t

```
typedef enum {
   DM_RESP_INVALID,
   DM_RESP_CONTINUE,
   DM_RESP_ABORT,
   DM_RESP_DONTCARE
} dm_response_t;
```

Description

When a DM application responds to an event via `dm_respond_event()`, the response codes must be an enum taken from `dm_response_t`.

5.19 dm_right_t

```
typedef enum {
   DM_RIGHT_NULL,
   DM_RIGHT_SHARED,
   DM_RIGHT_EXCL
} dm_right_t;
```

Description

The `dm_right_t` enum specifies the access rights for an object.

5.20 dm_size_t

Description

This is an unsigned scalar used to represent sizes of extents within files (among other things).

5.21 dm_ssize_t

Description

This is a signed scalar used for function calls that return the size of the file or -1 to indicate error.

5.22 dm_sessid_t

Description:

The session ID is an opaque scalar. Implementations of the DMAPI may differ in the information they require to be stored with the session ID. A special value, `DM_NO_SESSION`, is available for use as an invalid or non-existent session.

5.23 dm_stat_t

```
struct dm_stat {
   dm_vardata_t   dt_handle;
   dm_vardata_t   dt_compname;
   dm_eventset_t  dt_emask;
   int            dt_nevents;
   int            dt_pers;
   int            dt_pmanreg;
   time_t         dt_dtime;
   u_int          dt_change;

   dev_t          dt_dev;
   ino_t          dt_ino;
   mode_t         dt_mode;
   nlink_t        dt_nlink;
   uid_t          dt_uid;
   gid_t          dt_gid;
   dev_t          dt_rdev;
   dm_off_t       dt_size;
   time_t         dt_atime;
   time_t         dt_mtime;
   time_t         dt_ctime;
   u_int          dt_blksize;
   dm_size_t      dt_blocks;
};
typedef struct dm_stat  dm_stat_t;
```

Description

The `dm_stat` structure is used for retrieving per file attributes via the `dm_get_fileattr()`, `dm_get_dirattrs()`, and `dm_get_bulkattr()` functions. The structure is composed of at least fields found in the Spec 1170 `struct stat` as well as additional DMAPI specific fields.

Since handles are variable length, the structure is somewhat problematic as `dm_get_bulkattr()` can return a variable number of variable length structures. To access the variable length data in an individual structure, the `DM_GET_VALUE()` macro should be used. To move between structures, the `DM_STEP_TO_NEXT()` macro is used.

The `dm_stat` structure contains at least the following fields:

`dt_handle`	A `dm_vardata_t` structure that contains the offset from the beginning of the structure where the data for the handle begins. The `DM_GET_VALUE()` macro should be used to access the handle.
`dt_compname`	A `dm_vardata_t` structure that contains the offset from the beginning of the structure where the name of the file begins. This field is valid only for `dm_get_dirattrs()`. The `DM_GET_VALUE()` macro should be used to access the file name.
`dt_emask`	The event bindings that are stored persistently with the file. The event types in this set can be manipulated by the `dm_set_eventlist()` routines.

	Not all DMAPI implementations support persistent event bindings. Support can be determined through use of the `dm_get_config()` function.
`dt_nevents`	The number of events in the event mask, `dt_emask`.
`dt_pers`	A boolean indicating whether the file does or does not have associated persistent opaque data management attributes. A non-zero value indicates presence of persistent attributes.
`dt_pmanreg`	A boolean indicating whether the file does or does not have associated persistent managed regions. A non-zero value indicates presence of persistent managed regions.
`dt_dtime`	The value of the data management attribute timestamp if `dt_pers` is a non-zero value. This field will be set to zero if persistent attributes are not supported by the DMAPI or associated with this file.
`dt_change`	This field is a file change indicator that is changed on each file modification (of either data or metadata). It can be used by DM applications during lock upgrade to determine whether the file has changed state. If the same value is returned in `dt_change` between two calls, the file is guaranteed not to have changed; however, some implementations of the DMAPI may return a different `dt_change` value even if the file has not really been modified.

The following fields are taken from the Spec 1170 `struct stat`.

`dt_dev`	ID of device containing file
`dt_ino`	File serial number
`dt_mode`	Mode of file
`dt_nlink`	Number of links to the file
`dt_uid`	User ID of file
`dt_gid`	Group ID of file
`dt_rdev`	Device ID (if file is character/block special)
`dt_size`	File size in bytes (if file is regular file)
`dt_atime`	Time of last access
`dt_mtime`	Time of last data modification
`dt_ctime`	Time of last status change
`dt_blksize`	A file system specific preferred I/O block size for this object. In some file system types, this may vary from file to file.
`dt_blocks`	Number of blocks of a file system-specific size allocated for this object.

5.24 dm_token_t

Description

A `dm_token_t` is an opaque scalar. There are two distinguished values, `DM_NO_TOKEN`, which signifies automatic token acquition, and `DM_INVALID_TOKEN`, which signifies no legal value.

5.25 dm_vardata_t

```
struct dm_vardata {
   ssize_t  vd_offset;
   size_t   vd_length;
};
typedef struct dm_vardata     dm_vardata_t;
```

Description

The offset field (`vd_offset`) in `dm_vardata_t` records the distance in bytes from the beginning of the structure in which the variable length data begins. In the case of an event message, it records the beginning of the actual event-specific data. For other structures, such as `dm_stat_t`, it indicates where the handle data can be found.

5.26 Get Attribute Defines

```
#define DM_AT_HANDLE          xxx
#define DM_AT_EMASK           xxx
#define DM_AT_PMANR           xxx
#define DM_AT_PATTR           xxx
#define DM_AT_DTIMES          xxx
#define DM_AT_CFLAG           xxx
#define DM_AT_STAT            xxx
```

Description

The `#defines` shown above are used by the `dm_get_bulkattr()`, `dm_get_dirattrs()` and `dm_get_fileattr()` functions. They are OR-able flags. For descriptions of these flags, see the man pages for these functions.

5.27 Get Config Defines

```
#define DM_CONFIG_MAX_ATTRIBUTE_SIZE      xxx
#define DM_CONFIG_MAX_MANAGED_REGIONS     xxx
#define DM_CONFIG_MAX_MESSAGE_DATA        xxx
#define DM_CONFIG_PERS_ATTRIBUTES         xxx
#define DM_CONFIG_PERS_EVENTS             xxx
#define DM_CONFIG_PERS_MANAGED_REGIONS    xxx
#define DM_CONFIG_PUNCH_HOLE              xxx
#define DM_CONFIG_TOTAL_ATTRIBUTE_SPACE   xxx
#define DM_CONFIG_WILL_RETRY              xxx
#define DM_CONFIG_PUNCH_HOLE              xxx
```

Description

The `#defines` shown above are used by the `dm_get_config()` function. For descriptions of these flags, see the man pages for this function.

5.28 Miscellaneous Defines

```
#define DM_SESSION_INFO_LEN    xxx
#define DM_NO_SESSION          xxx

#define DM_WAIT                xxx
#define DM_NOWAIT              xxx

#define DM_TRUE                xxx
#define DM_FALSE               xxx

#define DM_INVALID_TOKEN       xxx
#define DM_NO_TOKEN            xxx

#define DM_GLOBAL_HANP         xxx
#define DM_GLOBAL_HLEN         xxx

#define DM_UNMOUNT_FORCE       xxx
```

Description

The `#defines` shown above are referenced by various DMAPI routines. The DMAPI does not mandate their value, as is indicated by the *xxx*. Each implementation of the DMAPI is free to chose a suitable value.

`DM_SESSION_INFO_LEN` is the length of the string that can be associated with the session on the `dm_create_session()` call. 256 bytes is the recommended length.

`DM_NO_SESSION` is used on `dm_create_session()` to indicate that session assumption is not wanted. The implementation of the DMAPI should choose a meaningful value.

`DM_WAIT` and `DM_NOWAIT` are used on `dm_get_events()` and `dm_request_right()` to indicate if the calling process should block.

`DM_TRUE` and `DM_FALSE` are boolean values used in several functions such as `dm_get_bulkattr()` and `dm_get_fileattr()`.

`DM_INVALID_TOKEN` is a special token value which signifies no legal token value.

`DM_NO_TOKEN` is a special token value which is used in calls to functions which require a token but the caller wants the DM implementation to automatically acquire the appropriate rights.

`DM_GLOBAL_HANP` and `DM_GLOBAL_HLEN` are used to specify the global handle.

`DM_UNMOUNT_FORCE` is used to specify a forced unmount mode in *`preunmount`* and *`unmount`* events.

6.0 DMAPI Definitions

The following pages describe the interfaces of the DMAPI in detail. The interfaces are given in man page format, and are in alphabetical order. In the man pages, parameters are described as follows:

- `(I)` indicates an input parameter
- `(O)` indicates an output parameter.
- `(I/O)` indicates both an input and an output parameter.

Many of the following routines reference data structures that are defined by the DMAPI. To provide a consistent API from platform to platform, these data structures will be defined in a header file named `dmapi.h`. DM applications that need to use these routines must include this header file. The specification of this header file will not be shown on the following function definitions, but is of the following format:

```
#include <dmapi.h>
```

There are two paradigms used for DMAPI functions; *iterative* , which allow DM applications to make repeated calls to the same function to obtain all available data, and *non-iterative* , which require a DM application to provide a buffer large enough to hold all available data in a single call.

The errno values specified in these man pages are the recommended values. A DMAPI implementation may define others.

6.1 Non-iterative Functions

Several functions accept and return variable length data structures, such as dm_get_dmattr(). Other functions may return a variable number of structures in a list, such as dm_getall_inherit(). Both these function types use the *E2BIG* paradigm. In this paradigm, the DM application specifies the size of the buffer in one of the input arguments. If the buffer is not large enough to hold the requested information, then no data will be copied, the error `E2BIG` is returned, and one of the output arguments will be updated to indicate the required size of the buffer.

There are two distinct flavors of the *E2BIG* paradigm; functions that return variable sized data structures, and those that return a variable number of fixed size structures. Generally, functions that return variable sized structures specify the size of the input buffer in bytes via the `buflen` parameter. If `E2BIG` is returned, then the parameter `rlenp` is set to the required buffer size. These types of functions take a `void *` pointer to a buffer.

Functions that return fixed size structures specify the number of elements that will fit in the input buffer via the `nelem` parameter. If `E2BIG` is returned, the `nelemp` parameter is updated to indicate the number of elements that are available. DM applications can use this return value to resize their buffer. These types of functions take a buffer pointer to a specific type.

This is summarized in the following table:

Parameters	Function type	Buffer Type
`buflen / rlenp`	variable *length* structures	void *
`nelem / nelemp`	variable *number* of structures	specific structure pointer

Table 7: Non-iterating function types

6.2 Iterating Functions

Several DMAPI functions are *iterating* functions, allowing the DM application to make a number of calls to the same function to retrieve all the available data. These functions typically contain an offset parameter, providing the DM application with the ability to specify the starting point in a file or file system for data retrieval.

These functions return -1 on error and set the global `errno` to indicate the cause of error. If a zero is returned, the function completed successfully, and no more data is available. If the function completed successfully, but more data is available, a value of one is returned.

The following functions are the only iterating functions:

Function Name
dm_getalloc_info()
dm_get_bulkattr()
dm_get_dirattrs()

Table 8: Iterating functions

Name

DMEV_CLR() — removes an event from an event list
DMEV_ISSET() — determines if an event is set in an event list
DMEV_SET() — adds an event to an event list
DMEV_ZERO() — determines if an event is not set in an event list

Synopsis

```
#define DMEV_SET(event_type, event_list)

#define DMEV_CLR(event_type, event_list)

#define DMEV_ISSET(event_type, event_list)

#define DMEV_ZERO(event_list)
```

Description

These macros manipulate event sets as defined by dm_eventset_t. DMEV_SET() adds the specified event_type to the event_list, while DMEV_CLR() removes the specified event from the list. DMEV_ISSET() evaluates to non-zero if the specified event_type is a member of the event_list set, otherwise it evaluates to zero. DMEV_ZERO() initializes the event_list to contain no events.

See Also

dm_get_eventlist(), dm_set_eventlist()

Name

dm_clear_inherit() — clear an attribute's inherit-on-create status

Synopsis

```
int
dm_clear_inherit(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_attrname_t  *attrnamep)
```

Description

The function dm_clear_inherit() marks the named attribute as no longer inheritable on the specified file system.

dm_sessid_t sid (I)
A session ID.

void *hanp (I)
The filesystem handle.

size_t hlen (I)
The length of the handle in bytes.

dm_token_t token (I)
The token referencing the access right for the file system handle. The access right must be DM_RIGHT_EXCL or the token DM_NO_TOKEN may be used and the interface will acquire the appropriate rights.

dm_attrname_t *attrnamep (I)
The attribute name that will marked as no longer inheritable.

Return Value

Zero is returned on success. On error, -1 is returned, and the global errno is set to one of the following values:

[EACCES]	The access right referenced by the token for the handle is not DM_RIGHT_EXCL.
[EBADF]	The filesystem handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The argument attrname specifies an attribute that has not been marked inheritable.
[EINVAL]	The argument token is not a valid token.
[EINVAL]	The session is not valid.
[EIO]	The attempt to clear the inheritable attribute resulted in an I/O error.
[ENOSYS]	The DMAPI implementation does not support this optional function.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

dm_set_inherit(), dm_getall_inherit()

Notes

dm_clear_inherit() is an optional component of the DMAPI.

Name

`dm_create_session()` — create a session

Synopsis

```
int
dm_create_session(
   dm_sessid_t     oldsid,
   char           *sessinfop,
   dm_sessid_t    *newsidp)
```

Description

dm_create_session() establishes a DMAPI session. Sessions are guaranteed to be unique as long as the system that grants them is up; sessions are not unique across reboots. Applications should treat the session id as opaque.

To later identify the session during recovery, an arbitrary string can be associated with the session. This NULL-terminated string must be at most `DM_SESSION_INFO_LEN` bytes (including the NULL) or less. The `sessinfop` parameter is an arbitrary string; the DMAPI assign no semantics to it whatsoever. It is not an error for more than one session to have identical `sessinfop` strings.

To support recovery, the DMAPI supports the notion of session assumption. This is provided by the `oldsid` parameter. If specified, the DMAPI will atomically update all messages that refer to `oldsid` to be associated with a new session id, which is returned to the caller. When `dm_create_session()` returns, `oldsid` is no longer valid; attempts to use it in subsequent DMAPI calls will result in an error.

To determine a previous session identifier, the DM application can use `dm_getall_sessions()` and `dm_query_session()` to find a specific old session identifier. If `oldsid` is provided, then when `dm_create_session()` returns, the new session has assumed what was an active, open session. It is not necessary to register to receive events; the new session has assumed the context of the previous session, including all default event registrations. If a `sessinfop` string is provided, it replaces any existing `sessinfop` string associated with the old session.

Sessions are not tied to a particular process. Once a session has been established, any processes with sufficient permissions (such as the super-user) may use the session. This permits an application to hand a session off to another process for further operations. The hand off can be accomplished merely by conveying the value of the session ID to the other process via any mechanism (shared memory, message, pipe, file, etc.). Because sessions are not tied to any particular process, no DMAPI calls are needed to accomplish session hand off.

`dm_sessid_t  oldsid (I)`
: A value previously returned by `dm_create_session()`, or `DM_NO_SESSION` if no assumption is desired.

`char  *sessinfop (I)`
: A NULL-terminated string of at most `DM_SESSION_INFO_LEN` to be associated with the session.

`dm_sessid_t  *newsidp (O)`
: The new session id created by the DMAPI.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[E2BIG]	`sessinfop` string was longer than `DM_SESSION_INFO_LEN` bytes.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EINVAL]	The argument `oldsid` contains an invalid session id.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_destroy_session()`, `dm_set_disp()`, `dm_getall_sessions()`, `dm_query_session()`

Name

`dm_create_userevent()` — generate a user pseudo-event message

Synopsis

```
int
dm_create_userevent(
   dm_sessid_t     sid,
   size_t          msglen,
   void           *msgdatap,
   dm_token_t     *tokenp)
```

Description

Tokens are always associated with an event message. If a DM application needs to obtain a token to get specific access rights to an object, it must generate a pseudo-event message to define a context for that token.

