# Direct Access File System Application Programming Interface (DAFS API)

## Version: 1.0

**Revision Date: November 17, 2001**

Document Status

The DAFS Application Programming Interface, Version 1.0 was created by the DAFS Collaborative.

# CHAPTER 1: OVERVIEW AND INTRODUCTION

## 1.1 OVERVIEW

This document specifies an Applications Programming Interface (API) for the Direct Access File System (DAFS). DAFS is a new protocol for "local file sharing" over advanced memory-to-memory networks such as the Virtual Interface (VI) Architecture and Infiniband. The DAFS Protocol Specification is presented in a separate document.

This version of the API specification contains a set of interfaces that capture the basic file operations (open, close, read, write) in DAFS. It will grow over time to include the full richness of DAFS including locking, fencing, and chaining.

It is intended that this will be a common, portable API, covering equally well (at a minimum) POSIX-compliant and Win32-compliant systems.

### 1.1.1 INTRODUCTION

A main goal of the DAFS API is enabling the richness of DAFS for the application developer, while concurrently enabling maximum performance. A second goal is to allow for the use of alternative transports as they develop. Finally, to be most useful, the right blend of exposure and hiding of internal complexity must be achieved.

## 1.2 ATTRIBUTES OF THE API

The DAFS API provides a convenient programmatic interface that allows applications to securely access files over low-latency, high-throughput networks. The interface is simplified and file-oriented, hiding many of the details of the DAFS protocol.

### 1.2.1 RICHNESS

The DAFS protocol provides a rich and comprehensive set of operations for manipulating, reading and writing, and synchronizing file objects on a server.

To the protocol, the DAFS API adds session and local resource management, signalling, flow control, convenient interfaces to direct the numerous modes of DAFS data transfers, and many others. These additions do not supersede the DAFS protocol interface. Instead, they form the foundation on which the protocol can be easily directed and employed by the application programmer.

The DAFS API will also address the case where the DAFS server may make a request of the client, for example lock management or flow control. This important subset of the DAFS protocol requires careful implementation on the client, and the DAFS API is intended to substantially support this need.

### 1.2.2 PERFORMANCE

A significant performance advantage can be obtained by programs which do not transition into the kernel for network data transfer. The DAFS API is intended to be implemented completely in user space, except of course for making necessary requests of transport provider-dependent kernel support during connection setup and teardown, event management, and so on.

Combined with kernel avoidance and a careful event strategy, this can result in full performance up to the capabilities of the hardware of the machine, the network and of the connected DAFS server.

Therefore, an application employing the DAFS API can expect significant advantages over a kernel-based implementation, such as an NFS client over a traditional high-speed TCP connection.

### 1.2.3 INTEGRATION WITH CONVENTIONAL I/O SUBSYSTEM

Since this API is completely new and is being constructed from scratch, transparent interoperation with existing file access APIs is not planned. In fact, to provide any measure of transparency would require compromising performance. Therefore it is not a requirement to do so. However, the DAFS API is intended to be file-like in nature, and to make the effort of creating or moving an application to be DAFS-enabled a relatively straightforward process.

At the lower layer, it is a goal for the DAFS Provider to operate over a variety of transports. While internal changes must be in place to use any new transport, it is important the fundamental DAFS API be stable across all instances.

### 1.2.4 BALANCE

As with any API, a decision must be made to balance usefulness and complexity. It is a goal of this specification to provide the “right” balance of what the API exposes and hides from the application. It is expected that this balance will be the subject of much discussion.

A primary goal is to encapsulate transport-specific details and features to the extent possible. For example, when running over a compliant VI implementation, reliable delivery semantics are required. Setting this attribute will be performed automatically by the DAFS library. Likewise,

since the interface to an appropriate transport may change over time, those transport-specific details should be handled behind the scenes.

However, there are valid reasons to want certain transport-specific details to be accessible to the application. One example is memory registration, for transports which require this to be done up-front. Since NIC resources (such as TPT slots) are likely to be scarce, an application might reasonably want to register memory once, subsequently using it to back both DAFS buffers and descriptors for use with the native transport interface.

While it should be possible to use the DAFS API without knowledge of what is going on under the hood, these conflicting goals argue for the inclusion of a set of interfaces that allow the extraction (and perhaps tweaking) of various transport-specific bits of data. Further complicating the design of these interfaces is the goal of being able to implement DAFS on top of a variety of low-latency, high-throughput transports.

## 1.3 Details to be addressed by the API

### 1.3.1 Memory registration

Applications which are being coded to the DAFS API are recommended to pre-register memory buffers for DAFS data transfer operations. This important difference from POSIX and Win32 enables the performance advantages of DAFS by removing memory wiring operations and their user-to-kernel transitions from the critical I/O paths. While the DAFS Provider can register and bind application memory as needed, the pre-registration enables the Provider to most effectively manage scarce hardware resources for maximum efficiency and performance.

Memory regions may be registered one or more times by the application, though there is no good reason to do this, and may waste scarce resources. Once registered, such memory can be used for any valid purpose.

### 1.3.2 Sync/Async

The DAFS API will be fundamentally asynchronous. Asynchronous interfaces will be the most direct method for supporting the asynchronous capabilities of the DAFS protocol, and will allow maximum throughput. The synchronous interfaces offer ease of application porting and low latency, but suffer from lower transport utilization.

A method of indicating the desired asynchronous I/O completion at I/O initiation time is an important element of the DAFS API. By specifying precisely where a completion will occur, the specific event may be awaited by the application if desired. Or, events may be received in completion order by application-defined groups. This feature of the DAFS API will allow applications to most efficiently use communications channels for all pur-

poses, without having to invoke additional overhead at I/O completion time.

### 1.3.3 PROTOCOL PRIMITIVES

The DAFS System Development Kit (SDK) provides a set of low-level interfaces for encoding and decoding raw DAFS requests and responses over the wire. These interfaces are not a part of the DAFS API, but do provide a way for vendors to get up and running quickly.

# CHAPTER 2: DAFS PROVIDER INTERFACE ARCHITECTURE

## 2.1 OVERVIEW

The overall Architecture for the DAFS Provider and its interactions with both the DAFS Consumer and memory-to-memory transport provider is shown in Figure 1.

The DAFS Provider has four subcomponents:

- Client DAFS File System Engine. This component is responsible for interpreting the Primary DAFS Provider API. This includes both Consumer interfaces for file-level operations and transport resource management.
- DAFS SDK/DAFSgen protocol stubs. This component is responsible for marshalling and unmarshalling parameters into the on-the-wire DAFS Protocol format.
- Transport Resource Management. This component is responsible for management of DAFS transport resources in a relatively transport-independent manner.
- DAFS Transport engine. This component is responsible for mapping high level operations on to the specific APIs offered by a given memory-to-memory transport provider.

There are two separate APIs shown that the DAFS Consumer may use to interact with the DAFS Provider.