DM applications can use the user defined message data to record information about the state of the operation they are performing. This information could be useful during recovery processing. The contents of the message data is opaque to the DMAPI.

`dm_create_userevent()` allows a token to be associated with arbitrary message data. The DMAPI will create a new token, and associate it with an event message whose data is given by the `msgp` parameter. The message will be enqueued on the specified session, and will be in the same state as synchronous messages that have been received by DM applications via `dm_get_events()` but not yet responded to. The type of the message will be set by the DMAPI to be `DM_EVENT_USER`. The token will be valid until the DM application does a `dm_respond_event()` on the token.

The generated token will not reference any access rights for any object handles. DM applications must use `dm_request_right()` to obtain access rights.

`dm_sessid_t  sid (I)`
: The session to be used for the newly created message.

`size_t  msglen (I)`
: The length of the message data in bytes.

`void  *msgdatap (I)`
: The data that is to be included in the message.

`dm_token_t  *tokenp (O)`
: A newly created token identifying the message.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[E2BIG]	The user message data is too big to fit into implementation defined limits. The limit can be determined by calling the `dm_get_config()` function using the value `DM_CONFIG_MAX_MESSAGE_DATA`.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The session is not valid.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

3042 **See Also**

3043 `dm_request_right()`, `dm_send_msg()`

Name

`dm_destroy_session()` — destroy the specified session

Synopsis

```
int
dm_destroy_session(
   dm_sessid_t    sid)
```

Description

To perform a graceful shutdown, DM applications must shut down and destroy a session as part of their termination procedures. Upon successful return from `dm_destroy_session()` the session id is invalid.

Unless the file system is quiescent or the session is not registered for any events, `dm_destroy_session()` will not be able to invalidate the session id. If there are any outstanding events, or if events are still being generated, `dm_destroy_session()` will fail.

`dm_sessid_t  sid (I)`
The session id to destroy.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[EBUSY]	There are outstanding events for the session that have either not been delivered, or there are synchronous events that have not been responded to.
[EINVAL]	The session is not valid.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_create_session()`

Name

dm_find_eventmsg() — get the message for the event

Synopsis

```
int
dm_find_eventmsg(
   dm_sessid_t     sid,
   dm_token_t      token,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Description

Obtains the message that is associated with the token. It will only be possible to obtain messages that are outstanding, i.e., messages that have been delivered, but not responded to.

Tokens are always associated with an event message; receipt of a message is the only way a DM application can obtain a token. Once a message has been responded to, the token (and the corresponding message) is no longer valid.

dm_sessid_t sid (I)
: The session to be queried.

dm_token_t token (I)
: The token that corresponds to an outstanding event message.

size_t buflen (I)
: The length of the message buffer in bytes.

void *bufp (O)
: The buffer that should be filled in with a dm_eventmsg structure.

size_t *rlenp (O)
: The size of the requested information. If the buffer is not large enough to hold the requested information (as indicated by the buflen parameter), then no data will be copied, E2BIG is returned, and rlenp is set to the required size. rlenp is updated if either the call was successful, or if the call fails with E2BIG, but is undefined otherwise.

Return Value

Zero is returned on success. On error, -1 is returned, and the global errno is set to one of the following values:

[E2BIG]	The information is too large to fit into the buffer.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The session or token is not valid.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

dm_getall_tokens(), dm_getall_sessions(), dm_respond_event(), dm_query_session()

Name

`dm_get_allocinfo()` — return the current residency information for a file

Synopsis

```
int
dm_getalloc_info(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_off_t       *offp,
   u_int           nelem,
   dm_extent_t    *extentp,
   u_int          *nelemp)
```

Description

`dm_get_allocinfo()` returns residency information for a file. This call is iterative in nature and allows the application to begin retrieving extent information about a file at any byte offset.

The `dm_extent_t` structures returned by the call to `dm_get_allocinfo()` contain the fields `ex_type` that contain one of two indicators for the type of extent. `DM_EXTENT_RES` indicates the portion of the file defined by this extent is resident. `DM_EXTENT_HOLE` indicates the extent is not resident. The value `ex_offset` is the byte offset into the file where the extent begins, and `ex_length` is the byte count of the extent.

`dm_sessid_t  sid (I)`
: A session ID.

`void  *hanp (I)`
: The handle for the target file.

`size_t  hlen (I)`
: The length of the handle in bytes.

`dm_token_t  token (I)`
: The token referencing the access right for the handle. The access right must be at least `DM_RIGHT_SHARED` or the token `DM_NO_TOKEN` may be used and the interface will acquire the appropriate rights.

`dm_off_t  *offp (I/O)`
: The byte address of the beginning of the extent following the last returned extent and may point beyond the current size of the file. Assigning zero to `*offp` causes the initial extents of the file to be returned.

`u_int  nelem (I)`
: The number of elements in the extent array.

`dm_extent_t  *extentp (O)`
: The residency information for the file.

`u_int  *nelemp (O)`
: The number of elements returned.

Return Value

Zero indicates success and that no more information is available. 1 is returned on success and indicates more information is available. On error, -1 is returned and the global `errno` is set to indicate the error.

[EACCES]	The access right referenced by the token for the handle is not at least `DM_RIGHT_SHARED`.
[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	`nelem` is zero and must be at least one.
[EINVAL]	The argument `offp` does not point to a valid extent.
[EINVAL]	The file handle does not refer to a regular file.
[EINVAL]	The session is not valid.
[EINVAL]	The argument `token` is not a valid token.
[EIO]	An attempt to read the file residency information resulted in an I/O error.
[ENOSYS]	The DMAPI implementation does not support this optional function.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_punch_hole()`, `dm_probe_hole()`

Name

`dm_get_bulkattr()` — gets bulk attributes for an entire file system
`dm_get_dirattrs()` — gets bulk attributes for a directory
`dm_init_attrloc()` — initializes a bulk attribute location offset

Synopsis

```
int
dm_get_bulkattr(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           mask,
   dm_attrloc_t   *locp,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)

int
dm_get_dirattrs(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           mask,
   dm_attrloc_t   *locp,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)

int
dm_init_attrloc(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_attrloc_t   *locp)
```

Description

The function `dm_get_bulkattr()` retrieves both standard file attributes and DM specific file attributes for the file system specified by the argument `hanp`. The `bufp` argument will be filled with one or more `dm_stat` structures on return. The argument `mask` indicates which fields of the `dm_stat` structure to return. For fields that won't be returned, the corresponding fields of the `dm_stat` structure are undefined; the `dt_compname` field is always undefined. The file system specified by `hanp` must be mounted.

The `dm_get_dirattrs()` routine performs a similar function for directories. It returns the file name, attributes, and DM specific file attributes for the files in the directory specified by `hanp`. The `bufp` argument will be filled with one or more `dm_stat` structures on return. The argument `mask` indicates which fields of the `dm_stat` structure to return. For fields that won't be returned, the corresponding fields of the `dm_stat` structure are undefined. The `dt_compname` field is always filled in. The directory specified by `hanp` must be on a mounted file system. If a file is named several times within the directory (i.e., multiple links to the same file), a `dm_stat` structure is returned for each one.

These functions can be repeated and each time fill the buffer, `bufp`, with as many `dm_stat` structures as can be returned, until all `dm_stat` structures have been returned. The argument

locp must be initialized via dm_init_attrloc() before calling dm_get_bulkattr() or dm_get_dirattrs(). The same handle must be used for initializing locp as is used for retrieving the attributes.

The arguments to **dm_get_bulkattr()** are as follows:

dm_sessid_t sid (I)
A session ID.

void *hanp (I)
The handle for the file system. Information on all allocated files is returned; unallocated inodes are skipped.

size_t hlen (I)
The length of the handle in bytes.

dm_token_t token (I)
The token referencing the access right for the handle. The access right must be at least DM_RIGHT_SHARED or the token DM_NO_TOKEN may be used and the interface will acquire the appropriate rights.

u_int mask (I)
The mask argument controls which fields in the dm_stat structure should be returned. The mask is constructed by OR-ing together one or more of the following flags:

DM_AT_HANDLE	The file's handle will be returned.
DM_AT_EMASK	The file's persistent event mask will be returned.
DM_AT_PMANR	A boolean (DM_TRUE or DM_FALSE) will be returned indicating where the file has persistent managed regions.
DM_AT_PATTR	A boolean (DM_TRUE or DM_FALSE) will be returned indicating whether the file has persistent data management attributes.
DM_AT_DTIMES	The time stamp of the persistent attributes if associated with the file.
DM_AT_CFLAG	The file change indicator will be returned.
DM_AT_STAT	The file's general attributes as defined by the Spec 1170 struct stat.

dm_attrloc_t *locp (I/O)
This is an offset, which is opaque to the calling DM application, that can be used by the DMAPI implementation to indicate the location in the object. It should not be modified by the DM application. It must be initialized via dm_init_attrloc() before calling dm_get_bulkattr() or dm_get_dirattrs().

size_t buflen (I)
The size of the buffer, in bytes.

`void *bufp (O)`
This is filled in with the `dm_stat` information for the file system.

`size_t *rlenp (O)`
The size of the returned information. If the buffer is not large enough to hold the requested information (as indicated by the `buflen` parameter), then no data will be copied, `E2BIG` is returned, and `rlenp` is set to the required size. `rlenp` is updated if either the call was successful, or if the call fails with `E2BIG`, but is undefined otherwise.

The arguments to **`dm_get_dirattrs()`** are identical to `dm_get_bulkattr()` with the exception of the specification of the file handle:

`void *hanp (I)`
The handle for the directory. Information on all the files in the directory is returned. Subdirectories are not traversed. If a subdirectory is a mount point, `dm_get_dirattrs()` returns information on the underlying mounted-on subdirectory, not the new mount point.

The arguments to **`dm_init_attrloc()`** are as follows:

`dm_sessid_t sid (I)`
A session ID.

`void *hanp (I)`
The handle for the file system or the directory.

`size_t hlen (I)`
The length of the handle in bytes.

`dm_token_t token (I)`
The token referencing the access right for the handle. The access right must be at least `DM_RIGHT_SHARED` or the token `DM_NO_TOKEN` may be used and the interface will acquire the appropriate rights.

`dm_attrloc_t *locp(O)`
Pointer to an offset cookie to initialize.

Return Value

`dm_get_bulkattr()` and `dm_get_dirattrs()` both return 1 to indicate success and more information is available. zero to indicate success and no more information is available. -1 is returned on error; the global `errno` is set to indicate the error. `dm_init_attrloc()` returns a value of zero upon successful completion. Otherwise a value of -1 is returned and `errno` is set to one of the following values:

[E2BIG]	The information is too large to fit into the buffer.
[EACCES]	The access right referenced by the token for the handle is not at least `DM_RIGHT_SHARED`.
[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The argument `locp` or `mask` is not a valid value.
[EINVAL]	The argument `token` is not a valid token.
[EINVAL]	The file handle does not represent a directory or the file system object.

[EINVAL]	The session is not valid.
[EIO]	An attempt to read the attributes resulted in an I/O error.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_punch_hole()`, `dm_get_allocinfo()`, `dm_get_fileattr()`

Name

`dm_get_config()` — return DMAPI implementation details and limits

Synopsis

```
int
dm_get_config(
   void           *hanp,
   size_t          hlen,
   dm_config_t     flagname,
   dm_size_t      *retvalp)
```

Description

The function `dm_get_config()` returns information specific to an implementation of the DMAPI.

`void *hanp (I)`
A handle for any file on the file system.

`size_t hlen (I)`
The length of the handle in bytes.

`dm_config_t flagname (I)`
The argument `flagname` is an enum and may contain one of the following values :

`DM_CONFIG_MAX_ATTRIBUTE_SIZE`
The maximum size in bytes per file that a single persistent attribute can occupy is returned. This is the size of the attribute data only.

`DM_CONFIG_MAX_MANAGED_REGIONS`
The maximum number of managed regions supported per file is returned.

`DM_CONFIG_MAX_MESSAGE_DATA`
The number of bytes of data in the largest user created event message. DM applications can use this value determine the largest buffer size allowed in calls to `dm_create_userevent()` and `dm_send_msg()`.

`DM_CONFIG_PERS_ATTRIBUTES`
A boolean value indicating whether the file system does or does not support persistent, opaque data management attributes.

`DM_CONFIG_PERS_EVENTS`
A boolean value is returned that indicates whether the DMAPI implementation supports / does not support persistent event masks.

`DM_CONFIG_PERS_MANAGED_REGIONS`
A boolean value is returned that indicates whether the DMAPI implementation supports / does not support persistent managed regions.

DM_CONFIG_PUNCH_HOLE

A boolean value indicating whether the file system does or does not support punching holes (see dm_punch_hole()).

DM_CONFIG_TOTAL_ATTRIBUTE_SPACE

The total available space per file for the storage of all persistent data management attributes, in bytes.

DM_CONFIG_WILL_RETRY

A boolean value is returned that indicates whether the DMAPI implementation will / will not retry the operation that caused a *nospace* event to be generated if a DM_RESP_CONTINUE response is returned.

For boolean values, a non-zero return value indicates that the property is supported, while zero indicates that the property is not supported. Other return values are dependent on the value of the flagname argument.

dm_size_t *retvalp (O)

The value of the requested flag if the call succeeds. If dm_get_config() returns -1, the value is undefined.

Return Value

Zero is returned on success. On error, -1 is returned, and the global errno is set to one of the following values:

[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The argument flag_name does not contain a valid option.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

dm_punch_hole(), dm_get_allocinfo()

Name

dm_get_dmattr() — retrieve a data management attribute

Synopsis

```
int
dm_get_dmattr(
    dm_sessid_t      sid,
    void            *hanp,
    size_t           hlen,
    dm_token_t       token,
    dm_attrname_t   *attrnamep,
    size_t           buflen,
    void            *bufp,
    size_t          *rlenp)
```

Description

The dm_get_dmattr() function retrieves a single, specific data management attribute for a file. The file's attribute timestamp is not altered.

dm_sessid_t sid (I)
: A session ID.

void *hanp (I)
: The handle for the file for which the attributes should be retrieved.

size_t hlen (I)
: The length of the handle in bytes.

dm_token_t token (I)
: The token referencing the access right for the handle. The access right must be at least DM_RIGHT_SHARED or the token DM_NO_TOKEN may be used and the interface will acquire the appropriate rights.

dm_attrname_t *attrnamep (I)
: The attribute to be retrieved.

size_t buflen (I)
: The size of the buffer in bytes.

void *bufp (O)
: The buffer to be filled in with the value of the specified attribute's data.

size_t *rlenp (O)
: The size of the requested information. If the buffer is not large enough to hold the requested information (as indicated by the buflen parameter), then no data will be copied, E2BIG is returned, and rlenp is set to the required size. rlenp is updated if either the call was successful, or if the call fails with E2BIG, but is undefined otherwise.

Return Value

Zero is returned on success. On error, -1 is returned, and the global errno is set to one of the following values:

[E2BIG]	The information is too large to fit into the buffer.
[EACCES]	The access right referenced by the token for the handle is not at least DM_RIGHT_SHARED.

[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The argument `token` is not a valid token.
[EINVAL]	The session is not valid.
[ENOENT]	The attribute was not found.
[ENOMEM]	The DMAPI could not acquire the required resources to complete the call.
[ENOSYS]	The DMAPI implementation does not support this optional function.
[EIO]	An attempt to read the attribute resulted in an I/O error.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_set_dmattr()`, `dm_remove_dmattr()`, `dm_getall_dmattr()`

Notes

`dm_get_dmattr()` is an optional component of the DMAPI.

Name

`dm_get_eventlist()` — get the list of enabled events for an object

Synopsis

```
int
dm_get_eventlist(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           nelem,
   dm_eventset_t  *eventsetp,
   u_int          *nelemp)
```

Description

Return the list of enabled events for the specified object. The object can be either a regular file or the file system object. The returned event list is the set of all events for the object. DM applications can use the `DMEV_ISSET()` macro to determine if a specific event is enabled.

For DMAPI implementations that do not store event lists persistently, `dm_get_eventlist()` will return only those events which have been set via `dm_set_eventlist()` during the life of this session. It will not cause a *`debut`* event to be generated.

The synchronous managed region events are set via the `dm_set_region()` interface. If there are any managed regions that have any events set (as determined by the `rg_flags` field in the `struct dm_region`), `dm_get_eventlist()` will return the union of all of those flags fields.

`dm_sessid_t  sid (I)`
: A session ID.

`void  *hanp (I)`
: The handle for the object.

`size_t  hlen (I)`
: The length of the handle in bytes.

`dm_token_t  token (I)`
: The token referencing the access right for the handle. The access right must be at least `DM_RIGHT_SHARED` or the token `DM_NO_TOKEN` may be used and the interface will acquire the appropriate rights.

`u_int  nelem (I)`
: The number of elements in the event set array.

`dm_eventset_t  *eventsetp (O)`
: Buffer to be filled in with the list of events.

`u_int  *nelemp (O)`
: The number of elements returned.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[E2BIG]	The information is too large to fit into the buffer. `nelem` needs to be at least as large as specified in `nelemp`..
[EACCES]	The access right referenced by the token for the handle is not at least `DM_RIGHT_SHARED`.
[EBADF]	The handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The argument `token` is not a valid token.
[EINVAL]	The session is not valid.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_set_eventlist()`, `dm_set_disp()`, `dm_get_events()`, `DMEV_ISSET()`

Name

`dm_get_events()` — get the next available event messages

Synopsis

```
int
dm_get_events(
    dm_sessid_t     sid,
    u_int           maxmsgs,
    int             nowait,
    size_t          buflen,
    void           *bufp,
    size_t         *rlenp)
```

Description

Get the next event or events for a session. The event message is copied to the message buffer, and includes both the common and event-specific fields of the event.

Messages can be retrieved in bulk; up to maxmsgs can be copied to the output buffer, providing that enough space is available.

If no messages are immediately available, and the `nowait` flag is not set, then the calling process will block interruptibly.

`dm_sessid_t  sid (I)`
The session from which the next message(s) should be obtained.

`u_int  maxmsgs (I)`
The maximum number of messages that should be transferred in a single call. A value of zero indicates return all available messages that will fit into the message buffer.

`int  nowait (I)`
If set to `DM_NOWAIT`, the calling process will return with `EAGAIN` if no messages are immediately available. If `DM_WAIT`, the calling process will block interruptibly, waiting for messages to be enqueued on the session.

`size_t  buflen (I)`
The size of the message buffer in bytes. Must be large enough to hold at least one maximum sized event message.

`void  *bufp (O)`
The message buffer to be filled in with one or more `dm_eventmsg` structures.

`size_t  *rlenp (O)`
The size of the requested information. If the buffer is not large enough to hold even the first event message, then no data will be copied, `E2BIG` is returned, and `rlenp` is set to the required size. `rlenp` is updated if either the call was successful, or if the call fails with `E2BIG`, but is undefined otherwise.

Return Value

Zero indicates success. On error, -1 is returned and the global `errno` is set to indicate the error.

[E2BIG]	The information is too large to fit into the buffer.
[EAGAIN]	The `nowait` flag was `DM_NOWAIT`, and no messages are immediately available.

[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The session is not valid.
[EINTR]	The process was blocked waiting for messages and was interrupted.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_set_eventlist(), dm_set_disp()`

Name

`dm_get_fileattr()` — return file attributes

Synopsis

```
int
dm_get_fileattr(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           mask,
   dm_stat_t      *statp)
```

Description

The function `dm_get_fileattr()` retrieves both standard file attributes and DM specific file attributes for the file specified by handle.

`dm_sessid_t  sid (I)`
A session ID.

`void  *hanp (I)`
The handle for the file for which the file attributes should be retrieved.

`size_t  hlen (I)`
The length of the handle in bytes.

`dm_token_t  token (I)`
The token referencing the access right for the handle. The access right must be at least `DM_RIGHT_SHARED` or the token `DM_NO_TOKEN` may be used and the interface will acquire the appropriate rights.

`u_int  mask (I)`
The argument `mask` indicates which fields of the `dm_stat` structure to return. The mask is constructed by OR-ing together one or more of the following flags:

`DM_AT_EMASK`	The file's persistent event mask will be returned.
`DM_AT_PMANR`	A boolean (`DM_TRUE` or `DM_FALSE`) will be returned indicating whether the file has persistent managed regions.
`DM_AT_PATTR`	A boolean (`DM_TRUE` or `DM_FALSE`) will be returned indicating whether the file has persistent data management attributes.
`DM_AT_DTIMES`	The time stamp of the persistent attributes if associated with the file.
`DM_AT_CFLAG`	The file change indicator will be returned.
`DM_AT_STAT`	The file's general attributes as defined by the Spec 1170 struct stat.

`dm_stat_t *statp (O)`
The buffer to be filled in with the required information. For fields which won't be returned the corresponding fields of the `dm_stat` structure are undefined.

Return Value

Zero is returned on sucess. On error, -1 is returned, and the global `errno` is set to one of the following values:

[EACCES]	The access right referenced by the token for the handle is not at least `DM_RIGHT_SHARED`.
[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The argument `mask` is not valid.
[EINVAL]	The argument `token` is not a valid token.
[EINVAL]	The session is not valid.
[EIO]	An attempt to read the attributes resulted in an I/O error.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_punch_hole()`, `dm_get_allocinfo()`, `dm_get_bulkattr()`, `dm_get_dirattr()`

Name

`dm_get_mountinfo()` — return the information that was delivered on a mount event

Synopsis

```
int
dm_get_mountinfo(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Description

During initialization and often during recovery, DM applications will need to get information about the file systems they are operating on. To facilitate this, `dm_get_mountinfo()` provides the same information that was originally generated on a *`mount`* event. This information will most likely be kept in a static area in the DMAPI implementation; thus it is not a large burden on the implementation to return it both in a message and via this interface.

`dm_sessid_t  sid (I)`
: A session ID.

`void  *hanp (I)`
: The handle for the file system meta-object.

`size_t  hlen (I)`
: The length of the handle in bytes.

`size_t  buflen(I)`
: The length of the input buffer in bytes.

`void  *bufp (O)`
: A pointer to a buffer that is to be filled in with the information. The buffer should point to a `dm_namesp_event` structure, since that is how the *`mount`* event is specified.

`size_t *rlenp (O)`
: The size of the requested information. If the buffer is not large enough to hold the requested information (as indicated by the `buflen` parameter), then no data will be copied, `E2BIG` is returned, and `rlenp` is set to the required size. `rlenp` is updated if either the call was successful, or if the call fails with `E2BIG`, but is undefined otherwise.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[E2BIG]	The buffer is not large enough to hold the requested information.
[EBADF]	The handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The handle is not a file system handle.
[EINVAL]	The session is not valid.
[EIO]	An attempt to read the mount information resulted in an I/O error.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.

[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_set_disp()`, *`mount`* event

Name

`dm_get_region()` — get the managed regions for a file

Synopsis

```
int
dm_get_region(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           nelem,
   dm_region_t    *regbufp,
   u_int          *nelemp)
```

Description

Get the set of managed regions for a file.

`dm_sessid_t  sid (I)`
A session ID.

`void  *hanp (I)`
The handle for the regular file to be interrogated.

`size_t  hlen (I)`
The length of the handle in bytes.

`dm_token_t  token (I)`
The token referencing the access right for the handle. The access right must be at least `DM_RIGHT_SHARED` or the token `DM_NO_TOKEN` may be used and the interface will acquire the appropriate rights.

`u_int  nelem (I)`
The number of elements in the region buffer.

`dm_region_t  *regbufp (O)`
A pointer to the structure defining the regions to be filled in.

`u_int  *nelemp (O)`
The number of elements read. If the buffer is not large enough to hold the requested information (as determined by the `nelem` parameter), then no data will be copied, `E2BIG` is returned, and `nelemp` is set to the number of elements that are available. `nelemp` is updated if either the call was successful, or if the call fails with `E2BIG`, but is undefined otherwise.

Return Values

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[E2BIG]	The information is too large to fit into the buffer.
[EACCES]	The access right referenced by the token for the handle is not at least `DM_RIGHT_SHARED`.
[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The handle does not refer to a regular file.
[EINVAL]	The session is not valid.
[EINVAL]	The argument `token` is not a valid token.

[ENOMEM]	The DMAPI could not acquire the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_set_region()`

Name

`dm_getall_disp()` — get disposition of events for all file systems for a session

Synopsis

```
int
dm_getall_disp(
   dm_sessid_t     sid,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Description

Get the set of all dispositions of events for all file systems for the indicated session. Since a session may be receiving events on more than one file system, `dm_getall_disp()` will return the disposition of events for every file system that was specified in a `dm_set_disp()` call.

`dm_sessid_t  sid (I)`
The session id to return the information for.

`size_t  buflen (I)`
The length of the input buffer in bytes.

`void  *bufp (O)`
The buffer to be filled in with `dm_dispinfo` structures.

`size_t  *rlenp (O)`
The size of the requested information. If the buffer is not large enough to hold the requested information (as indicated by the `buflen` parameter), then no data will be copied, `E2BIG` is returned, and `rlenp` is set to the required size. `rlenp` is updated if either the call was successful, or if the call fails with `E2BIG`, but is undefined otherwise.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[E2BIG]	The information is too large to fit into the buffer.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The session is not valid.
[ENOMEM]	The DMAPI could not acquire the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_set_disp()`

Name

dm_getall_dmattr() — retrieve all of a file's data management attributes

Synopsis

```
int
dm_getall_dmattr(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Description

The dm_getall_dmattr() function returns **all** the attributes and attribute data associated with the specified file system object.

dm_sessid_t sid (I)
A session ID.

void *hanp (I)
The handle for the file for which the attributes should be retrieved.

size_t hlen (I)
The length of the handle in bytes.

dm_token_t token (I)
The token referencing the access right for the handle. The access right must be at least DM_RIGHT_SHARED or the token DM_NO_TOKEN may be used and the interface will acquire the appropriate rights.

size_t buflen (I)
The size of the buffer in bytes.

void *bufp (O)
The buffer to be filled in with dm_attrlist structures.

size_t *rlenp (O)
The size of the requested information. If the buffer is not large enough to hold the requested information (as indicated by the buflen parameter), then no data will be copied, E2BIG is returned, and rlenp is set to the required size. rlenp is updated if either the call was successful, or if the call fails with E2BIG, but is undefined otherwise.

Return Value

Zero is returned on success. On error, -1 is returned, and the global errno is set to one of the following values:

[E2BIG]	The information is too large to fit into the buffer.
[EACCES]	The access right referenced by the token for the handle is not at least DM_RIGHT_SHARED.
[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The argument token is not a valid token.
[EINVAL]	The session is not valid.

[EIO]	An attempt to read the attributes resulted in an I/O error.
[ENOMEM]	The DMAPI could not acquire the required resources to complete the call.
[ENOSYS]	The DMAPI implementation does not support this optional function.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_get_dmattr(), dm_attrlist_t`

Notes

`dm_getall_dmattr()` is an optional component of the DMAPI.

Name

dm_getall_inherit() — get all inheritable attributes for the specified file system

Synopsis

```
int
dm_getall_inherit(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           nelem,
   dm_inherit_t   *inheritbufp,
   u_int          *nelemp)
```

Description

The dm_getall_inherit() function returns a list of all inheritable attributes associated with the given file system object. The inheritable property of attributes is not persistent across reboots. The attributes are returned in a struct dm_inherit, which gives the attribute name and the file type that the attribute applies to.

dm_sessid_t sid (I)
: A session ID.

void *hanp (I)
: The file system handle.

size_t hlen (I)
: The length of the handle in bytes.

dm_token_t token (I)
: The token referencing the access right for the handle. The access right must be at least DM_RIGHT_SHARED or the token DM_NO_TOKEN may be used and the interface will acquire the appropriate rights.

u_int nelem (I)
: The number of elements to return. No more than nelem will be returned.

dm_inherit_t *inheritbufp (O)
: The buffer to be filled with inheritable attributes.

u_int *nelemp (O)
: The number of elements read. If the buffer is not large enough to hold the requested information (as determined by the nelem parameter), then no data will be copied, E2BIG is returned, and nelemp is set to the number of elements that are available. nelemp is updated if either the call was successful, or if the call fails with E2BIG, but is undefined otherwise.

Return Value

Zero is returned on success. On error, -1 is returned, and the global errno is set to one of the following values:

[E2BIG]	The information is too large to fit into the buffer.
[EACCES]	The access right referenced by the token for the handle is not at least DM_RIGHT_SHARED.
[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.

[EINVAL]	The argument `token` is not a valid token.
[EINVAL]	The handle is not a file system handle.
[EINVAL]	The session is not valid.
[EIO]	An attempt to read the attributes resulted in an I/O error.
[ENOSYS]	The DMAPI implementation does not support this optional function.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_set_inherit()`, `dm_clear_inherit()`, `dm_inherit_t`

Notes

`dm_getall_inherit()` is an optional component of the DMAPI.

Name

dm_getall_sessions() — get all extant sessions

Synopsis

```
int
dm_getall_sessions(
    u_int           nelem,
    dm_sessid_t    *sidbufp,
    u_int          *nelemp)
```

Description

This will obtain all the sessions that exist on a system.

u_int nelem (I)
The number of elements to return. No more than nelem will be returned.

dm_sessid_t *sidbufp (O)
The buffer where the active sessions are to be deposited.

u_int *nelemp (O)
The number of elements read. If the buffer is not large enough to hold the requested information (as determined by the nelem parameter), then no data will be copied, E2BIG is returned, and nelemp is set to the number of elements that are available. nelemp is updated if either the call was successful, or if the call fails with E2BIG, but is undefined otherwise.

Return Value

Zero is returned on success. On error, -1 is returned, and the global errno is set to one of the following values:

[E2BIG]	The information is too large to fit into the buffer.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

dm_find_eventmsg(), dm_create_session(), dm_getall_tokens(), dm_query_session()

Name

`dm_getall_tokens()` — get all outstanding tokens for a session

Synopsis

```
int
dm_getall_tokens(
   dm_sessid_t     sid,
   u_int           nelem,
   dm_token_t     *tokenbufp,
   u_int          *nelemp)
```

Description

This will get all the outstanding tokens that are associated with a session. An outstanding token is a token that corresponds to a synchronous message that has been delivered but not responded to. User event messages created with dm_create_userevent() are implicitly synchronous, and if tokens exist for these messages, they will also be returned.

If the session is active, the results of `dm_getall_tokens()` are not guaranteed to be exact, since event messages can be added and subtracted from the session at any point. It may not be possible to obtain the tokens for active sessions, and the size of the buffer required to hold all the tokens may keep increasing, causing the DM application to repeatedly receive `E2BIG`.

`dm_sessid_t  sid (I)`
: The session that is to be queried.

`u_int  nelem (I)`
: The number of elements to return. No more than `nelem` will be returned.

`dm_token_t  *tokenbufp (O)`
: The buffer that should be filled in with all outstanding tokens.

`u_int  *nelemp (O)`
: The number of elements read. If the buffer is not large enough to hold the requested information (as determined by the `nelem` parameter), then no data will be copied, `E2BIG` is returned, and `nelemp` is set to the number of elements that are available. `nelemp` is updated if either the call was successful, or if the call fails with `E2BIG`, but is undefined otherwise.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[E2BIG]	The information is too large to fit into the buffer.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The session is not valid.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_getall_sessions()`, `dm_find_eventmsg()`, `dm_respond_event()`, `dm_create_session()`

Name

`dm_handle_cmp()` — file handle comparison

Synopsis

```
int
dm_handle_cmp(
   void     *hanp1,
   size_t    hlen1,
   void     *hanp2,
   size_t    hlen2)
```

Description

Compare two handles for equality.

`void *hanp1 (I)`
The first handle to compare.

`size_t hlen1 (I)`
The length of the first handle in bytes.

`void *hanp2 (I)`
The second handle to compare.

`size_t hlen2 (I)`
The length of the second handle in bytes.

Return Value

< 0	hanp1 is *less than* (according to an implementation defined rule) `hanp2`.
0	`hanp1` and `hanp2` represent the same object.
> 0	`hanp1` is *greater than* (according to an implementation defined rule) `hanp2`.

The rationale behind multiple return values is so that DM applications can use `dm_handle_cmp()` in sorting.

Name

`dm_handle_hash()` — hashes the contents of a handle

Synopsis

```
u_int
dm_handle_hash(
   void     *hanp,
   size_t    hlen)
```

Description

Hashes the contents of a handle and returns the hashed value.

`void  *hanp (I)`
A pointer to a handle.