- Primary DAFS Provider API. This is the API defined by this document. It is in interface that is independent of the underlying memory-to-memory transport(s) in use. It provides an abstract file-like interface to the DAFS protocol, with some additional operations for resource management.
- Secondary DAFS Provider API. There are multiple instances of this API--one for each Transport Provider. Instances of this API for important transports (such as VI and Infiniband) will be incorporated as non-normative appendices to this Specification.

.

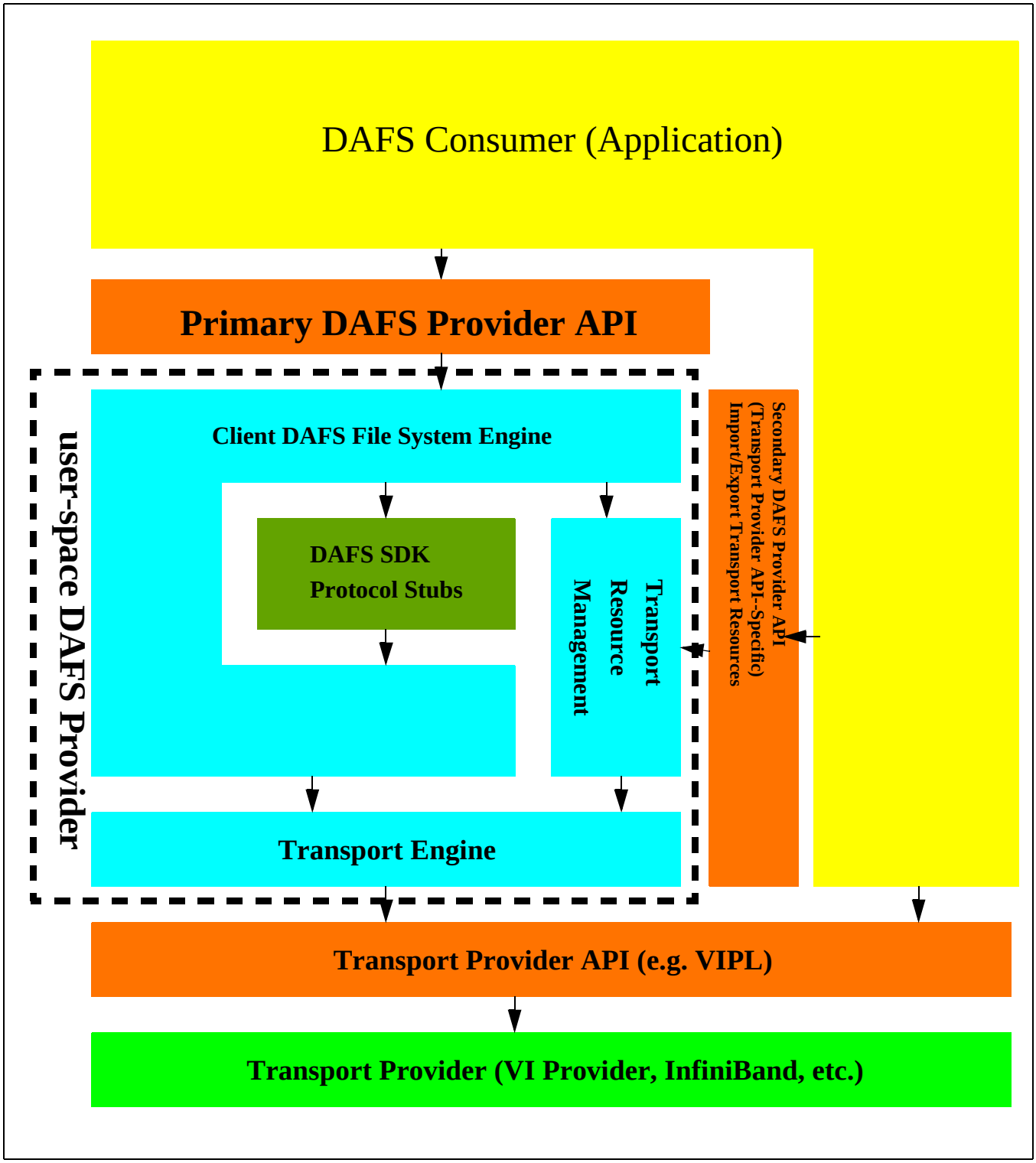


**Figure 1 DAFS Provider Architecture**

## 2.2 API Relationship to the Native Operating System I/O Subsystem

The Consumer interface to the DAFS Provider is independent of the native operating system I/O Subsystem. That is, data structures and objects to represent DAFS Provider resources (e.g. open files) that are passed across the DAFS API are only meaningful to the DAFS Provider, the Consumer is not able to pass handles to these DAFS Provider resources to native operating system APIs. In other words:

- In a POSIX environment DAFS_FILE_HANDLE is not a file descriptor. It may not be passed to select(2), read(2), write(2), aio_read(3), aio_write(3) or any other standard system interface. aio_wait(3) is not able to wait for the completion of a DAFS read or write request.
- In a Win32 environment a DAFS_FILE_HANDLE is not a Win32 handle to a kernel-mode object (not a File Object, not an Event object). It may not be passed to WaitForSingleObject() WaitForMultipleObjects(), ReadFile(), WriteFile() or any other standard system interface.

Furthermore, a DAFS Handle is only valid in the context of the process in which the DAFS Handle was created. Consequently, any attempt to use a DAFS Handles in a child process created by a POSIX fork(2) will result in undefined behavior.

Signals are not generated when DAFS I/O operations complete. If a signal handler that interrupts the DAFS Provider performs a longjump outside the DAFS provider, the behavior of the DAFS Provider is undefined.

## 2.3 API Relationship to Transport Provider and DAFS Protocol

### 2.3.1 Primary API to DAFS Provider

The primary Consumer interface to the DAFS Provider is independent of the underlying memory-to-memory transport(s) in use. It provides a file level API to the Consumer that hides transport details and details of the DAFS Protocol. As much as possible the DAFS API tries to present a file access API that is similar to and functionally compatible with both POSIX and Win32, though the requirement that memory be registered prior to initiation of DAFS I/O differs from both.

#### 2.3.1.1 Abstracting Transport-Level Resources

Individual transport-level resources are abstracted by this API allowing the DAFS Provider the opportunity to do all of the following without requiring explicit Consumer involvement:

- to transparently recover from transport-level errors (including recovery involving the server's replay cache)
- to offer transparent path load balancing
- to offer transparent path failover capabilities

Important structures that the DAFS Provider must construct in order to interact with the transport provider are wrapped by the DAFS Provider. From the perspective of the primary application interface to the DAFS Provider, the transport-level constructs are entirely hidden by the DAFS Provider underneath higher level abstractions presented to the consumer by the DAFS Provider.

Therefore, even though they may have similar names, and serve similar functions, the Application must never directly use a DAFS Handle directly with the transport provider. Specifically, a DAFS_MEM_HANDLE is not a VIP_MEM_HANDLE, nor is it an Infiniband memory key. Attempts by the Consumer use a DAFS_MEM_HANDLE in a VIPL or Infiniband interface will produced undefined results. Similarly attempts by the Consumer to use a VIP_MEM_HANDLE or memory key in a DAFS Provider interface will produce undefined results.