`size_t  hlen (I)`
The length of the handle in bytes.

Return Value

Returns a value derived in an implementation defined manner from the contents of the handle. `dm_handle_hash()` has no failure indication.

Name

dm_handle_to_path() — get an absolute pathname

Synopsis

```
int
dm_handle_to_path(
   void     *dirhanp,
   size_t    dirhlen,
   void     *targhanp,
   size_t    targhlen,
   size_t    buflen,
   char     *pathbufp,
   size_t   *rlenp)
```

Description

Takes two object handles, one of which must be the handle of a directory containing the file identified by the other handle. It returns, if possible, an absolute pathname corresponding to the file identified by the target file handle.

Symbolic links may cause the pathname returned to be different from the name used to create the file.

`void *dirhanp (I)`
A handle for the directory containing the target file.

`size_t dirhlen (I)`
The length of the directory handle in bytes.

`void *targhanp (I)`
A file handle for the target file. The target file must be in the directory represented by `dirhanp`.

`size_t targhlen (I)`
The length of the target handle in bytes.

`size_t buflen (I)`
The length of the pathname buffer in bytes.

`char *pathbufp (O)`
The buffer for the pathname.

`size_t *rlenp (O)`
Always updated if the call is successful and undefined if `E2BIG` is returned.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[E2BIG]	The information is too large to fit into the buffer.
[EBADF]	Either `dirhanp` or `targhanp` does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

Name

`dm_init_service()` — perform implementation-defined initialization

Synopsis

```
int
dm_init_service(void)
```

Description

Each process that uses the DMAPI must call dm_init_service() before any other DMAPI function. This allows the implementation of the DMAPI to perform any required initialization.

The results of using any other DMAPI function are undefined if `dm_init_service()` is not called before other DMAPI functions.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[ENOSYS]	The DMAPI is not supported on this platform.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

Name

dm_move_event() — move an event from one session to another

Synopsis

```
int
dm_move_event
   dm_sessid_t     srcsid,
   dm_token_t      token,
   dm_sessid_t     targetsid,
   dm_token_t     *rtokenp)
```

Description

To hand off event servicing to another thread or process without having to copy the message, a DM application can use dm_move_event() to move an outstanding event message between sessions.

The event message remains outstanding, even though it is now enqueued on a different session. Once the call returns successfully, the old token that references the message is no longer valid. The token returned in rtokenp must be used in subsequent calls to reference the message.

dm_sessid_t srcsid (I)
The source session identifier.

dm_token_t token (I)
The token that identifies the message that is to be moved.

dm_sessid_t targetsid (I)
The new session that is to receive the outstanding event.

dm_token_t *rtokenp (O)
The new token that must be used to refer to the message.

Return Value

Zero is returned on success. On error, -1 is returned, and the global errno is set to one of the following values:

[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The source or target sessions are not valid.
[ENOENT]	There is no message corresponding to the token.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

dm_get_events(), dm_respond_event()

Name

`dm_fd_to_handle()` — creates a file handle from a path descriptor
`dm_path_to_fshandle()` — creates a file system handle from a path name
`dm_path_to_handle()` — creates a file handle from a path name
`dm_handle_free()` — frees the storage allocated for a handle

Synopsis

```
int
dm_path_to_handle(
   char      *path,
   void     **hanpp,
   size_t    *hlenp)

int
dm_fd_to_handle(
   int        fd,
   void     **hanpp,
   size_t    *hlenp)

int
dm_path_to_fshandle(
   char      *path,
   void     **hanpp,
   size_t    *hlenp)

void
dm_handle_free(
   void      *hanp,
   size_t     hlen)
```

Description

This set of functions deals with file handles, which are opaque to a DM application. File handles uniquely identify an object and are governed by the rules and restrictions specified in the Requirements Document.

`dm_path_to_handle()` converts a pathname into a file handle. If the final component of the pathname is a symbolic link, the file handle returned is for the symbolic link itself, and not for the object that the symbolic link references (if any).

`char *path (I)`
The pathname of the object.

`void **hanpp (O)`
A pointer that will be initialized by the DMAPI to point to a region of memory containing an opaque DM handle. The caller is responsible for freeing the allocated memory.

`size_t *hlenp (O)`
The length of the handle in bytes.

`dm_fd_to_handle()` converts a file descriptor into a file handle.

`int fd (I)`
A valid, open, file descriptor to the object.

`void **hanpp (O)`
A pointer that will be initialized by the DMAPI to point to a region of memory containing an opaque DM handle. The caller is responsible for freeing the allocated memory.

`size_t *hlenp (O)`
The length of the handle in bytes.

`dm_path_to_fshandle()` returns the file system handle given a pathname to any file in the file system. If the final component of the pathname is a symbolic link, the file handle returned is the file system handle for the file system containing the symbolic link and not the file handle for the file system containing the object that the symbolic link references (if any). The file system handle is used by many DMAPI functions to identify a file system.

`char *path (I)`
A pathname to any file in the file system.

`void **hanpp (O)`
A pointer that will be initialized by the DMAPI to point to a region of memory containing an opaque DM handle that represents the file system meta object. The caller is responsible for freeing the allocated memory.

`size_t *hlenp (O)`
The length of the handle in bytes.

`dm_handle_free()` frees the storage allocated for a handle that was previously returned by `dm_path_to_handle()`, `dm_fd_to_handle()`, or `dm_path_to_fshandle()`.

`void *hanp (I)`
Pointer to the storage area for the file handle that should be freed.

`size_t hlen (I)`
The length of the handle in bytes.

Return Value

`dm_handle_free()` has no failure indication. The other functions return zero is on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[EACCES]	Search permission is denied for a component of `path`.
[EBADF]	`fd` is not a valid open file descriptor.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[ENAMETOOLONG]	A component of `path` or the entire length of `path` exceeded the file system limits.
[ENOENT]	A component of `path` that must exist does not exist.
[ENOMEM]	Insufficient memory to allocate the file handle.
[ENOTDIR]	A component of the specified pathname was not a directory when a directory was expected.
[ENXIO]	The file system containing the file opened on `fd` or the final component of `path` does not support the DMAPI.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

Name

`dm_punch_hole()` — punchs a hole in a file freeing all associated disk blocks
`dm_probe_hole()` — returns the rounded result of the area where the hole is to be punched

Synopsis

```
int
dm_punch_hole(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_off_t        off,
   dm_size_t       len)

int
dm_probe_hole(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_off_t        off,
   dm_size_t       len,
   dm_off_t       *roffp,
   dm_size_t      *rlenp)
```

Description

dm_punch_hole() is used to make a hole in a regular file, logically writing zeroes to the designated area while bypassing normal DMAPI event generation. The DMAPI implementation may then optionally free media resources for that area. `dm_probe_hole()` is used to interrogate the DMAPI implementation for size and offset around the area where the hole is to be punched. The DMAPI implementation can impose boundary and rounding constraints within which the DM application must be willing to work.

The DMAPI guarantees that if the `DM_RIGHT_EXCL` access right is held across a call to `dm_probe_hole()` and then the returned results of the probe call are used to make a `dm_punch_hole()` call, the call will not fail because of rounding.

`dm_punch_hole()` takes the following arguments:

`dm_sessid_t sid (I)`
A session ID.

`void *hanp (I)`
File handle for the regular file that should have its storage manipulated.

`size_t hlen (I)`
The length of the handle in bytes.

`dm_token_t token (I)`
The token referencing the access right for the handle. The access right must be `DM_RIGHT_EXCL` or the token `DM_NO_TOKEN` may be used and the interface will acquire the appropriate rights.

`dm_off_t off (I)`
The starting offset in the file to begin punching a hole. If the offset does not exactly match what the DMAPI implementation can support, `EAGAIN` is returned.

`dm_size_t len (I)`
The length of the hole to punch, in bytes. If the length does not exactly match what the DMAPI implementation can support, `EAGAIN` is returned. If zero is specified, punch hole to the end of file.

`dm_probe_hole()` takes the following arguments:

`dm_sessid_t sid (I)`
A session ID.

`void *hanp (I)`
File handle for the regular file that should have its storage manipulated.

`size_t hlen (I)`
The length of the handle in bytes.

`dm_token_t token (I)`
The token referencing the access right for the handle. The access right must be at least `DM_RIGHT_SHARED` or the token `DM_NO_TOKEN` may be used and the interface will acquire the appropriate rights.

`dm_off_t off (I)`
The desired starting offset in the file for the hole.

`dm_size_t len (I)`
The desired length of the hole to punch, in bytes. A length of zero corresponds to the end of the file.

`dm_off_t *roffp (O)`
The rounded starting offset of the hole that the DMAPI implementation supports.

`dm_size_t *rlenp (O)`
The rounded length of the hole, in bytes, that the DMAPI implementation supports.

Returns

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[E2BIG]	The argument `off` in `dm_probe_hole()` is larger then file size or the sum of `off` and `len` is beyond the end of the file.
[EACCES]	For `dm_probe_hole()`, the access right referenced by the token for the handle is not at least `DM_RIGHT_SHARED`.
[EACCES]	For `dm_punch_hole()`, the access right referenced by the token for the handle is not `DM_RIGHT_EXCL`.
[EAGAIN]	Rounding of the offset and length is required for a punch operation.
[EBADF]	The handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The argument `token` is not a valid token.
[EINVAL]	The file handle does not refer to a regular file.
[EINVAL]	The session is not valid.
[EIO]	I/O error resulted in failure of operation.
[ENOSYS]	The DMAPI implementation does not support this optional function.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

[EROFS] The operation is not allowed on a read-only file system.

See Also

`dm_getalloc_info()`

Notes

Not all file systems support punching holes; see `dm_get_config()`.

Name

dm_query_right() — determine the set of access rights to an object

Synopsis

```
int
dm_query_right(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_right_t     *rightp)
```

Description

This will determine the access rights to the specified object that are referenced by the token.

dm_sessid_t sid (I)
The session associated with the token.

void *hanp (I)
The handle that is being queried for access rights.

size_t hlen (I)
The length of the handle in bytes.

dm_token_t token (I)
The token that references the access rights.

dm_right_t *rightp (O)
The location where the access rights should be returned.

Return Value

Zero is returned on success. On error, -1 is returned, and the global errno is set to one of the following values:

[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The session or token is not valid.
[ENOENT]	The token does not reference any access rights for the handle.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

dm_create_userevent(), dm_request_right(), dm_respond_event()

Name

`dm_query_session()` — query a session for information

Synopsis

```
int
dm_query_session(
   dm_sessid_t     sid,
   size_t          buflen,
   void           *bufp,
   size_t         *rlenp)
```

Description

DM applications can associate up to DM_SESSION_INFO_LEN bytes of information with a session via the dm_create_session() call. To retrieve this information during recovery, a DM application must use dm_query_session().

`dm_sessid_t  sid (I)`
The session ID to be queried.

`size_t  buflen (I)`
The length of the input buffer in bytes.

`void  *bufp (O)`
The buffer to be filled in with the session information.

`size_t  *rlenp (O)`
The size of the requested information. If the buffer is not large enough to hold the requested information (as indicated by the `buflen` parameter), then no data will be copied, `E2BIG` is returned, and `rlenp` is set to the required size. `rlenp` is updated if either the call was successful, or if the call fails with `E2BIG`, but is undefined otherwise.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[E2BIG]	The information is too large to fit into the buffer.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The session is not valid.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_getall_sessions()`

Name

dm_read_invis() — reads a file bypassing DMAPI events
dm_write_invis() — writes to a file bypassing DMAPI events

Synopsis

```
dm_ssize_t
dm_read_invis(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_off_t        off,
   dm_size_t       len,
   void           *bufp)

dm_ssize_t
dm_write_invis(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_off_t        off,
   dm_size_t       len,
   void           *bufp)
```

Description

dm_read_invis() and dm_write_invis() read and write files respectively bypassing normal DMAPI event processing. Also neither function updates a file's access or modified time stamps. The semantics of the call are the same as a POSIX compliant read() or write() call. All writes are synchronous. dm_write_invis() has the same semantics as a standard write followed by an fsync(2) call.

dm_sessid_t sid (I)
A session ID.

void *hanp (I)
File handle for the regular file that will be read to or written from.

size_t hlen (I)
The length of the handle in bytes.

dm_token_t token (I)
The token referencing the access right for the file. The access right must be DM_RIGHT_EXCL on writes, and at least DM_RIGHT_SHARED on reads or the token DM_NO_TOKEN may be used and the interface will acquire the appropriate rights.

dm_off_t off (I)
The starting offset in the file to begin reading or writing.

dm_size_t len (I)
The length to read or write. Note that same POSIX, this is an unsigned value and that it must fit into a dm_ssize_t value as a positive number.

void *bufp (I/O)
When reading, this is an output parameter, and will be filled in with the data from the file. When writing, it is an input parameter, and contains the data that will be written to the file.

Return Value

Number of bytes read or written on success, -1 on failure and global `errno` set to one of the following or other implementation specific values:

[EACCES]	For `dm_read_invis()`, the access right referenced by the token for the handle is not at least `DM_RIGHT_SHARED`.
[EACCES]	For `dm_write_invis()`, the access right referenced by the token for the handle is not `DM_RIGHT_EXCL`.
[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EFBIG]	The sum of `off` and `len` is larger than maximum file size for write operation.
[EINTR]	The operation was interrupted by a signal and should be re-tried. On `dm_write_invis()`, no data was written to the file.
[EINVAL]	The argument `len` is too large and will overflow if placed into a `dm_ssize_t` value.
[EINVAL]	The argument `off` is larger than the file size for read.
[EINVAL]	The argument `token` is not a valid token.
[EINVAL]	The file handle does not refer to a regular file.
[EINVAL]	The session is not valid.
[EIO]	I/O error resulted in failure of operation.
[ENOSPC]	No space left in the file system.
[EPERM]	The caller does not hold the appropriate privilege (root or other file system permissions).
[EROFS]	The operation is not allowed on a read-only file system.

Name

dm_release_right() — release all access rights to an object

Synopsis

```
int
dm_release_right(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token)
```

Description

The access rights referenced by the token for the specified object are released. If the access rights can not be immediately relinquished, EAGAIN is returned.

dm_sessid_t sid (I)
The session associated with the token.

void *hanp (I)
The handle for the file system object for which the access rights are being released.

size_t hlen (I)
The length of the handle in bytes.

dm_token_t token (I)
The token referencing the access rights for the specified object.

Return Value

Zero is returned on success. On error, -1 is returned, and the global errno is set to one of the following values:

[EACCES]	The token does not have rights to the object.
[EAGAIN]	The right could not be immediately released.
[EBADF]	The file handle does not refer to an existing object.
[EBUSY]	The DMAPI implementation does not support releasing this type of right.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The session is not valid.
[EINVAL]	The token is not valid.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

dm_request_right(), dm_query_right(), dm_create_userevent()

Name

dm_remove_dmattr() — remove a data management attribute

Synopsis

```
int
dm_remove_dmattr(
    dm_sessid_t     sid,
    void           *hanp,
    size_t          hlen,
    dm_token_t      token,
    int             setdmtime,
    dm_attrname_t  *attrnamep)
```

Description

The dm_remove_dmattr() function removes the attribute specified by attrname. When a file is removed, the attributes associated with the file will be removed automatically.

dm_sessid_t sid (I)
: A session ID.

void *hanp (I)
: The handle for the file for which the attributes should be retrieved.

size_t hlen (I)
: The length of the handle in bytes.

dm_token_t token (I)
: The token referencing the access right for the handle. The access right must be DM_RIGHT_EXCL or the token DM_NO_TOKEN may be used and the interface will acquire the appropriate rights.

int setdmtime (I)
: The file's attribute time stamp will be updated if setdmtime is non-zero.

dm_attrname_t *attrnamep (I)
: The attribute to be removed.

Return Value

Zero is returned on success. On error, -1 is returned, and the global errno is set to one of the following values:

[EACCES]	The access right referenced by the token for the handle is not DM_RIGHT_EXCL.
[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The argument token is not a valid token.
[EINVAL]	The session is not valid.
[EIO]	I/O error resulted in failure of operation.
[ENOSYS]	The DMAPI implementation does not support this optional function.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).
[EROFS]	The operation is not allowed on a read-only file system.

See Also

dm_set_dmattr()

4687 **Notes**

4688 `dm_remove_dmattr()` is an optional component of the DMAPI.

Name

`dm_request_right()` — request a specific access right to an object

Synopsis

```
int
dm_request_right(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   int             nowait,
   dm_right_t      right)
```

Description

Add the requested right to the access rights referenced by the token for the object specified by the handle. This is the mechanism by which DM applications can gain specific access rights to an object.

There are two access rights that can be requested; `DM_RIGHT_SHARED` and `DM_RIGHT_EXCL`. `dm_request_right()` will block until the requested right can be obtained. If a DM application needs to determine if the right can be obtained without blocking, then the `nowait` flag should be set. If `nowait` is set and the requested right can not immediately be granted, then `EAGAIN` is returned.

If a process holds a shared access right (`DM_RIGHT_SHARED`) to an object and requests an exclusive right (`DM_RIGHT_EXCL`) in a blocking mode (the `nowait` flag is clear), then request will fail immediately with an `EINVAL`. The shared lock will not be released.

If a process attempts to upgrade from a `DM_RIGHT_SHARED` to an `DM_RIGHT_EXCL` lock in a non-blocking mode (the `nowait` flag is set), then the request may work. If the request fails, `EBUSY` is returned, but the shared lock is not released.

When requesting access rights to an object via `dm_request_right()`, the requested right may not be immediately available. If the DM application has specified that it wants to block until the right becomes available, the DM application may or may not be blocked interruptibly. The implementation of the DMAPI will specify the semantic that is provided for interrupting blocked processes. When interrupts are allowed and occur, the `EINTR` error will be returned

`dm_sessid_t  sid (I)`
: The session associated with the token.

`void  *hanp (I)`
: The handle for the file system object for which access rights are being requested.

`size_t  hlen (I)`
: The length of the handle in bytes.

`dm_token_t  token (I)`
: The token that references the access rights for the object to be manipulated.

`int  nowait (I)`
: If set to a non-zero value, the calling process will return with `EAGAIN` if the requested right is not immediately available. If clear (zero), the calling process will block, waiting for the requested right be become available.

`dm_right_t right (I)`
The requested access right.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[EACCES]	The process attempted to upgrade a lock in blocking mode.
[EAGAIN]	The `nowait` flag is set and the process would be delayed in requesting the right.
[EBADF]	The file handle does not refer to an existing object.
[EBUSY]	The upgrade from `DM_RIGHT_SHARED` to `DM_RIGHT_EXCL` cannot be granted.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINTR]	The DMAPI implementation allows interruption and the process was interrupted.
[EINVAL]	The requested right is not valid.
[EINVAL]	The session or token is not valid.
[ENFILE]	Too many rights currently granted in the system.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_create_userevent()`, `dm_respond_event()`, `dm_release_right()`, `dm_get_fileattr()`, `dm_query_right()`

Name

`dm_respond_event()` — respond to an event

Synopsis

```
int
dm_respond_event(
   dm_sessid_t      sid,
   dm_token_t       token,
   dm_response_t    response,
   int              errno,
   size_t           buflen,
   void            *respbufp)
```

Description

Once a synchronous message has been received via `dm_get_events()`, it must be responded to. Otherwise, the application that caused the event to be generated will be blocked, system resources will be tied up, and so forth.

The token identifies the message. Once this call has returned, the token is no longer valid.

Some DMAPI implementations may provide extensions to the basic event facility, such as the ability to return attribute information when responding to an event. These extensions will utilize the `respbufp` parameter. Normal event responses will only make use of the response code and corresponding `errno` value.

`dm_sessid_t  sid (I)`
: The session for the token.

`dm_token_t  token (I)`
: The token identifying the message.

`dm_response_t  response (I)`
: The action to be taken by the operating system. Valid actions are `DM_RESP_CONTINUE`, which continues the operation, or `DM_RESP_ABORT`, which aborts the operation and returns the error `errno` (if supplied) to the user process that caused the event. `DM_RESP_DONTCARE` is also valid when responding to the *`mount`* event.

`int  errno (I)`
: If the operation is to be aborted (`response` is set to `DM_RESP_ABORT`), return `errno` to the application that caused the event. `errno` will be returned to the user, even if it is set to zero.

`size_t  buflen (I)`
: The length of the response buffer in bytes. Only used by implementations that provide extensions to the DMAPI.

`void  *respbufp (I)`
: Buffer for DMAPI implementation extensions. The format of this buffer is not defined by the DMAPI.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[E2BIG]	`buflen` is larger than the implementation defined limit. The limit can be determined by calling the `dm_get_config()` function using `DM_CONFIG_MAX_ATTRIBUTE_SIZE`.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The session is not valid.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).
[ESRCH]	The token does not reference a valid message.

See Also

`dm_get_events()`

Name

`dm_send_msg()` — send a message to the indicated session

Synopsis

```
int
dm_send_msg(
   dm_sessid_t     targetsid,
   dm_msgtype_t    msgtype,
   size_t          buflen,
   void           *bufp)
```

Description

For testing purposes, it is often desirable to send a message to a session. `dm_send_msg()` allows an event message of type *`user`* to be enqueued on the indicated session. The contents of the message data are opaque to the DMAPI. The message will remain enqueued until the indicated target session executes `dm_get_events()`.

`dm_send_msg()` can send both synchronous and asynchronous messages. If the type is `DM_MSGTYPE_SYNC`, then a synchronous message is created, and the calling process blocks until a response is returned. The DMAPI will generate a token that references no access rights. If the type is `DM_MSGTYPE_ASYNC`, then an asynchronous event message is created, and the calling process is not blocked. The DMAPI will not generate a token for asynchronous messages. The lack of a valid token is identified to the receiver of the event by a value of `DM_INVALID_TOKEN` in the token field of the event message.

`dm_sessid_t  targetsid (I)`
: The session the message should be enqueued upon.

`dm_msgtype_t  msgtype (I)`
: The type of message to be enqueued: `DM_MSGTYPE_SYNC` or `DM_MSGTYPE_ASYNC`.

`size_t  buflen (I)`
: The length of the data buffer in bytes.

`void  *bufp (I)`
: The data that is to be incorporated into the generated message. If `buflen` is zero, this parameter is ignored.

Return Value

Upon successful completion, the function `dm_send_msg()` will return zero for asynchronous messages; the return value for synchronous messages is dependent on the return code from the event servicing. Otherwise a value of -1 is returned and `errno` is set to one of the following values:

[E2BIG]	The user message data is too big to fit into implementation defined limit. The limit can be determined by calling the `dm_get_config()` function using `DM_CONFIG_MAX_MESSAGE_DATA`.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINTR]	Interrupted. The event was not delivered.
[EINVAL]	The session is not valid.
[EINVAL]	The type is not `DM_MSGTYPE_SYNC` or `DM_MSGTYPE_ASYNC`.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

4879 **See Also**

4880 `dm_get_events()`, `dm_create_userevent()`

Name

`dm_set_disp()` — set the disposition of events on a file system for a session

Synopsis

```
int
dm_set_disp(
   dm_sessid_t      sid,
   void            *hanp,
   size_t           hlen,
   dm_token_t       token,
   dm_eventset_t   *eventsetp,
   u_int            maxevent)
```

Description

After establishing a session, a DM application must indicate which events to deliver to the session. The list is the complete set of all events, including managed region events, that the DM application will be monitoring during the life of this session. This event list is not persistent across reboots.

The main purpose of this call is to establish an (*event, file system, session*) binding that lets the DMAPI know which session to send events to. Until `dm_set_disp()` has completed, the DMAPI has no way of knowing what session should receive events that are generated on a file system. Once `dm_set_disp()` has returned to the calling DM application, the session will start receiving events as they are generated by the file system.

If a specific event in `eventlist` already has been established for the file system indicated by `hanp`, then the prior (*event, file system, session*) binding is replaced by the one specified by the `eventset` parameter. It is not possible to target the same event to multiple sessions simultaneously.

DM applications can only register the event list and session defaults using the file system handle; it is not possible to register on arbitrary file system objects (such as regular files). Since the file system handle can only be obtained after the file system has been mounted, it is not possible to specify the *`mount`* event using the file system handle. The global handle must be used. If the *`mount`* event is specified using the file system handle, it will be undefined. If events other than the *`mount`* event are specified for the global handle, they will also be undefined.

`dm_sessid_t  sid (I)`
: The session to deliver events to.

`void  *hanp (I)`
: The handle for the file system or the global handle.

`size_t  hlen (I)`
: The length of the handle in bytes.

`dm_token_t  token (I)`
: The token referencing the access right for the handle. The access right must be `DM_RIGHT_EXCL` or the token `DM_NO_TOKEN` may be used and the interface will acquire the appropriate rights.

`dm_eventset_t  *eventsetp (I)`
: The list of default events to be monitored for this session.

`u_int maxevent (I)`
The number of events to be checked for dispositions in the event set. The events from `0` to `maxevent - 1` are examined.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[EACCES]	The access right referenced by the token for the handle is not `DM_RIGHT_EXCL`.
[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The handle is not a file system handle.
[EINVAL]	The session is not valid.
[EINVAL]	The token is not valid.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[ENXIO]	The implementation of the DMAPI does not support registering for events on the specified handle.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_create_session(), dm_set_eventlist(), dm_getall_disp()`

Name

dm_set_dmattr() — create or replace a data management attribute

Synopsis

```
int
dm_set_dmattr(
    dm_sessid_t      sid,
    void            *hanp,
    size_t           hlen,
    dm_token_t       token,
    dm_attrname_t   *attrnamep,
    int              setdmtime,
    size_t           buflen,
    void            *bufp)
```

Description

dm_set_dmattr() creates or replaces the named attribute with the specified data. If replacing an attribute either the original contents or the new contents will be retained, not a mixture of both.

`dm_sessid_t sid (I)`
A session ID.

`void *hanp (I)`
The handle for the file for which the attributes should be created or replaced.

`size_t hlen (I)`
The length of the handle in bytes.

`dm_token_t token (I)`
The token referencing the access right for the handle. The access right must be `DM_RIGHT_EXCL` or the token `DM_NO_TOKEN` may be used and the interface will acquire the appropriate rights.

`dm_attrname_t *attrnamep (I)`
The attribute to be created or replaced.

`int setdmtime (I)`
The file's attribute time stamp will be updated if `setdmtime` is non-zero.

`size_t buflen (I)`
The size of the buffer in bytes.

`void *bufp (I)`
The buffer containing the attribute data.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[E2BIG]	`buflen` is larger than the implementation defined limit. The limit can be determined by calling the `dm_get_config()` function using `DM_CONFIG_MAX_ATTRIBUTE_SIZE`.
[EACCES]	The access right referenced by the token for the handle is not `DM_RIGHT_EXCL`.
[EBADF]	The file handle does not refer to an existing object.

[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EIO]	An attempt to write the new or updated attribute resulted in an I/O error.
[EINVAL]	The argument `token` is not a valid token.
[EINVAL]	The session is not valid.
[ENOMEM]	The DMAPI could not acquire the required resources to complete the call.
[ENOSPC]	An attempt to write the new or updated attribute resulted in an error due to no free space being available on the device.
[ENOSYS]	The DMAPI implementation does not support this optional function.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).
[EROFS]	The operation is not allowed on a read-only file system.

See Also

`dm_get_dmattr()`, `dm_remove_dmattr()`, `dm_getall_dmattr()`

Notes

`dm_set_dmattr()` is an optional component of the DMAPI.

Name

`dm_set_eventlist()` — set the list of events to be enabled for an object

Synopsis

```
int
dm_set_eventlist(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   dm_eventset_t  *eventsetp,
   u_int           maxevent)
```

Description

To receive events on an object, a DM application must bind a set of events to an object, indicating to the DMAPI that the application is monitoring the object. `dm_set_eventlist()` will establish a binding of events on the indicated object that may be persistent. Persistence of event bindings is implementation defined, and can be determined via the `dm_get_config()` call.

The two functions `dm_set_disp()` and `dm_set_eventlist()` work together to enable DM applications to receive events. `dm_set_eventlist()` controls the *generation* of events on an object, by specifying which file system operations should generate a DMAPI event. `dm_set_disp()` controls the *delivery* of events by specifying which session should receive the generated event.

Events can be enabled on any file system object, including regular files. The managed region events can not be specified via `dm_set_eventlist()` since they are specific to each managed region. The *`mount`* event can also not be specified, since it is generated by all file system types that support the DMAPI.

`dm_sessid_t  sid (I)`
: A session id.

`void  *hanp (I)`
: The handle for the object. The handle can be either the file system handle, or a file handle.

`size_t  hlen (I)`
: The length of the handle in bytes.

`dm_token_t  token (I)`
: The token referencing the access right for the handle. The access right must be `DM_RIGHT_EXCL` or the token `DM_NO_TOKEN` may be used and the interface will acquire the appropriate rights.

`dm_eventset_t  *eventsetp (I)`
: The list of events to be enabled for the object.

`u_int  maxevent (I)`
: The number of events to be checked for dispositions in the event set. The events from `0` to `maxevent - 1` are examined.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[EACCES]	The access right referenced by the token for the handle is not `DM_RIGHT_EXCL`.
[EBADF]	The handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The argument `token` is not a valid token.
[EINVAL]	The session is not valid.
[EINVAL]	Tried to set event on a global handle.
[ENOMEM]	The DMAPI could not obtain the required resources to complete the call.
[ENXIO]	The implementation of the DMAPI does not support enabling event delivery on the specified handle.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).
[EROFS]	The operation is not allowed on a read-only file system.

See Also

`dm_get_eventlist(), dm_set_disp(), dm_set_region(), DMEV_SET()`

Name

`dm_set_fileattr()` — set file timestamps, ownership and mode

Synopsis

```
int
dm_set_fileattr(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           mask,
   dm_fileattr_t  *attrp)
```

Description

The function dm_set_fileattr() allows DM applications to set various file attributes as defined by the dm_fileattr structure.

Any valid field (as determined by the mask argument) will be used to set the corresponding attribute in the file. For fields that are not specified to be set, then the current value will not be modified. For example, if `DM_AT_CTIME` is not set, then the file's `ctime` field will not be modified.

If the call to `dm_set_fileattr()` requests updates of both `dm_ctime` and `dm_dtime`, and the DMAPI implementation overloads `dm_ctime` and `dm_dtime`, then `dm_ctime` will be used.

`dm_sessid_t  sid (I)`
A session ID.

`void  *hanp (I)`
The handle for the file for which the file attributes should be set.

`size_t  hlen (I)`
The length of the handle in bytes.

`dm_token_t  token (I)`
The token referencing the access right for the handle. The access right must be `DM_RIGHT_EXCL` or the token `DM_NO_TOKEN` may be used and the interface will acquire the appropriate rights.

`u_int  mask (I)`
The argument mask indicates which fields of the `dm_fileattr` structure to set. The mask is constructed by OR-ing together one or more of the following flags:

`DM_AT_ATIME`	The file's access timestamp.
`DM_AT_MTIME`	The file's data modification timestamp.
`DM_AT_CTIME`	The file's status change timestamp.
`DM_AT_DTIMES`	The time stamp of the persistent attributes if associated with the file. The value is ignored if no DM attributes exist.

DM_AT_UID	User ID of the file's owner.
DM_AT_GID	Group ID of the file's owner.
DM_AT_MODE	The mode of the file as described by the mknod(2) system call.

dm_fileattr_t *attrp (I)

The dm_fileattr structure with the appropriate fields defined. For fields which won't be set, the corresponding fields of the dm_fileattr structure can be undefined.

Return Value

Zero is returned on success. On error, -1 is returned, and the global errno is set to one of the following values:

[EACCES]	The access right referenced by the token for the handle is not DM_RIGHT_EXCL.
[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The argument mask is not valid.
[EINVAL]	The argument token is not a valid token.
[EINVAL]	The session is not valid.
[EIO]	I/O error resulted in failure of operation.
[ENOMEM]	The DMAPI could not acquire the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).
[EROFS]	The operation is not allowed on a read-only file system.

See also

dm_fileattr_t

Name

`dm_set_inherit()` — mark an attribute inheritable

Synopsis

```
int
dm_set_inherit(
    dm_sessid_t      sid,
    void            *hanp,
    size_t           hlen,
    dm_token_t       token,
    dm_attrname_t   *attrnamep,
    mode_t           mode)
```

Description

The function `dm_set_inherit()` marks the attribute specified by `attrnamep` as an inheritable attribute for the file system specified by handle. An attribute is inherited from the parent directory during file creation. The inheritable property of attributes is not persistent across reboots. The access right referenced by the token for the object must be `DM_RIGHT_EXCL`

`dm_sessid_t  sid (I)`
A session ID.