#### 2.3.1.2 Abstracting DAFS Protocol Structures

Individual DAFS Protocol operations and constructs are abstracted underneath the Primary DAFS Provider API. For example, none of the following are directly exposed to the Consumer:

- protocol-level sessions
- protocol-level handles to files
- protocol-level cookies representing protocol-level state for an open file (e.g. stateid)
- protocol-level cookies representing session state (e.g. registered credentials)

Instead of exposing them directly, the DAFS Provider abstracts these entities, providing a DAFS Handle to the DAFS Consumer. Thus, for example, when the DAFS Provider returns a DAFS_FILE_HANDLE from dafs_open_file, the DAFS_FILE_HANDLE is not the same thing as the protocol file handle returned by the server in response to a lookup operation. The same applies to all other DAFS Handles.

### 2.3.2 Secondary APIs to Import/Export Transport Provider Resources

Because some transport provider resources are severely limited, it is desirable to allow a DAFS Consumer that also directly uses the transport provider for non-DAFS purposes (e.g. for peer-to-peer communications) to be able to share transport resources with the DAFS Provider.

It is possible that a series of non-normative appendices could be added to the normative part of this specification in order to specify secondary, transport provider API-specific interfaces that allow certain critical resources to be shared between the DAFS Provider and the DAFS Consumer. As an example, the Appendix for VIPL might provide a means to import/export: VIP_NIC_HANDLEs, VIP_PROTECTION_HANDLEs, and VIP_MEM_HANDLEs, and possibly VI Host Addresses. We do not anticipate that VIP_VI_HANDLEs or VIP_CQ_HANDLEs would be exposed through this interface--connection endpoints and completion management structures are not shared between the DAFS Provider and the DAFS Consumer. To date this has not been considered necessary, and so though the possibility remains, this has not been done.

### 2.3.3 Lack of Secondary APIs to expose Raw DAFS Protocol operations

Although it might be possible to provide a secondary set of APIs to import/export DAFS Protocol State associated with a DAFS Handle representing a protocol construct (similar to the secondary APIs for importing/exporting transport resources), it is our current feeling that doing so will introduce significant complexity and provides inappropriate access “under the hood” of the DAFS Provider. An example of a difficulty here is the need for the DAFS Provider to maintain consistency of the view of its client filesystem engine across any direct protocol operations that the Consumer may choose to invoke.

For example, if the DAFS Provider has a DAFS_FILE_HANDLE representing an open file, and the Consumer performs a locking operation on the file directly, the DAFS Provider must be aware of this operation,.

Why? The locking operation will change the stateid on the server for the file, and subsequent operations by the DAFS Provider using its old stateid will fail.

The complexity of determining and defining all of these interactions leads us to the conclusion that defining an API that allows the Consumer direct access to the protocol bypassing the DAFS Provider is outside the scope of the definition of a standard DAFS Provider API.

For Applications that find the DAFS Provider inadequate, the DAFS SDK is available for use by applications that wish to construct their own DAFS Protocol messages and submit them directly to the transport provider. However, there is no standard means available for the Consumer to share the DAFS Protocol state with the DAFS Provider. Thus direct use of the DAFS SDK by an Application is an all-or-nothing affair on a given set of DAFS Protocol resources.

## 2.4 Server Connection Management

The DAFS Provider is responsible for managing the low level transport and protocol interactions with a physical DAFS server. In particular, the DAFS Provider is responsible for transport connection establishment, and for DAFS session establishment. The DAFS Provider is responsible for low level error recovery when transport errors occur. In addition, when multiple physical paths exist from the physical DAFS client to the physical DAFS server, the DAFS Provider is responsible for performing any trunking, multi-pathing, or load balancing, and for dealing with any server failover or dataset migration that may occur. Abstraction is key to achieving these goals. The DAFS API provides no notion of server or session, dealing only in path names, files, directories, and their contents, and memory buffers.

If multiple physical paths exist to the physical server, a sophisticated DAFS Provider may choose to create multiple DAFS protocol sessions (perhaps one per physical path), multiplexing independent operations over the various paths.

## 2.5 Completion Groups

The DAFS Provider API introduces the notion of a DAFS Completion Group. A DAFS Completion Group provides a mechanism that allows the Consumer to provide to the DAFS Provider an indication of when a group of I/O operations are likely to be waited together. Specifically:

- If POSIX file I/O were being used, then their aio_result structures would have been passed in together into aio_suspend().
- If Win32 file I/O were being used, either they would have been bound to the same I/O completion port or, their Overlapped Event Handles would have been passed in together into a WaitForMultipleObjects().

The DAFS Completion Group provides an abstract way to allow the DAFS Provider to take advantage of transport provider completion queues. The DAFS Completion Group abstraction improves the DAFS Consumer's ability to provide good hints on I/O grouping. At I/O initiation time (e.g. in dafs_async_read() or dafs_async_write()) the Consumer may specify a Completion Group to which the I/O completion notification for the operation should be delivered. When a Consumer thread wants to find the next I/O completion notification in a Completion Group the Consumer invokes dafs_cg_done() or dafs_cg_wait() which will return a description of the next I/O completion notification available in the completion group.

Alternatively, the application may specify a NULL Completion Group handle at I/O initiation time, thereby indicating that the Consumer does not wish for the I/O completion notification to be delivered to any Completion Group. In this case the Consumer must obtain the I/O completion notifi-

cation by querying the state of particular I/O operations using either dafs_io_done() or dafs_io_wait().

There are absolutely no restrictions on which I/O operations the Consumer may group together into a single DAFS Completion Group. In particular:

- Different I/O operations on different open server instances may specify the same Completion Group.
- Different I/O operations on different open file instances may specify the same Completion Group.
- Different I/O operations on a single open server instance may specify different Completion Groups.
- Different I/O operations on a single open file instance may specify different Completion Groups.

Furthermore, it is expected that DAFS Completion Groups will span transport hardware resources. Specifically, even if the transport provider does not support the grouping of I/Os using different NICs, a single DAFS Completion Group shall be able to group arbitrary I/Os, regardless of the location of, or path to, the target files.

Some background behind the design of DAFS Completion Groups is provided in Appendix A.

## 2.6 MEMORY MANAGEMENT

DAFS has been designed to take advantage of the fundamental capability of memory-to-memory transports to allow safe Remote DMA operations to Consumer memory that has been registered with the transport provider. Because registered memory is a highly critical resource (both because it requires pinning virtual memory regions, and because it consumes NIC resources to track registered memory regions) the DAFS Provider API provides exposes a memory registration abstraction to the Consumer.

The dap_register_mem() function is used to allow the Provider to be cognizant of the regions of application memory that used to buffer I/O requests and data, and to manage them in appropriately sized chunks, avoiding fragmentation and waste. The Provider will bind the memory to the appropriate transport endpoints as needed, driven by the application's use of the registered memory handles in I/O requests.

A similar function, dap_register_shbuffer(), performs the same action of binding memory to appropriate transport endpoints, but indicates to the Provider that the buffer is in memory shared among cooperating processes. This enables the Provider to optimize resource usage on the host, particularly transport resources consumed in memory registration.

Since so few applications are structured to easily allow the preregistration of buffer memory, the Provider allows a NULL memory handle to be used, indicating that the DAFS API Provider library is expected to register as well as bind the memory on the fly. The Provider **MAY** cache these registrations and bindings, but since the application almost certainly has more detailed knowledge of its buffer usage patterns, this feature should be considered an aid to facilitate ease of porting, and used sparingly.

## 2.7 Authorization and Credentials

The DAFS Provider attempts to handle file migration and server fail-over from the application to the extent possible. Authentication is therefore managed largely behind the scenes, though it may be necessary, for example, to have gotten a Kerberos ticket prior to running the DAFS application. This hiding of servers from the application necessitates that any authentication be managed via callback (registered using dap_auth_callback() and dap_cred_callback()). Since the first contact with a new server may be initiated by an attempt to open a file or directory, and since authentication and a credential handle may be necessary for that operation, the application must obtain the credential handle prior to attempting to open the file or directory, and must have registered callbacks to supply the authentication and credential information before the open is attempted. The Provider will then use the callbacks to obtain the specific information from the application.

## 2.8 File Management

The file namespace of the DAFS Provider is not integrated with the native client filesystem. Thus conventional local files, and non-DAFS mounted remote filesystems are not visible to the DAFS Provider.

The DAFS consumer brings a remote DAFS filesystem into scope implicitly, whenever a file or directory is opened. In addition, the DAFS provider does not maintain a “current working directory” notion, since that presents pitfalls in the face of multi-threaded operation. Rather a directory is either explicitly presented by the DAFS Consumer in the DAFS open file interface, or an absolute path name is used in combination with a NULL directory handle. Thus a DAFS open file is always interpreted relative to a Consumer-specified directory, which also serves to indicate which server an operation is to be performed upon.

## 2.9 File Namespace

The functions specified in the DAFS API access DAFS objects (files, directories, symbolic links, and named attributes) through ’pathnames’, which are interpreted in conjunction with a directory handle. These pathnames are equivalent in structure and format to the pathnames supported on the host platform, and since the pathnames of various host platforms

differs, their exact syntax is outside the scope of the DAFS API. Nevertheless, pathnames have a general structure, which we discuss here.

The namespace is hierarchical, and is rooted in some location that is established by convention. On a UNIX-style host, this might mean that all DAFS directories begin with "/dafs". The components of the hierarchical namespace must be separated by some character, and this too is a matter of host platform convention. On a UNIX-style host, the separator is typically '/' while on a Windows platform '\' is used. Multiple separators are ignored, as are instances of "." between separators. Other sequences between separators are called pathname components, and each is (logically) looked up on the appropriate DAFS server. If a component resolves to a symbolic link, the contents of that link are read, inserted into the pathname being interpreted, and the process continues. If a component is equivalent to ".." then a DAFS_PROC_LOOKUPP protocol message is used to determine where the next component logically resides. The details of how the overall namespace is constructed (or in UNIX-speak, where the mount points reside) are not specified by the DAFS API. This implies that the details of crossing these so-called "mount points" as well as their administration is outside the scope of this document, except that we do note that it is an error to attempt to access non-DAFS files using these interfaces.

As mentioned above, a directory handle is used in conjunction with the pathname to determine which DAFS object is being specified. An absolute pathname is one which begins with the separator character, and if an absolute pathname is given the directory handle may be NULL. Otherwise, the directory handle serves to indicate which DAFS directory is to be used as the root of the pathname, or in other words, where in the hierarchical namespace the processing of the supplied pathname should begin. The DAFS API avoids problems with thread safety by requiring that file objects be specified completely (using the pathname and perhaps directory handle) rather than maintaining any notion of "current location."

## 2.10 File Sharing

The DAFS API fully supports the sharing of files between clients. A sharing key may be supplied when opening a file, so that only applications supplying the matching key will be allowed to open the file. This allows the first instance of an application to prevent access by subsequent "rogue" instances which might corrupt the data. NFS-style access masks may also be supplied, allowing (for example) shared read access but not write access. The removal of a file that is being held open may be denied by supplying the DAP_NO_DELETE option when the file is opened. An application that must initialize data in a shared file prior to making it available to other clients may use the DAP_UNLINKED option to create the file without making it visible in the DAFS file name space. A subsequent call to dap_flink() will then link the existing file handle into the name space so

that it becomes available to other clients. Finally, various types of locking and the delegation of file access authority to clients are available to mediate shared file access and enforce data consistency.

## 2.11 ATTRIBUTE HANDLING

Each file has a set of attributes that may be queried and set. These attributes include such things as size, file type (file, symbolic link, directory), owner, group, access and modification times, and several others. In addition, a file or directory may have arbitrary attributes attached to it. These are known as named attributes. To access the named attributes, the application passes the DAP_NATTR_DIR flag to dap_open_dir() to open the named attribute directory, supplying as the path argument the path to the file whose attributes are being examined. This will return a directory handle that can be passed to dap_async_read_dir() so that the list of named attributes can be examined. In order to read or write a named attribute, dap_open_nattr() is used to obtain a file handle, which can be used to read or write the contents of the named attribute as if it was a regular file. Named attributes can be removed by using the dap_remove() routine.

# APPENDIX A: DESIGN BACKGROUND ON COMPLETION GROUPS

DAFS Completion Groups are motivated by VI Completion Queues, Win32 I/O Completion Ports (IOCP), and the Solaris aiowait(3) interface. The main feature that they all have in common is that the application has a single place where it goes and is informed about “the next” completion operation in a set of “eligible” outstanding operations. All of these mechanisms differ in the manner in which a given “operation” is deemed eligible for one entity versus another. In all cases though, it is unambiguously specified to the I/O provider at I/O initiation time where the Application is to look to block for notification of the completion.

Specifically:

- VI Completion Queues: All operations posted to VI Work Queues bound to a given CQ have their completion notifications placed on the bound CQ.
- Win32 I/O Completion ports: All operations issued on file handles bound to a given I/O Completion Port have their completion notifications placed on the bound IOCP.
- Solaris aiowait(3): All aio operations initiated by the calling process have their completion notifications reported by aiowait(3).

For DAFS Completion groups, our preliminary analysis leads us to believe that Applications will see the best benefit of DAFS if we allow the DAFS Completion Area to be specified on a per I/O basis (as opposed to binding an open file instance to a completion area at file open time, or binding an open server instance to a completion area at server open time).

This is very different than the paradigm of using POSIX aio_suspend(3) or Win32 WaitForMultipleObjects() where each asynchronous I/O operation has some form of async I/O handle specified at I/O initiation time, but when a thread wishes to block waiting for a set of I/Os to complete, the thread specifies the set of handles for the set of I/Os it is interested in. The difficulty with this style of API is that the I/O Provider does not know which APIs will be waited on by a given thread until the thread actually makes a blocking call.