`void  *hanp (I)`
The file system handle for which the inheritable attributes should be set.

`size_t  hlen (I)`
The length of the handle in bytes.

`dm_token_t  token (I)`
The token referencing the access right for the handle. The access right must be `DM_RIGHT_EXCL` or the token `DM_NO_TOKEN` may be used and the interface will acquire the appropriate rights.

`dm_attrname_t  *attrnamep (I)`
The attribute to be marked as inheritable.

`mode_t  mode (I)`
The argument `mode` limits the scope of inheritance to file objects created of specified values. `mode` is taken from Spec 1170 values. For example if `mode` indicates a regular file and then a directory is created, an event will not be generated and the attribute will not be copied to the new directory.

Return Value

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[EACCES]	The access right referenced by the token for the handle is not `DM_RIGHT_EXCL`.
[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The argument `fs_hanp` does not refer to a file system handle.
[EINVAL]	The argument `mode` does not contain valid file types.
[EINVAL]	The session or token is not valid.
[ENOMEM]	The DMAPI could not acquire the required resources to complete the call.

[ENOSYS]	The DMAPI implementation does not support this optional function.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).

See Also

`dm_clear_inherit()`, `dm_getall_inherit()`

Notes

`dm_set_inherit()` is an optional component of the DMAPI.

Name

`dm_set_region()` — set the managed regions for a file

Synopsis

```
int
dm_set_region(
   dm_sessid_t     sid,
   void           *hanp,
   size_t          hlen,
   dm_token_t      token,
   u_int           nelem,
   dm_region_t    *regbufp,
   u_int          *exactflagp)
```

Description

`dm_set_region()` replaces the set of managed regions for a regular file. The implementation has the right to round and or change the number of managed regions. The final set of managed regions must cover the requested regions. Overlapping managed regions are not permitted.

The `dm_region` structure defines the range of bytes that are managed in the file. The `rg_flags` field can be set to generate synchronous read, write, and truncate events whenever the associated operation is performed within the managed region. A region may extend outside the current valid portions of a file. A region may extend to the end of a file and allow control on files that are appended to by setting `rg_size` to zero. Regions may also be allocated that have no flags set. This may be used to allocate space for future control of a region of the file in DMAPI implementations that provide persistent managed regions.

The implementation is free to reorder the managed region set. A subsequent call to `dm_get_region()` may return the managed regions in a different order than they were provided to `dm_set_region()`. The DMAPI implementation is also permitted to coalesce contiguous or noncontiguous regions with identical `rg_flags`. The `exactflagp` is set to a non-zero value if the file system did not reorder the managed region set.

For operations involving multiple managed regions within a file, only one region event will be generated for the operation. It is up to the DM application to respond appropriately to all managed regions that are effected by an operation.

`dm_sessid_t  sid (I)`
: A session ID.

`void  *hanp (I)`
: Handle for the regular file to be affected.

`size_t  hlen (I)`
: The length of the handle in bytes.

`dm_token_t  token (I)`
: The token referencing the access right for the handle. The access right must be `DM_RIGHT_EXCL` or the token `DM_NO_TOKEN` may be used and the interface will acquire the appropriate rights.

`u_int  nelem (I)`
: The number of input regions in `regbufp`.

`dm_region_t *regbufp (I)`
A pointer to the structure defining the regions to be set.

`u_int *exactflagp (O)`
If set to a non-zero value, the file system did not reorder the requested managed region set.

Valid values for the `rg_flags` field of the region structure are created by OR'ing together one or more of the following values:

`DM_REGION_READ`	Enable synchronous event for read operations that overlap this managed region.
`DM_REGION_WRITE`	Enable synchronous event for write operations that overlap this managed region.
`DM_REGION_TRUNCATE`	Enable synchronous event for truncate operations that overlap this managed region.
`DM_REGION_NOEVENT`	Do not generate any events for this managed region.

Return Values

Zero is returned on success. On error, -1 is returned, and the global `errno` is set to one of the following values:

[E2BIG]	The number of regions specified by `nelem` is insufficient for the rounding required by the DMAPI.
[EACCES]	The access right referenced by the token for the handle is not `DM_RIGHT_EXCL`.
[EBADF]	The file handle does not refer to an existing object.
[EFAULT]	The system detected an invalid address in attempting to use an argument.
[EINVAL]	The argument `token` is not a valid token.
[EINVAL]	The file handle does not refer to a regular file.
[EINVAL]	The regions passed in are not valid because they overlap or some other problem.
[EINVAL]	The session is not valid.
[EIO]	An I/O error resulted in failure of operation.
[ENOMEM]	The DMAPI could not acquire the required resources to complete the call.
[EPERM]	The caller does not hold the appropriate privilege (root or file system permissions).
[EROFS]	The operation is not allowed on a read-only file system.

See Also

`dm_get_region()`

7.0 Implementation Notes

This section describes some issues and hints that may be of use to the implementor of the DMAPI. These hints are given in this section because they are relevant to the DMAPI, yet they may not be applicable to all DMAPI implementations.

7.1 `select`(2) and Sessions

As discussed in the *Session Management* section, session IDs should not be interpreted as file descriptors. Some DM applications, however, may desire the ability to `select(2)` on a session in order to determine if a message is ready to be received. This can be very useful if one process is the dispatcher for both DMAPI messages and network messages.

This can be implemented with a pseudo-driver. The driver would be responsible for implementing the `select(2)` callout, and would have to scan the DMAPI message queue to see if messages are available. Obviously, there must be a high level of communication and understanding between the driver and the implementation of the DMAPI.

7.2 Event Encodings

As discussed in the *Non-Opaque Data Management Attributes* section, event bit masks may exist that allow specific encodings of events to be represented with a number of fixed bit patterns. These bit patterns are implementation defined, are not guaranteed to be the same from platform to platform, and in fact are not visible through any of the DMAPI interfaces.

A single bit might allow the following event list encodings:

- `0` The file has no event lists.
- `1` The file has an event list but there is not enough persistent storage available to encode the events.

By increasing the number of bits, well known combinations of events can be stored persistently with the file. For example, with 2 bits the following event list encodings may exist:

- `00` The file has no event lists.
- `01` Read events will be generated for this file.
- `10` Write events will be generated for this file.
- `11` The file has an event list but there is not enough persistent storage available to encode the events.

For example, if the file has both read and write events set, then this cannot be encoded in two bits. In this case the *`debut`* event would need to be generated in order to load the event list.

The encoding of events is not defined by the DMAPI. It is expected that different DMAPI implementations would encode events differently, depending on those data management applications which would be supported.

It must be noted that due to the requirement regarding implementation of the DMAPI, support for persistent non-opaque attributes is a DMAPI implementation option.

7.3 Event Ordering

Each implementation of the DMAPI may send a number of different events to a DM application as a result of a single system call. It may be helpful but is not required for DM application writers to know in advance the sequence of events for each system call. Unfortunately, determining this list is a non-trivial exercise. It is suggested that the DM application writer work with the implementor of the DMAPI in order to determine the sequence and ordering of events that occurs for system calls of interest.

If the application is multi-threaded, then the DM app writer's job is even more difficult. For the paired namespace events (such as *create* and *postcreate*), special care should be taken. Synchronization primitives are necessary so that one process is not left hanging waiting for an event that has been serviced by another process.

7.4 Lock Releasing

Some DMAPI implementations may have special lock restrictions. Some may be unable to upgrade an access right from DM_RIGHT_SHARED to DM_RIGHT_EXCL without sleeping, while others may have special primitives that allow them to grant whatever right the DM app requires. Some DMAPI implementations may also have special requirements with regard to releasing locks. During the servicing of events, it may not always be possible to relinquish a right. DM app writers should not assume that it is always possible to release a right (via dm_release_right()). The careful DM application will always check return codes from **all** DMAPI routines.

7.5 Tokens, Messages, and Handles

Tokens reference access rights for handles, and must always be associated with a message. Some implementations may wish to view a token a simply a message ID. When viewed this way, the token is similar to a file descriptor, in that it just references state maintained in the kernel. Given the one-to-one correspondence of tokens and messages, one can think of them (within the kernel) as separate objects or one object independent of how they are actually implemented.

7.6 mmap

There are many operating systems that use mmap(2) as a mechanism for utilities to copy file data. Unfortunately, this can make implementation of the DMAPI difficult since a page fault often occurs at a much later point in time than when the file is actually mapped. Usually, there are also very stringent locking restrictions in place at the time the page fault occurs.

In order to get around these problems, some systems may have to adopt a paradigm that at the time the file is mapped (at the time the actual mmap(2) call is executed), the file will have to be made resident if it is non-resident. This is sub-optimal from a disk space utilization standpoint, and it also means that a large file that is mmap'd can not be partially resident. However, the option is to place DMAPI callouts deep in the bowels of the VM subsystem which is usually a very painful exercise and is not recommended for the faint of heart.

7.7 Invisible I/O

Invisible read and write can also place special burdens on the implementor of the DMAPI. Invisible I/O is typically used by a DM application to reload data for a non-resident file. This operation should not modify any of a file's timestamps, nor should it cause events to be generated.

In systems where pages are encached on the vnode, this can lead to painful locking and coherency issues. How is the actual write to disk performed? Is the page cache bypassed? How do you ensure that encached pages are dealt with correctly?

7.8 Generation of Events

The placement of code that actually generates events will be different from platform to platform. *When* an event is actually generated will be different on all systems, and should not concern the DM application writer. The DMAPI implementor will need to take into consideration lock state, likelihood of the operation succeeding, and so forth to determine the best location for the actual callout to occur.

In the case of managed region events, the DMAPI implementor must ensure that a poorly-behaved DM application does not cause the system to behave in an unexpected manner. For example, if a file has multiple managed regions that represent non-resident data, and a DM application only restores the data for one of those regions, the system must be sure that operations such as read-ahead do not cause multiple events to occur. See the *Managed Region* section for more information.

7.9 Locking Across Operations

To ensure consistency, some DM application may wish to enforce their own locking scheme across operations. They may want to develop a wrapper around some operations in order to synchronize and/or serialize accesses to files.

DM applications must also implement special locking to ensure consistency if they are intercepting the *midcreate* event. When the *midcreate* event is received, there is still no guarantee that the operation that caused the event to be generated with succeed. The DM application will most likely need to intercept the *postcreate* event to be certain of the status of the operation.

7.10 Tokens and Multiple Handles

As discussed in the *Token Management* section, tokens may reference access rights for more than one file handle. This makes certain operations easier, such as obtaining access rights to a list of file handles; the same token can be reused without incurring the overhead of `dm_create_userevent()` to craft up a new token for each file handle. However, allowing a single token to reference multiple handles can make recovery more difficult. How does a DM application determine which file handles have access rights referenced by a single token?

During recovery, a DM application can always execute `dm_respond_event()`, give it the offending token, and let the DMAPI release any and all access rights associated with the token. If the DM application needs to selective about which file handles have their access rights released, then it (the DM application) must provide some mechanism external to the DMAPI to log which file handles are associated with a token. The DMAPI will not provide interfaces to identify multiple file handles from a single token.

7.11 Structure Lists

A number of DMAPI routines returns lists of structures. Some of these routines return lists of variable-length structures. Since the length of the structure is not known, DMAPI implementations must provide a mechanism for the DM app to access the various members of the list.

The DMAPI specifies that DM applications should use the DM_STEP_TO_NEXT() macro to access variable length structures that are in a list. However, the actual implementation of this macro

is not defined. One possibility is that each variable length structure have a field in a well-known position (say offset zero) or use a special field name that is opaque to the DM application. For example, using the field approach, the definition of the dm_eventmsg_t would become:

```
struct dm_eventmsg {
   ssize_t        _link;
   dm_type_t      ev_type;
   dm_token_t     ev_token;
   dm_vardata_t   ev_data;
};
```

The definition of the DM_STEP_TO_NEXT macro would then become:

```
#define DM_STEP_TO_NEXT(p, type) \
     ((type)((p)->_link ? (char *)(p) + (p)->_link : NULL))
```

7.12 Undeliverable Event Messages

The implementation of the DMAPI will need to define the guidelines for delivery of a synchronous event message when no session exists to receive it. There are only two choices with regard to synchronous event message delivery when no session exists:

- block the requesting process.
- return an error to the process that instigated the event.

If an error is returned to the process, it may be specific to the operation that caused the event. This means that depending on the operation, two different errors can be returned for the same event type.

The implementation must also define the behavior for asynchronous events as well.

8.0 Using the Data Management Interfaces

In order to give some insight as to how the DMAPI could be used, the following examples will be given:

- *Mr. Mean* — a recovery and cleanup application.
- A simple application that monitors the mount events for a particular file system.
- A primitive dump application
- A simple migration application, showing stagein and stageout
- A multi-threaded application

A word about the examples. They are intended to show features of the DMAPI and how to use them. They are not necessarily examples of shining C coding practices. In particular, most functions are not checked for an error return; adding error checking code made the examples unnecessarily cluttered and hard to read. Structures that are `malloc'd` are frequently not released. There are few subroutines; everything is in line. Error messages are not very descriptive, and in many cases, not helpful at all. Also, the examples sometimes go to great lengths to do things in a convoluted manner in order to illustrate a property of the DMAPI. There is no guarantee that these examples will even compile on your system.

8.1 Recovery and Cleanup

The following is an example of a cleanup application. This application (`mr_mean`) finds the session for a (presumably) dead DM application, assumes the session, and then releases all the tokens. Mr_Mean could also be modified to perform more useful (and less hostile) application recovery.

The following concepts are illustrated:

- Session assumption
- Handling of variable length data structures
- Creating tokens and obtaining rights
- Setting event disposition
- Finding outstanding tokens
- Resizing a buffer on `E2BIG` errors

There are a couple of routines that are not shown:

`err_msg()`	generic error reporting routine
`resize_buf()`	increase the size of a buffer, probably by calling `realloc()` or something similar

```
char              mean_msg = "Mr Mean's token";

size_t            nsids, nret_sids;
size_t            ses_retlen;
size_t            tokenbuf_len, tok_retlen, start;
size_t            ndisp;
size_t            fs_hlen;
int               more_sessions, more_disp, found;
int               msglen;
void             *fs_hanp;
dm_sessid_t      *sidbuf, sid;
dm_dispinfo_t    *disp_buf, *disp;
dm_token_t        fs_token, *token_buf, *tokenp
dm_event_set_t    zero_eventset;
char              nameinfo_buf[DM_SESSION_INFO_LEN];


if (dm_init_service() == -1 ) {
     err_msg("Can't initialize DMI\n");
     return(0);
}

/*
 * Get some space for a buffer that will be used to store all
 * active sessions. We choose an arbitrary number of 64 as a guess
 * on the number of extant sessions
 */
nsids  = 64;
sidbuf = (dm_sessid_t *)malloc(nsids * sizeof(dm_sessid_t));
do {
     more_sessions = dm_getall_sessions(nsids, &sidbuf, &nret_sids);
     if (more_sessions == -1 ) {
          err_msg("Can't get sessions\n");
          do_cleanup();
          return(1);
     }

     found = 0;
     sidp  = sidbuf;
     for (i=0; i< nret_sids; i++ ) {
          /*
           * Now we have all the active sessions in the system. Query each
           * session looking for our target session info. For this example,
           * we're looking for a DM app that opened a session under the name
           * "MultiForest".
           */
          dm_query_session(*sidp, DM_SESSION_INFO_LEN, nameinfo_buf,
                           &ses_retlen);
          if ((strncmp("MultiForest", nameinfo_buf, ses_retlen)) == 0) {
               found = 1;
               break;
          }
```

```
                sidp++;
        }
    } while (more_sessions == 1 & !found);

    if (!found) {
        err_msg("Can't find target session\n");
        do_cleanup();
        return(1);
    }
```