# APPENDIX B: DESIGN BACKGROUND ON ATTRIBUTE HANDLING

Several interfaces hand back variable amounts of information, among them dap_async_read_dir(), dap_async_read_dir2(), and dap_get_attr(). In the case of reading through the contents of a directory, the number of entries may not be known ahead of time, and a set of attributes may be requested with each entry. In the interest of efficiency, it was a design goal to avoid requiring the application to read entries one at a time, as is the case with the POSIX readdir() interface. It is also useful to allow the application to fetch a large number of entries at one time, while avoiding the problems of maintaining consistency in the face of behind-the-scenes Provider data caching.

The method chosen is to allow the application to hand in a buffer to be filled in with the requested information. The application can chose to use a very large buffer in order to grab the entire load of data in one operation, or it can use a smaller buffer and iterate. In the case of iterating through the contents of a directory, a token is returned with each item which acts as an index to allow subsequent items to be retrieved using further calls.

To cope with the the variable-length aspect of the data, fixed-size descriptors are always returned at the beginning of the output buffer. Variable-length items are placed after the fixed-sized descriptors, and pointer fields within the fixed-sized descriptors are fixed up by the Provider to point at the (typically NUL-terminated) variable items. This method allows the application to easily determine how many items have been returned, then to walk that number of fixed-sized descriptors at the beginning of the output buffer, parsing the fields of those descriptors in a natural fashion.

# APPENDIX C: DISCUSSION OF DAFS PROVIDER CACHING

Should the DAFS Provider perform hidden read-ahead and/or write-behind on sequential access patterns? To date, the consensus has been that adding these features increases overhead and complexity without offering benefits sufficient to offset the cost. Maintaining data consistency in the face of behind-the-scenes caching is the major cause of the additional complexity.

If these features were shown to be desireable, the DAFS Provider could autonomously perform reads into a local buffer on the client, copying the data from that buffer when the application requests it. On an application write, the DAFS Provider could autonomously perform a copy into a local buffer on the client then start writing the data to the server, completing the application write (potentially) before the write to the server completes.

Any hidden data caching would require that data consistency be maintained:

- across distinct open instances of the same file within a single Application.
- across processes on a single Node
- across nodes.

Please note: Even if the DAFS provider does not do any hidden operations identified above, we could still introduce APIs that explicitly provide support for application-controlled read-ahead and write-behind.

```
/*
 *  Copyright (C) 2000, 2001  DAFS Collaborative
 *
 *  Direct Access File System Application Programming Interface (DAFS API)
 *  header file:  dafs_api.h
 *
 *  Abstract:
 *
 *      dafs_api.h defines the complete user interface to the
 *      Direct Access Provider.  DAFS API Version 1.0
 */
#ifndef _DAFS_API_H_
#define _DAFS_API_H_

/*
 * Basic type definitions are platform-dependent
 */
#include    "dafs_api_pd.h"


/*
 * Handles
 */
struct dap_cg_handle;
struct dap_mem_handle;
struct dap_cred_handle;
struct dap_file_handle;
typedef struct dap_cg_handle *          DAP_CG_HANDLE;
typedef struct dap_mem_handle *         DAP_MEM_HANDLE;
typedef struct dap_cred_handle *        DAP_CRED_HANDLE;
typedef struct dap_file_handle *        DAP_FILE_HANDLE;
typedef DAP_OPAQUE                      DAP_SHBUFF_KEY;

#define DAP_DIRECTORY_HANDLE            DAP_FILE_HANDLE
#define DAP_HANDLE                      DAP_FILE_HANDLE

#define DAP_NULL_CG_HANDLE              ((DAP_CG_HANDLE) 0UL)
#define DAP_NULL_MEM_HANDLE             ((DAP_MEM_HANDLE) 0UL)
#define DAP_NULL_CRED_HANDLE            ((DAP_CRED_HANDLE) 0UL)
#define DAP_NULL_FILE_HANDLE            ((DAP_FILE_HANDLE) 0UL)
#define DAP_NULL_DIRECTORY_HANDLE       ((DAP_DIRECTORY_HANDLE) 0UL)


/*
 * File + directory management flags
 */
#define DAP_READ        0x000001
#define DAP_WRITE       0x000002
#define DAP_READ_WRITE  0x000003

#define DAP_UNLINKED    0x000004
#define DAP_NATTR_DIR   0x000008
#define DAP_CREATE      0x000010
#define DAP_APPEND      0x000020
#define DAP_TRUNCATE    0x000040
#define DAP_EXCLUSIVE   0x000080
                                    /* Behavioral Hints     */
#define DAP_RANDOM      0x000100
#define DAP_SEQUENTIAL  0x000200
#define DAP_NONBLOCK    0x000400
#define DAP_BUFFERED    0x000800
#define DAP_NO_FOLLOW   0x001000
#define DAP_NO_DELETE   0x002000
#define DAP_SEQ_REVERSE 0x004000
#define DAP_FILESYSTEM  0x008000        /* valid only for fencing */
```

```
                                        /* Extended Sharing     */
#define DAP_SHAREKEY            0x0010000

#define DAP_SHARE_DENY_RD       0x0020000
#define DAP_SHARE_DENY_WR       0x0040000
#define DAP_SHARE_DENY_MASK     (DAP_SHARE_DENY_RD | DAP_SHARE_DENY_WR)
#define DAP_SHARE_DENY_BOTH     (DAP_SHARE_DENY_RD | DAP_SHARE_DENY_WR)


/* DAP_CREATE_MODE bits */
#define DAP_MODE_OTH_X  000001          /* other - X    */
#define DAP_MODE_OTH_W  000002          /* other - W    */
#define DAP_MODE_OTH_R  000004          /* other - R    */
                                        /*              */
#define DAP_MODE_GRP_X  000010          /* group - X    */
#define DAP_MODE_GRP_W  000020          /* group - W    */
#define DAP_MODE_GRP_R  000040          /* group - R    */
                                        /*              */
#define DAP_MODE_OWN_X  000100          /* owner - X    */
#define DAP_MODE_OWN_W  000200          /* owner - W    */
#define DAP_MODE_OWN_R  000400          /* owner - R    */
                                        /*              */
#define DAP_MODE_ISUID  004000          /* setuid       */
#define DAP_MODE_ISGID  002000          /* setgid       */
#define DAP_MODE_ISTXT  001000          /* sticky       */

/*
 * DAFS API Error codes
 */
typedef
enum dap_errors
{
#define DAP_SUCCESS (0U)
        DAP_ERROR_ACCESS = 0x0DAFE000,  /* Direct Access File Errors */
        DAP_ERROR_BADCOOKIE,
        DAP_ERROR_DQUOT,
        DAP_ERROR_FBIG,
        DAP_ERROR_IO,
        DAP_ERROR_LOCKED,
        DAP_ERROR_MLINK,
        DAP_ERROR_NAMETOOLONG,
        DAP_ERROR_NODEV,
        DAP_ERROR_NOTEMPTY,
        DAP_ERROR_NOT_DIRECTORY,
        DAP_ERROR_NXIO,
        DAP_ERROR_SYMLINK,
        DAP_ERROR_WRITE_TOOBIG,
        DAP_ERROR_AUTH_DENIED,
        DAP_ERROR_AUTH_TYPE,
        DAP_ERROR_BAD_ARG,
        DAP_ERROR_BOUND_MEMORY,
        DAP_ERROR_BUFFER_TOO_SMALL,
        DAP_ERROR_CG_INVALID,
        DAP_ERROR_DENIED,
        DAP_ERROR_DIRECTORY,
        DAP_ERROR_FILE_EXISTS,
        DAP_ERROR_INVALID_ACE,
        DAP_ERROR_INVALID_ADDRESS,
        DAP_ERROR_INVALID_ATTR,
        DAP_ERROR_INVALID_CG_HANDLE,
        DAP_ERROR_INVALID_CRED_HANDLE,
        DAP_ERROR_INVALID_CRED_TYPE,
        DAP_ERROR_INVALID_DIR_HANDLE,
        DAP_ERROR_INVALID_FILE_HANDLE,
```

```
        DAP_ERROR_INVALID_FLAGS,
        DAP_ERROR_INVALID_IO_RESULT,
        DAP_ERROR_INVALID_KEY,
        DAP_ERROR_INVALID_MEM_HANDLE,
        DAP_ERROR_INVALID_NATTR,
        DAP_ERROR_IO_CANCELLATION,
        DAP_ERROR_IO_OVERLAP,
        DAP_ERROR_KEY_MISMATCH,
        DAP_ERROR_LEASE_EXPIRED,
        DAP_ERROR_LOCK_BROKEN,
        DAP_ERROR_LOCK_DENIED,
        DAP_ERROR_LOCK_RANGE,
        DAP_ERROR_LOOP,
        DAP_ERROR_NOT_IMPLEMENTED,
        DAP_ERROR_NOT_SUPPORTED,
        DAP_ERROR_NO_AUTH,
        DAP_ERROR_NO_IO_PENDING,
        DAP_ERROR_NO_RESOURCES,
        DAP_ERROR_PATH,
        DAP_ERROR_PENDING_IO,
        DAP_ERROR_PERM,
        DAP_ERROR_TIMED_OUT,
        DAP_ERROR_TRANSPORT_FAILURE,
        DAP_ERROR_UNKNOWN_LOCATION,
        DAP_ERROR_UNKNOWN_PATH,
        DAP_ERROR_UNKNOWN_SERVER,
        DAP_ERROR_UNREACHABLE,
        DAP_ERROR_UNREGISTERED_MEM,
        DAP_ERROR_WOULD_BLOCK
} DAP_ERROR;


/*
 * Descriptors
 */
typedef
struct dap_mem_desc
{                           /* describes an application memory region */
        DAP_MEM_HANDLE      dap_mem_handle;
        DAP_PVOID           dap_bufferp;
        DAP_LENGTH          dap_buffer_len;
} DAP_MEM_DESC;


typedef
struct dap_io_desc
{                           /* describes an I/O completion result */
        DAP_PVOID           dap_lib_private;
        DAP_PVOID           dap_app_private;
        DAP_LENGTH          dap_length;
        DAP_ERROR           dap_error;
} DAP_IO_RESULT;


/*
 * The following two structs are used only by dap_async_listio().
 *
 * DAP_IO_REQUEST is a variable sized structure, dynamically allocated
 * by the application, with the size of the two arrays indicated by the
 * two DAP_COUNTs.
 *
 * Thus the starting address of io_req.mem_chunks[0] in the
 * general case is actually:
 *
 *      (DAP_MEM_DESC)(&io_req.file_chunks[io_req.num_file_chunks]);
```

```
 *
 * and the total size of the structure should be allocated as:
 *
 *      sizeof(DAP_IO_REQUEST)
 *              + sizeof(DAP_FILE_DESC) * (io_req.num_file_chunks - 1)
 *              + sizeof(DAP_MEM_DESC)  * (io_req.num_mem_chunks  - 1)
 *
 * If num_file_chunks == num_mem_chunks == 1 then the structure may
 * be used as-is, and the dap_async_listio() call is equivalent to either