We have now come to a point where we found the session we want to assume. We can now assume the session. At this point, we would be acting just as the previous instantiation of the session. The DMAPI makes no distinction between us and ole' dead MultiForest. In case the session might still be active, once we've assumed it we set the event disposition to zero by saying this session is not interested in any event. This effectively halts generation of any more events. The DMAPI will follow some implementation-defined rules from that point on with regard to how event generation is handled.

```
    /*
     * Now that we've found the session we want to assume, we'll
     * try to assume it
     */
    dm_create_session(*sidp, &sid, "mr_mean");


    /*
     * Since we assumed the old session, we are now acting just like
     * the old DM application. Since we're trying to clean up after an
     * aborted session (we assume "MultiForest" crashed, but we could
     * just be taking over its session for other reasons as well), the first
     * step is to disable event generation so that we can drain the session.
     *
     * We arbitrarily obtain information on 4 file systems at a time
     */
    ndisp = 4;
    disp_buf = (dm_dispinfo_t *)malloc(ndisp * sizeof(dm_dispinfo_t));

    DMEV_ZERO(zero_eventset);

    more_disp = dm_getall_disp(sid, , &disp_buf, &ret_ndisp);
    while (more_disp != -1) {
        if (more_disp == 0)
            break;

        /*
         * Walk through the list of file system handles we just obtained
         * and turn off event generation for each file system
         */
        disp = DM_STEP_TO_NEXT(disp, dm_dispinfo_t *);
        while (disp != NULL) {
            fs_hanp = DM_GET_VALUE(disp, fs_handle, void *);
            fs_hlen = DM_GET_LEN(disp, fs_handle, size_t)

            /*
             * Obtain a token for the handle, then get exclusive
             * access to it
             */
            msglen = strlen(mean_msg) + 1;
            dm_create_userevent(sid, msglen, mean_msg, &fs_token);
            dm_request_right(sid, fs_hanp, fs_hlen, fs_token,DM_WAIT,
                             DM_RIGHT_EXCL);

            /*
             * Now turn off events for this file system
             */
            dm_set_disp(sid, fs_hanp, fs_hlen, fs_token, &zero_eventset,
                        DM_EVENT_MAX);

            /*
             * Now that we're done with this token, respond to the event
             * to free up the resources. We could also just release the
             * right and re-use the token
             */
            dm_respond_event(sid, fs_token, DM_RESP_CONTINUE, 0, 0,
                             (void *)0);
```

```
            disp = DM_STEP_TO_NEXT(disp, dm_dispinfo_t *);
      }
      more_disp = dm_getall_disp(sid, , &disp_buf, &ret_ndisp);
}

if (more_disp == -1 ) {
      err_msg("Error obtaining disposition of events for file systems\n");
      do_cleanup();
      return(1);
}
```

Now that no more events can be generated, we can go about draining the session queues. We find all the messages and just respond to them so that the user processes that are awaiting a response can continue (albeit with an error from us).

```
/*
 * We are now at a point where no more new events will be added
 * to the session. We can get all the outstanding tokens and
 * release them by responding to their messages. We could also get
 * the actual messages they correspond to, but we don't bother with
 * that step in this example
 *
 * The length of the token buffer is arbitrary; we pick 1024
 * bytes and increase it if it's not enough.
 */
tokenbuf_len = 1024;
token_buf = (dm_token_t *)malloc(tokenbuf_len);

if (dm_getall_tokens(sid, tokenbuf_len, token_buf, &tok_retlen) == -1) {
      if (errno != E2BIG) {
            err_msg("Can't get tokens for session");
            do_cleanup();
            return(1);
      }
      tokenbuf_len = tok_retlen;
      resize_buf(token_buf, tokenbuf_len);
      if (dm_getall_tokens(sid, tokenbuf_len, token_buf,&tok_retlen)==-1){
            err_msg("Can't get tokens for session");
            do_cleanup();
            return(1);
      }
}

/*
 * We could do a dm_find_eventmsg to get the event message that
 * corresponds to this token, but since we're only interested in
 * returning the tokens so that the user application that was blocked
 * while the (now unserviceable) event was being handled can be
 * released and allowed to continue
 */
start = 0;
tokenp  = token_buf;
while (start < tok_retlen)
      dm_respond_event(sid, *tokenp, DM_RESP_ABORT, EINTR, 0, (void *)0);
      tokenp++;
      start += sizeof(dm_token_t);
}

/*
 * We're done. We can destroy our session
 */
dm_destroy_session(sid);

do_cleanup();
return(0);
```

8.2 Initialization

This example illustrates how a typical DM application would create a session, monitor *mount* events until the file system mounted at `/var/spool/bigqueue` is mounted, then set the disposition for the *read* and `write` managed region events.

The following concepts are illustrated:

- Mount even propagation
- Use of the global handle
- Simple lock upgrade
- Setting event disposition

```
#define OUR_MAX_UMSG   64
extern void            *global_handle;
extern size_t           global_hlen;

char                   *fsname;
size_t                  fslen;
size_t                  fs_hlen;
void                   *fs_hanp;
int                     found;
dm_token_t              fs_token;
dm_right_t              right;
dm_sessid_t             sid;
dm_token_t              token, global_token;
dm_event_set_t          eventset;
dm_eventmsg_t          *msg, *mountbuf;
dm_namesp_event_t      *event;
char                    umsgbuf[OUR_MAX_UMSG];


if (dm_init_service() == -1) {
      err_msg("Can't initialize DMI\n");
      return(0);
}

dm_create_session(DM_NO_SESSION, &sid, "generic_app");

/*
 * In order to set the disposition of the mount event, we have to have
 * exclusive access to the global handle. We first create a context
 * for the token, then request the right.
 */
strncpy(umsgbuf, "global token", OUR_MAX_UMSG);
dm_create_userevent(sid, OUR_MAX_UMSG, umsgbuf, &global_token);
dm_request_right(sid, global_handle, global_hlen, global_token, DM_WAIT,
                 DM_RIGHT_EXCL);

/*
 * Since we only are interested in mount events to begin with,
 * that's the only one we put in our event list
 */
DMEV_ZERO(eventset);
DMEV_SET(DM_EVENT_MOUNT, eventset);

dm_set_disp(sid, global_handle, global_hlen, global_token, &eventset,
            DM_MAX_MSG);
```

Now we just wait for the mount event.

```
/*
 * Since we can't do a dm_get_config() yet to determine the largest
 * possible message, we just guess at about how much space the
 * mount event should take and alloc a buffer that should be
 * more than adequate
 */
mountbuf = (void *)malloc(4096);
found = 0;
do {
      if (dm_get_events(sid, 4096, mountbuf, &rlenp, 1, DM_WAIT)) == -1 ) {
            err_msg("Can't get event list\n");
            continue;
      }
```

```
        msg = (dm_eventmsg_t *)mountbuf;
        if (msg->ev_type != DM_EVENT_MOUNT) {
              /*
               * We shouldn't have received this message, but
               * it never hurts to be even a little robust, even
               * in a somewhat braindead example
               */
              dm_respond_event(sid, msg->ev_token, DM_RESP_ABORT, EINVAL, 0,
                               (void *)0);
        }

        /*
         * Now we've received a mount event. We need to
         * check that it's for the file system we want
         */
        event  = DM_GET_VALUE(msg, data, dm_namesp_event_t *);
        fsname = DM_GET_VALUE(event, name1, char *);
        fslen  = DM_GET_LEN(event, name1, size_t);
        if (strncmp(fsname, "/var/spool/bigqueue", fslen) == 0) {
              /*
               * Found our file system. We need to respond with DM_RESP_CONTINUE
               * so that the DMAPI knows that we want it and not to
               * send the mount event to anyone else
               */
              dm_respond_event(sid, msg->token, DM_RESP_CONTINUE, 0, 0, (void *)0);

              fs_hanp = DM_GET_VALUE(event, handle2, void *);
              fs_hlen = DM_GET_VALUE(event, handle2, size_t);
              found = 1;
        } else {
              /*
               * Since we don't want this event, we do a DM_RESP_DONTCARE which
               * will cause the DMAPI to send the event to someone else
               */
              dm_respond_event(sid, msg->ev_token, DM_RESP_DONTCARE, 0, 0,
                               (void *)0);
        }
  } while (!found);
```

We found the file system we are interested in monitoring. We got a token with the message, but it may reference either no rights, or the wrong ones. We must use `dm_query_right()` to determine what we have, drop any existing locks if necessary, and then get our exclusive lock.

```
  /*
   * Check to see what the rights are that came with the message. If
   * they aren't exclusive, we must go get them
   */
  if (dm_query_right(sid, fs_hanp, fs_hlen, fs_token, &right) == -1 ) {
        err_msg("Can't query rights\n");
        do_cleanup();
        return(1);
  }

  if (right == DM_RIGHT_SHARED || right == DM_RIGHT_NULL) {
        if (right == DM_RIGHT_SHARED) {
              /*
               * Rather than try to upgrade the access right, we drop the
               * shared lock and the fall through to request the exclusive
               * lock
               */
              dm_release_right(sid,fs_hanp,fs_hlen,fs_token);
        }
        dm_request_right(sid, fs_hanp, fs_hlen, fs_token, DM_WAIT, DM_RIGHT_EXCL);
  }

  /*
   * Now we can set the disposition for the real events and release our access
   * rights. We don't respond to the event in case we want to use
   * it later in the life of this process.
   */
  DMEV_ZERO(eventset);
  DMEV_SET(DM_EVENT_READ, eventset);
  DMEV_SET(DM_EVENT_WRITE, eventset);
  dm_set_disp(sid, fs_hanp, fs_hlen, fs_token, &eventset, DM_MAX_MSG);

  dm_release_right(sid, fs_hanp, fs_hlen, fs_token);

  do_cleanup();
  return(0);
```

8.3 Dump

Assume we want to do a primitive dump of the files on the file system mounted at `/var/spool`. The following concepts are illustrated:

- Use of `dm_get_bulkattr()`
- Navigating through a variable number of variable length data structures
- Getting Data Management attribute information
- Getting managed region info
- Getting persistent event list info
- Use of `dm_get_config()` to obtain configuration information

The file system supports persistent opaque and non-opaque Data Management attributes. The following
routines are not shown:

`err_msg()`	generic error reporting routine
`dump_attrs()`	add attributes for a file to the dump file
`dump_mr()`	add the managed regions for a file to the dump file
`dump_eventlist()`	add the event lists for a file to the dump file
`dump_data()`	add the data for a file to the dump file

```
#define OUR_MAX_UMSG   32
#define MAX_FILEREQ    1024
#define NBBY           8     /* number of bits per byte */

size_t            fs_hlen, hlen;
size_t            buflen, attr_len;
size_t            rlenp;
size_t            num_eventbits, ret_event;
size_t            num_mr, ret_mr;
void             *fs_hanp, *hanp
u_int             mask;
int               done;
dm_token_t        fs_token, filetoken;
dm_stat_t        *statbuf, *sbuf
dm_attrlist_t    *attrbuf;
dm_region_t      *mrbuf;
dm_sessid_t       sid;
dm_event_set_t   *eventbuf;
char              umsgbuf[OUR_MAX_UMSG];


if (dm_init_service() == -1 ) {
      err_msg("Can't initialize DMAPI\n");
      return(1);
}

dm_create_session(DM_NO_SESSION, &sid, "DM_dump");

/*
 * Now create a token to hold the context for the rights we're
 * going to get for the file system handle. We store the string with
 * the message so that we can pick up later if we should happen to
 * crash
 */
strncpy(umsgbuf, "FS token", OUR_MAX_UMSG);
dm_create_userevent(sid, OUR_MAX_UMSG, umsgbuf, &fs_token);
dm_path_to_fshandle("/var/spool", &fs_hanp, &fs_hlen);
dm_request_right(sid, fs_hanp, fs_hlen, fs_token, DM_WAIT, DM_RIGHT_SHARED);
```

Before we do any real work, we go through a lot of buffer initialization foo.

```
/*
 * Allocate a buffer to hold all the stat information. Note
 * that if we fail later on in this example we don't free up
 * the buffer. In a real example, this would cause eventual memory
 * loss. We pick an arbitrary number for the maximum number of
 * files that we want on each call to dm_get_bulkattr().
 */
buflen = sizeof(dm_stat_t) * MAX_FILEREQ;
```

```
5935                    statbuf = (dm_stat_t *)malloc(buflen);
5936
5937                    /*
5938                     * Initialize the opaque offset cookie that we use in
5939                     * successive calls to dm_get_bulkattr()
5940                     */
5941                    dm_init_attrloc(sid, fs_hanp, fs_hlen, fs_token, &locp);
5942
5943
5944                    /*
5945                     * Before we start iterating over all the file info,
5946                     * we get a token that we'll use as a placeholder to get rights on
5947                     * each file object. We need the rights as a dump application so that
5948                     * the file doesn't change out from underneath us.
5949                     *
5950                     * We can use the same token and avoid the overhead of repeated calls to
5951                     * dm_create_userevent() within the loop. We don't care about associating
5952                     * any text message with this token
5953                     */
5954                    dm_create_userevent(sid, 0, (void *)0, &filetoken);
5955
5956                    /*
5957                     * Allocate a buffer that we think will be large enough to hold
5958                     * all the special DM attributes for a file.
5959                     */
5960                    attr_len = sizeof(dm_attrlist_t) * 1024;
5961                    attrbuf  = malloc(attr_len);
5962
5963                    /*
5964                     * We also need a buffer for the per-file managed region info.
5965                     * To accurately size our managed region buffer, we just get the
5966                     * maximum number of persistent managed regions supported for this
5967                     * file system. There may be more than this number, but they won't
5968                     * be persistent. If a file has more than the persistent number,
5969                     * we assume it must be active and so we just skip it
5970                     */
5971                    dm_get_config(fs_hanp, fs_hlen, DM_CONFIG_PERS_MANAGED_REGIONS, (long)&num_mr);
5972                    mrbuf = (dm_region_t *) malloc(num_mr * sizeof(dm_region_t));
5973
5974
5975                    /*
5976                     * Get the number of persistent event mask bits associated with this
5977                     * file system so that we can size our event mask buffer correctly
5978                     */
5979                    dm_get_config(fs_hanp, fs_hlen, DM_PERS_EVENT_COUNT, (long)&num_eventbits);
5980                    eventbuf = (dm_event_set_t *) malloc(num_eventbits/NBBY);
5981
5982
5983                    /*
5984                     * Set the mask to use in the dm_get_bulkattr() call. This mask
5985                     * says return everything but the file's change indicator
5986                     */
5987                    mask = DM_AT_HANDLE | DM_AT_EMASK | DM_AT_PMANR | DM_AT_PATTR |
5988                           DM_AT_DTIMES | DM_AT_STAT;
5989
5990                    /*
5991                     * Now we iterate over all the files in the file system
5992                     */
5993                    done = 0;
5994                    do {
5995                          more = dm_get_bulkattr(sid, fs_hanp, fs_hlen, fs_token, mask, *locp,
5996                                                 buflen, statbuf, &rlenp);
5997                          if (more == -1) {
5998                                err_msg("Can't get stats for file system\n");
5999                                do_cleanup();
6000                                return(1);
6001                          }
6002                          sbuf = statbuf;
6003
6004                          while (sbuf != NULL) {
6005                                /*
6006                                 * Get the file handle out of the stat struct. We use the macros
6007                                 * since it's variable length. Once we have the handle, get
6008                                 * shared access to it
6009                                 */
6010                                hanp = DM_GET_VALUE(sbuf, st_handle, void *);
6011                                hlen = DM_GET_LEN(sbuf, st_handle, size_t);
6012                                dm_request_right(sid,hanp,hlen,filetoken,DM_WAIT,DM_RIGHT_SHARED);
6013
6014                                /*
6015                                 * Now we get the DM attributes for the file. In a real example,
```

```
6016                          * we'd probably have to iterate over this call to get all the
6017                          * info.
6018                          */
6019                         dm_getall_dmattr(sid, hanp, hlen, filetoken, attr_len, attrbuf,
6020                                          attr_rlenp);
6021                         dump_attrs(hanp, hlen, attrbuf, attr_rlenp);
6022
6023                         /*
6024                          * Get the list of persistent managed regions for this file and
6025                          * save them to the dump file
6026                          */
6027                         dm_get_region(sid, hanp, hlen, filetoken, num_mr, mrbuf, &ret_mr);
6028                         dump_mr(hanp, hlen, mrbuf, ret_mr);
6029
6030
6031                         /*
6032                          * Get the list of persistent events that are set with this
6033                          * file. We only have room for one event set, so if there are
6034                          * more than that, we bail
6035                          */
6036                         if (dm_get_eventlist(sid, hanp, hlen, filetoken, 1, eventbuf,
6037                                              &ret_event) != 0 )
6038                         {
6039                               err_msg("Can't get event list for file\n");
6040                               continue;
6041                         }
6042                         dump_eventlist(hanp, hlen, eventbuf, ret_event);
6043
6044                         /*
6045                          * Now that we've got all the metadata, we can go off an do
6046                          * dm_read_invis() on the file to get the actual file data
6047                          */
6048                         dump_data(hanp, hlen, sbuf);
6049
6050                         /*
6051                          * Now release the shared right we had on the file so that we
6052                          * can get a right on another file, using the same token
6053                          */
6054                         dm_release_right(sid, hanp, hlen, filetoken);
6055
6056                         sbuf = DM_STEP_TO_NEXT(sbuf, dm_stat_t *);
6057                   }
6058
6059                   } while (more == 1);
6060
6061             /*
6062              * Now that we've dumped all the data we need, we can release the
6063              * tokens and our buffers
6064              */
6065             dm_respond_event(sid, fs_token, DM_RESP_CONTINUE, 0, 0, (void *)0);
6066             dm_respond_event(sid, filetoken, DM_RESP_CONTINUE, 0, 0, (void *)0);
6067
6068             do_cleanup();
6069             return(0);
```

8.4 Migration

There are two parts to this example. This first part illustrates how to stage out a file, while the second part handles a stagein.