 * dap_async_write() or dap_async_read(), but providing addtional control
 * over certain behavioral details.
 */
typedef
struct dap_file_chunk
{
        DAP_OFFSET          dap_file_offset;
        DAP_COUNT           dap_byte_count;
        DAP_UINT32          dap_cache_hint;
} DAP_FILE_DESC;

/*
 * dap_file_chunk.dap_cache_hint values
 *
 *  This provides the ability to give the server some information
 *  allowing it to better manage its buffer cache and I/O scheduling.
 *  These hints provide weighting information, indicating predictions
 *  about the client's intentions regarding future read and write
 *  accesses to the byte range.  Servers may safely ignore these
 *  hints, and neither the probabilities meant by "maybe" and
 *  "probably" nor what "near future" means are precisely defined.
 */
#define DAP_LIO_READHINT1       0x000001    /* won't read in near future    */
#define DAP_LIO_READHINT2       0x000002    /* probably won't read " " "    */
#define DAP_LIO_READHINT3       0x000003    /* maybe    won't read " " "    */
#define DAP_LIO_READHINT4       0x000004    /* unknown read probability     */
#define DAP_LIO_READHINT5       0x000005    /* maybe  read in near future   */
#define DAP_LIO_READHINT6       0x000006    /* probably will read " " "     */
#define DAP_LIO_READHINT7       0x000007    /* certain  will read " " "     */

#define DAP_LIO_WRITEHINT1      0x000010    /* won't write in near future   */
#define DAP_LIO_WRITEHINT2      0x000020    /* probably won't write " " "   */
#define DAP_LIO_WRITEHINT3      0x000030    /* maybe    won't write " " "   */
#define DAP_LIO_WRITEHINT4      0x000040    /* unknown write probability    */
#define DAP_LIO_WRITEHINT5      0x000050    /* maybe  will write in future  */
#define DAP_LIO_WRITEHINT6      0x000060    /* probably will write " " "    */
#define DAP_LIO_WRITEHINT7      0x000070    /* certain  will write " " "    */

#define DAP_LIO_READMASK        0x000007
#define DAP_LIO_WRITEMASK       0x000070


typedef
struct dap_io_request
{
        DAP_FILE_HANDLE     dap_file_handle;
        DAP_BOOLEAN         dap_write_request; /* else read request */
        DAP_COUNT           dap_num_file_chunks;
        DAP_COUNT           dap_num_mem_chunks;
        DAP_FILE_DESC       dap_file_chunks[ 1 /*dap_num_file_chunks*/ ];
        DAP_MEM_DESC        dap_mem_chunks[  1 /*dap_num_mem_chunks */ ];
} DAP_IO_REQUEST;


/*
 * Timeouts - timeouts are relative (to the current time)
```

```
 */
typedef
struct dap_timeout {
    DAP_UINT64          dap_seconds;
    DAP_UINT32          dap_nseconds;
} DAP_TIMEOUT;

#define DAP_WAIT_NOWAIT_INIT         {  0UL,  0UL }
#define DAP_WAIT_FOREVER_INIT        { ~0UL, ~0UL }

extern const DAP_TIMEOUT dap_nowait_constant;
extern const DAP_TIMEOUT dap_forever_constant;

#define DAP_WAIT_NOWAIT         dap_nowait_constant
#define DAP_WAIT_FOREVER        dap_forever_constant


/*
 * Timestamps
 *
 * Timestamps (as returned by dap_get_attr() for example)
 * are absolute times, expressed as the number of seconds
 * since January 1, 1970 UTC (Universal Coordinated Time).
 * A timestamp with a negative dap_seconds field refers
 * to times before the 0-hour January 1, 1970 UTC.
 */
typedef DAP_TIMEOUT DAP_TIMESTAMP;


/*
 * Authentication and Credentials
 */
typedef enum
dap_auth_type
{
        DAP_AUTH_NONE,
        DAP_AUTH_TEXT,
        DAP_AUTH_GSS,
        DAP_AUTH_DEFAULT
} DAP_AUTH_TYPE;

#define DAP_AUTH_NONE_MASK      (1UL << DAP_AUTH_NONE)
#define DAP_AUTH_TEXT_MASK      (1UL << DAP_AUTH_TEXT)
#define DAP_AUTH_GSS_MASK       (1UL << DAP_AUTH_GSS)
#define DAP_AUTH_DEFAULT_MASK   (1UL << DAP_AUTH_DEFAULT)

typedef struct dap_auth_text_data {
        DAP_CHAR *dap_auth_id;
        DAP_CHAR *dap_auth_password;
} DAP_AUTH_TEXT_DATA;

typedef void * DAP_AUTH_DATA;

typedef enum
dap_cred_type
{
        DAP_CRED_NAME,          /* "whoami@some.domain"         */
        DAP_CRED_ID,            /*  uid, gid, groups[]          */
        DAP_CRED_GSS,           /* structure as per GSS         */
        DAP_CRED_DEFAULT        /* void                         */
} DAP_CRED_TYPE;

typedef void DAP_CRED_DATA;     /* generic inputs               */

typedef DAP_UINT64  DAP_SHARE_KEY;
```

```
/*
 * ACLs
 */
typedef
struct dap_acl_info
{
    int         dap_type;
    int         dap_flag;
    int         dap_access_mask;
    DAP_CHAR    *dap_who;   /* user, a la dap_group and dap_user */
}   DAP_ACL_INFO;

/* dap_type bits */
#define DAP_ACL_ALLOW                   0x00000001
#define DAP_ACL_DENY                    0x00000002
#define DAP_ACL_AUDIT                   0x00000004
#define DAP_ACL_ALARM                   0x00000008

/* dap_flag bits */
#define DAP_ACL_INHERIT_FILE            0x00000001
#define DAP_ACL_INHERIT_DIR             0x00000002
#define DAP_ACL_NO_PROPAGATE            0x00000004
#define DAP_ACL_INHERIT_ONLY            0x00000008
#define DAP_ACL_SUCCESSFUL_ACC          0x00000010
#define DAP_ACL_FAILED_ACC              0x00000020
#define DAP_ACL_ID_GROUP                0x00000040

/* dap_access_mask bits */
#define DAP_ACL_READ_DATA               0x00000001
#define DAP_ACL_LIST_DIR                0x00000001
#define DAP_ACL_WRITE_DATA              0x00000002
#define DAP_ACL_ADD_FILE                0x00000002
#define DAP_ACL_APPEND_DATA             0x00000004
#define DAP_ACL_ADD_SUBDIR              0x00000004
#define DAP_ACL_READ_NAMED_ATTRS        0x00000008
#define DAP_ACL_WRITE_NAMED_ATTRS       0x00000010
#define DAP_ACL_EXECUTE                 0x00000020
#define DAP_ACL_DELETE_CHILD            0x00000040
#define DAP_ACL_READ_ATTRS              0x00000080
#define DAP_ACL_WRITE_ATTRS             0x00000100
                                        /* ...  */
#define DAP_ACL_DELETE                  0x00010000
#define DAP_ACL_READ_ACL                0x00020000
#define DAP_ACL_WRITE_ACL               0x00040000
#define DAP_ACL_WROTE_OWNER             0x00080000
#define DAP_ACL_SYNC                    0x00100000


/*
 * dap_who values will normally be of the form  "user@dns_domain".
 *
 * In addition, there are several distinguished values having
 * special meanings:
 *
 *  "OWNER@"            - the owner of the file
 *  "GROUP@"            - the group associated with the file
 *  "EVERYONE@"         - the world
 *  "INTERACTIVE@"      - access from an interactive terminal
 *  "NETWORK@"          - accessed via the network
 *  "DIALUP@"           - accessed as a dialup user
 *  "BATCH@"            - accessed from a batch job
 *  "ANONYMOUS@"        - unauthenticated
 *  "AUTHENTICATED@"    - the opposite of ANONYMOUS@
```

```
 *  "SERVICE@"          - accessed from a system service
 */


/*
 * Fencing
 *
 * Fence lists are arbitrary strings, indicating those clients who are