8.4.1 Stageout

This example will stage out a 512 megabyte file names `/test/foo`. The first 64K will remain as a fence post. The remainder of the file will be staged out in 32 megabyte clusters.

The following concepts are illustrated:

- Use of the file change indicator
- Invisible read
- Setting managed region
- Punching holes

```
char             *buf;
void             *hanp;
size_t            hlen;
size_t            retrgns;
size_t            nchunks;
int               change_end, change_start,
off_t             off;
dm_token_t        filetoken;
dm_stat_t         statbuf;
dm_region_t       rgnbuf[2];
dm_size_t         roff;
dm_off_t          rlen;
dm_sessid_t       sid;

if (dm_init_service() == -1) {
      err_msg("Can't initialize DMI\n");
      return(0);
}

dm_create_session(DM_NO_SESSION, &sid, "generic_app");
dm_path_to_handle("/test/foo", &hanp, &hlen);

/*
 * In order to stat the file, we need a shared lock.
 */
dm_create_userevent(sid, 0, (void *)0, &filetoken);
dm_request_right(sid,hanp,hlen,filetoken,DM_WAIT,DM_RIGHT_SHARED);

/*
 * While writing the file out to tertiary storage, we drop locks. We first
 * get the file change indicator, and query it after we're done.
 * If it changed, then we give up
 */
dm_get_fileattr(sid, hanp, hlen, filetoken, DM_AT_CFLAG, statbuf);
change_start = statbuf.dt_change;

/*
 * We don't bother with any DM attributes, just the data.
 * We write the file out in 32 meg chunks.
 */
nchunks = (512 * 1MEG) / CHUNKSIZE;
for (i=0; i<nchunks; i++) {
      dm_read_invis(sid, hanp, hlen, token, off, CHUNKSIZE, buf);
      dump_data_to_archive(hanp, hlen, off, buf);
      off += CHUNKSIZE;
}
dm_release_right(sid, hanp, hlen, filetoken);

/*
 * Store an DM application specific information, such as the file size,
 * file handle, etc., with the data
 */
dump_myinfo_to_archive(hanp, hlen);

/*
 * Check the file change indicator to see if the file changed
 * while we were doing other things. If not, then set a managed
 * region on the file
```

```
     */
    dm_request_right(sid, hanp, hlen, filetoken, DM_WAIT, DM_RIGHT_EXCL);
    dm_get_fileattr(sid, hanp, hlen, filetoken, DM_AT_CFLAG, statbuf);
    change_end = statbuf.st_change;
    if (change_start != change_end) {
         err_msg("File changed, bailing...\n");
         do_cleanup();
         return(1);
    }

    /*
     * Set up the managed regions so that the first 64K won't cause events
     * to be generated, but a foray into the rest of the file will generate
     * events
     */
    rgnbuf[0].rg_offset = 0;
    rgnbuf[0].rg_size   = FENCESIZE;
    rgnbuf[0].rg_flags  = DM_REGION_NOEVENT;
    rgnbuf[1].rg_offset = FENCESIZE;
    rgnbuf[1].rg_size   = (512 * 1MEG) - FENCESIZE;
    rgnbuf[1].rg_flags  = DM_REGION_READ | DM_REGION_WRITE | DM_REGION_TRUNCATE;
    dm_set_region(sid, hanp, hlen, filetoken, 2, rgnbuf, &retrgns);

    /*
     * Punch a hole in the file. We assume that we know that what the
     * rounding constraints are so that we don't have to do a dm_probe_hole()
     */
    dm_punch_hole(sid, hanp, hlen, token, rgnbuf[1].rg_offset, rgnbuf[1].rg_size);

    /*
     * We're done. Release the token
     */

    dm_respond_event(sid, filetoken, DM_RESP_CONTINUE, 0, 0, (void *)0);
```

8.4.2 Stagein

This example will stage in the file that was staged out in the above example.

There is a master daemon that receives messages from the kernel, and sends them on to worker bees for processing. The master daemon is only monitoring the *`read`*, *`write`*, and *`truncate`* managed region events in this example for the file system `/test`.

There is some magic here that is not shown. The master daemon knows about, and has access to, two other sessions that are used to perform event-specific handling. Information is shared between the master daemon and these other processes through some application-specific mechanism that is not shown. This could be shared memory, a socket, a well-known file, or any other mechanism. The details are left to the dazed reader.

The following concepts are illustrated:

- Sharing of tokens and sessions
- Setting event disposition
- A simple get_event loop
- Having a master daemon *move* an event to another session
- Complex lock upgrade
- Setting managed regions

```
    extern dm_sessid_t     rw_sid, trunc_sid;

    void             *fs_hanp;
    void             *msgbuf;
    size_t            fs_hlen;
    size_t            msgsize, msgbuflen;
    dm_sessid_t       sid;
    dm_token_t        fs_token, newtoken;
    dm_event_set_t    eventset;
    dm_eventmsg_t    *msg;

    if (dm_init_service() == -1) {
         err_msg("Can't initialize DMI\n");
```

```
        return(0);
}

dm_create_session(DM_NO_SESSION, &sid, "generic_app");

/*
 * Since we'll be communicating with other processes, we do some
 * magic setup to get their sids, establish communications channels,
 * etc. We could use dm_send_event(), stuff the data in a shared memory
 * region, open a socket, or whatever
 */
setup_communications(sid, &rw_sid, &trunc_sid);

dm_path_to_fshandle("/test", &fs_hanp, &fs_hlen);

/*
 * Now get a token and rights so that we can set the disposition
 * of events
 */
dm_create_userevent(sid, 0, (void *)0, &fs_token);
dm_request_right(sid, fs_hanp, fs_hlen, fs_token, DM_WAIT, DM_RIGHT_EXCL);

/*
 * Set the disposition of the events we want to monitor
 */
DMEV_ZERO(eventset);
DMEV_SET(DM_EVENT_READ, eventset);
DMEV_SET(DM_EVENT_WRITE, eventset);
DMEV_SET(DM_EVENT_TRUNCATE, eventset);
dm_set_disp(sid, fs_hanp, fs_hlen, fs_token, &eventset, DM_MAX_MSG);

dm_release_right(sid, fs_hanp, fs_hlen, fs_token);
```

The master daemon now enters a simple loop where it will spend all its time. It simply asks the DMAPI for more messages and dispatches them to its worker processes.

```
/*
 * Find out the size of the largest message that can be delivered
 * on this file system. We use this to size an event buffer to get
 * an arbitrary (16 in this example) number of messages at the same
 * time.
 */
dm_get_config(fs_hanp, fs_hlen, DM_MAXMSG_SIZE, (long)&msgsize);
msgbuflen = msgsize * 16;
msgbuf = (void *)malloc(msgbuflen);

/*
 * Enter a simple loop, looking for messages. We don't worry about
 * resizing the buffer
 */
for (;;) {
      dm_get_events(sid, msgbuflen, msgbuf, &ret_msglen, 0, DM_WAIT);
      msg = (dm_eventmsg_t *)msgbuf
      while (msg != NULL) {

            /*
             * For read and write events, we send them to other processes
             * with 'well known' sids that are handling these things.
             */
            if (msg->ev_type == DM_EVENT_READ || msg->ev_type == DM_EVENT_WRITE) {
                  dm_move_event(sid, msg->ev_token, rw_sid, &newtoken);

            } else if (msg->ev_type == DM_EVENT_TRUNCATE) {
                  dm_move_event(sid, msg->ev_token, trunc_sid, &newtoken);

            } else {
                  err_msg("Unknown event type\n");
                  dm_respond_event(sid, msg->ev_token,
                        DM_RESP_ABORT,EINVAL, 0, (void *)0);
                  continue;
            }
            msg = DM_STEP_TO_NEXT(msg, dm_eventmsg_t *);
      }
}
```

The worker bee processes also do some initial setup, which won't be shown. For a simple stagein, we'll assume we've receive a *DM_EVENT_READ* event on a managed region. We join our fearless process at the point in which it has received the event that was directed to it from the master daemon, and is starting to reload the data.

```
int               change_start;
void             *hanp;
size_t            hlen;
size_t            nchunks;
size_t            retrgns;
dm_off_t          off;
dm_size_t         len;
dm_right_t        right;
dm_sessid_t       sid;
dm_stat_t         statbuf;
dm_eventmsg_t    *msg;
dm_data_event_t  *read_event;
dm_region_t       rgnbuf[1];

msg = eventbuf;
read_event = DM_GET_VALUE(msg, data, dm_data_event_t *);

hanp = DM_GET_VALUE(read_event, handle, void *);
hlen = DM_GET_LEN(read_event, handle, size_t);

/*
 * Check to see what the rights are that came with the message. If
 * they aren't exclusive, we must go get them
 */
dm_query_right(sid, hanp, hlen, msg->ev_token, right);

if (right == DM_RIGHT_SHARED) {
      /*
       * We really need exclusive. We'll try to upgrade the lock,
       * but if that fails, we'll have to drop it and go to
       * sleep.
       */
      if (dm_request_right(sid, hanp, hlen, msg->ev_token, DM_RIGHT_EXCL,
                           DM_NOWAIT) == -1
      {
            if (errno != EAGAIN) {
                  err_msg("Can't upgrade lock\n");
                  do_cleanup();
                  return(1);
            }
            /*
             * Before we drop the lock, get the file change indicator
             */
            dm_get_fileattr(sid, hanp, hlen, msg->ev_token, DM_AT_CFLAG,
                            statbuf);
            change_start = statbuf.dt_change;

            dm_release_right(sid, hanp, hlen, msg->ev_token);
            dm_request_right(sid,fshanp,fshlen,msg->ev_token, DM_WAIT,
                             DM_RIGHT_EXCL);

            /*
             * Now that we've come back from sleeping, see if the file changed.
             * If so, we just bail.
             */
            dm_get_fileattr(sid, hanp,hlen,msg->ev_token,DM_AT_CFLAG,statbuf);
            if (statbuf.dt_change != change_start) {
                  err_msg("File changed. Bailing...\n");
                  do_cleanup();
                  return(1);
            }
      }
} else if (right == DM_NONE) {
     dm_request_right(sid,hanp,hlen,msg->ev_token,DM_WAIT,DM_RIGHT_EXCL);}
}
```

The worker bee is now at the point where it has exclusive access to the file. This is needed for `dm_write_invis()`.

```
6362                    /*
6363                     * Now that we have exclusive access to the file, we need to
6364                     * go off and find where we stored the file's data in our
6365                     * repository
6366                     */
6367                    offset = DM_GET_VALUE(read_event, offset, dm_off_t);
6368                    len    = DM_GET_VALUE(read_event, len, dm_size_t);
6369                    find_our_file(hanp, hlen, offset, len);
6370
6371
6372                    /*
6373                     * Restore the data for the file.
6374                     * We'll assume that the file length is some nice integral multiple
6375                     * of our chunksize;
6376                     */
6377                    nchunks = len / CHUNKSIZE;
6378                    for (i=0; i<nchunks; i++) {
6379                         get_data_from_archive(hanp, hlen, off, buf);
6380                         dm_write_invis(sid, hanp, hlen, token, off, CHUNKSIZE, buf);
6381                         off += CHUNKSIZE:
6382                    }
6383
6384
6385                    /*
6386                     * Clear the managed region
6387                     */
6388                    rgnbuf[0].rg_offset = 0;
6389                    rgnbuf[0].rg_size = 0;
6390                    rgnbuf[0].rg_flags = DM_REGION_NOEVENT;
6391                    dm_set_region(sid, hanp, hlen, msg->ev_token, 1, rgnbuf, &retrgns);
6392
6393
6394                    /*
6395                     * We're done. Release the token
6396                     */
6397
6398                    dm_respond_event(sid, msg->ev_token, DM_RESP_CONTINUE, 0, 0,(void *)0);
```

8.5 Multi-threaded Application

This example illustrates how multiple threads (or processes) could work together on the same file.

The master daemon hangs around waiting for events. When it finds a *read* event, it sends it via an application specific message system to a thread that handles those events.

The following concepts are illustrated:

- Using a master daemon to receive events from the DMAPI
- Sending message data to another thread or process
- Use of the same token and access rights by multiple threads
- Use of the same sid by multiple threads.

```
size_t            msgbuflen, ret_msglen;
void             *msgbuf;
dm_sessid_t       sid;
dm_eventmsg_t    *msg;
dm_data_event_t  *read_event;


for (;;) {
      dm_get_events(sid, msgbuflen, msgbuf, &ret_msglen, 0, DM_WAIT);
      msg = (dm_eventmsg_t *)msgbuf
      while (msg != NULL) {
            if (msg->ev_type == DM_EVENT_READ) {
                  read_event = DM_GET_VALUE(msg, data, dm_data_event_t *);
                  send_to_reader(sid, msg->ev_token, read_event, msg->ev_data);
            }
            msg = DM_STEP_TO_NEXT(msg, dm_eventmsg_t *);
      }
}
```

The reader thread sits in a loop scanning its message queues looking for work. When a message arrives, it first obtains exclusive access rights using the token that is part of the message. Then it parses the message to determine how large a read it is. If the read is for less than 4 megabytes, it handles it by itself. Otherwise, if the read is for more than 4 megabytes, it breaks it into 4 meg chunks, and lets its own private worker threads do the actual reading.

```
size_t            msglen;
size_t            hlen;
void             *hanp;
dm_off_t          curoff;
dm_size_t         nchunks;
dm_sessid_t       sid;
dm_token_t        token;
dm_data_event_t  *msg;


for (;;) {
      get_msg_from_master(sid, token, msg, msglen);

      hanp = DM_GET_VALUE(msg, handle, void *);
      hlen = DM_GET_LEN(msg, handle, size_t);
      get_exclusive_access(sid, token, hanp, hlen);

      if (msg->de_length < 4MEG) {
            doit_myself();
            release_access_rights(sid, token);
            continue;
      }

      /*
       * The request is large enough to be handled by our worker threads.
       * We split the request in 4 meg chunks and pass the request and
       * the token off to the threads. We assume the length and offset
       * are all nicely aligned.
       */
      curoff  = msg->de_offset;
      nchunks = msg->de_length / CHUNKSIZE;
      for (i=0; i<nchunks; i++ )
```

```
                send_to_workerbee(sid, token, hanp, hlen, curoff);
                curoff += CHUNKSIZE;
        }

        wait_for_completion();
        /*
         * Since we got the token, we take responsibility for releasing
         * it. Any one of our worker threads could also do it.
         */
        release_access_rights(sid, token);
}
```

The worker bee threads are very simple; they scan their own private message queues and look for work. All they have to do is write the data to the file. We assume that they know where to get the data.

```
for (;;) {
        get_msg_from_reader(sid, token, hanp, hlen, off);

        get_data_from_archive(hanp, hlen, off, CHUNKSIZE, buf);
        dm_write_invis(sid, hanp, hlen, token, off, CHUNKSIZE, buf);

        send_completion_to_reader();
}
```