 * allowed to access a file or file system, and are used by cooperating
 * processes to implement client access revocation.
 */

typedef enum
dap_fencelist_update
{
        DAP_FENCE_REPLACE,  /* new list         */
        DAP_FENCE_APPEND,   /* add to list      */
        DAP_FENCE_REMOVE    /* remove from list */
} DAP_FENCELIST_UPDATE;


/*
 * Attributes of file system objects
 *
 *  The bitmap valid_attrs indicates what portion of the remaining
 *  contents are valid.
 */
#define DAP_STAT_OBJECT_TYPE        (1ULL <<  0) /* mandatory */
#define DAP_STAT_CHANGE             (1ULL <<  1) /* mandatory */
#define DAP_STAT_OBJECT_SIZE        (1ULL <<  2) /* mandatory */
#define DAP_STAT_NAMED_ATTR         (1ULL <<  3)
/* #define DAP_STAT_ACL             (1ULL <<  4) not via set_attr */
#define DAP_STAT_ARCHIVE            (1ULL <<  5)
#define DAP_STAT_FILE_HANDLE        (1ULL <<  6)
#define DAP_STAT_FILE_ID            (1ULL <<  7) /* mandatory */
#define DAP_STAT_HIDDEN             (1ULL <<  8)
#define DAP_STAT_MIME_TYPE          (1ULL <<  9)
#define DAP_STAT_MODE               (1ULL << 10)
#define DAP_STAT_NUM_LINKS          (1ULL << 11)
#define DAP_STAT_OWNER              (1ULL << 12)
#define DAP_STAT_OWNER_GROUP        (1ULL << 13)
#define DAP_STAT_SYSTEM             (1ULL << 14)
#define DAP_STAT_TIME_ACCESS        (1ULL << 15)
#define DAP_STAT_TIME_ACCESS_SET    (1ULL << 16)
#define DAP_STAT_TIME_BACKUP        (1ULL << 17)
#define DAP_STAT_TIME_CREATE        (1ULL << 18)
#define DAP_STAT_TIME_DELTA         (1ULL << 19)
#define DAP_STAT_TIME_METADATA      (1ULL << 20)
#define DAP_STAT_TIME_MODIFY        (1ULL << 21)
#define DAP_STAT_TIME_MODIFY_SET    (1ULL << 22)
#define DAP_STAT_SPACE_USED         (1ULL << 23)
#define DAP_STAT_RAW_DEVICE         (1ULL << 24)


typedef enum
dap_filetype
{
        DAP_NONE,
        DAP_FILE,
        DAP_DIR,
        DAP_BLOCK_DEV,
        DAP_CHAR_DEV,
        DAP_SYMLINK,
        DAP_SOCKET,
        DAP_FIFO,
```

```
        DAP_ATTR_DIR,
        DAP_NATTR
} DAP_FILETYPE;

typedef         /* Internal FS handle, useful to detect FS crossings */
struct dap_fshandle
{
    DAP_OPAQUE dap_filesys_hdl[2];
}   DAP_FSHANDLE;


/* additional file type information */
typedef
struct dap_specdata
{
        DAP_UINT64          specdata1;
        DAP_UINT64          specdata2;
} DAP_SPECDATA;


typedef
struct dap_stat_desc
{
        DAP_BITMAP          dap_valid_attrs;

        DAP_BITMAP          dap_change;
        DAP_OPAQUE          dap_file_id;
        DAP_LENGTH          dap_object_size;
        DAP_LENGTH          dap_space_used;
        DAP_CREATE_MODE     dap_mode;
        DAP_COUNT           dap_num_links;
        DAP_TIMESTAMP       dap_access_time;
        DAP_TIMESTAMP       dap_access_set_time;
        DAP_TIMESTAMP       dap_backup_time;
        DAP_TIMESTAMP       dap_create_time;
        DAP_TIMESTAMP       dap_delta_time;
        DAP_TIMESTAMP       dap_metadata_time;
        DAP_TIMESTAMP       dap_modify_time;
        DAP_TIMESTAMP       dap_modify_set_time;
        DAP_SPECDATA        dap_raw_device;
        DAP_FILETYPE        dap_object_type;
        DAP_BOOLEAN         dap_is_hidden;
        DAP_BOOLEAN         dap_is_system;
        DAP_BOOLEAN         dap_is_named_attrs;
        DAP_BOOLEAN         dap_is_archive;
        DAP_CHAR            *dap_mimetype;
        DAP_CHAR            *dap_owner;
        DAP_CHAR            *dap_owner_group;
        DAP_FSHANDLE        dap_filesys_handle;
} DAP_STAT_DESC;


/*
 * Directory information
 */
typedef
struct dap_direntry
{
        DAP_FILETYPE    dap_direntry_type;
        DAP_OFFSET      dap_direntry_cookie;
        DAP_CHAR        *dap_direntry_name;
        DAP_STAT_DESC   *dap_direntry_attrp;
} DAP_DIRENTRY;

typedef
```

```
struct dap_readdir_result {
        DAP_OPAQUE      dap_cookiev;
        DAP_BOOLEAN     dap_end_flag;
        DAP_COUNT       dap_num_entries;
        DAP_DIRENTRY    dap_entry[1];       /* Variable-sized */
} DAP_READDIR_RESULT;


/*
 * Attributes of file systems
 *
 *  The bitmap dap_valid_attrs indicates what portion of the remaining
 *  contents are valid.
 */
#define DAP_FSSTAT_LINK_SUPPORT         (1ULL <<  0) /* mandatory */
#define DAP_FSSTAT_SYMLINK_SUPPORT      (1ULL <<  1) /* mandatory */
#define DAP_FSSTAT_CAN_SET_TIME         (1ULL <<  2)
#define DAP_FSSTAT_CASE_INSENSITIVE     (1ULL <<  3)
#define DAP_FSSTAT_CASE_PRESERVING      (1ULL <<  4)
#define DAP_FSSTAT_CHOWN_RESTRICTED     (1ULL <<  5)
#define DAP_FSSTAT_HOMOGENEOUS          (1ULL <<  6)
#define DAP_FSSTAT_NO_TRUNC             (1ULL <<  7)
#define DAP_FSSTAT_UNIQUE_HANDLE        (1ULL <<  8)
#define DAP_FSSTAT_LEASE_TIME           (1ULL <<  9) /* mandatory */
#define DAP_FSSTAT_RD_ATTR_SUPPORT      (1ULL << 10) /* mandatory */
#define DAP_FSSTAT_ACL_SUPPORT          (1ULL << 11)
#define DAP_FSSTAT_MAX_LINK             (1ULL << 12)
#define DAP_FSSTAT_MAX_NAME             (1ULL << 13)
#define DAP_FSSTAT_SUPPORTED_FATTRS     (1ULL << 14) /* mandatory */
#define DAP_FSSTAT_SUPPORTED_FSATTRS    (1ULL << 15) /* mandatory */
#define DAP_FSSTAT_FILES_AVAILABLE      (1ULL << 16)
#define DAP_FSSTAT_FILES_FREE           (1ULL << 17)
#define DAP_FSSTAT_FILES_TOTAL          (1ULL << 18)
#define DAP_FSSTAT_MAX_FILE_SIZE        (1ULL << 19)
#define DAP_FSSTAT_MAX_READ             (1ULL << 20)
#define DAP_FSSTAT_MAX_WRITE            (1ULL << 21)
#define DAP_FSSTAT_QUOTA_HARD           (1ULL << 22)
#define DAP_FSSTAT_QUOTA_SOFT           (1ULL << 23)
#define DAP_FSSTAT_QUOTA_USED           (1ULL << 24)
#define DAP_FSSTAT_SPACE_AVAIL          (1ULL << 25)
#define DAP_FSSTAT_SPACE_FREE           (1ULL << 26)
#define DAP_FSSTAT_SPACE_TOTAL          (1ULL << 27)
#define DAP_FSSTAT_FS_HANDLE            (1ULL << 28) /* mandatory */
#define DAP_FSSTAT_MAX_APPEND           (1ULL << 29) /* mandatory */
#define DAP_FSSTAT_PREF_IO_SIZE         (1ULL << 30)
#define DAP_FSSTAT_SPACE_USED           (1ULL << 31)
/*
 * failover_locations and new_locations are deliberately omitted,
 * being reserved for provider implementations, not exposed to apps.
 */

/*
 * File system behavioral attributes
 */
typedef
struct dap_filesys_desc
{
        DAP_BITMAP          dap_valid_attrs;

        DAP_BOOLEAN         dap_link_support;
        DAP_BOOLEAN         dap_symlink_support;
        DAP_BOOLEAN         dap_can_set_time;
        DAP_BOOLEAN         dap_ignores_case;
        DAP_BOOLEAN         dap_preserves_case;
        DAP_BOOLEAN         dap_chown_restricted;
```

```
        DAP_BOOLEAN         dap_homogeneous;
        DAP_BOOLEAN         dap_no_truncate;
        DAP_BOOLEAN         dap_unique_handle;
        DAP_UINT32          dap_lease_time;
        DAP_UINT32          dap_rd_attr_error;
        DAP_UINT32          dap_acl_support;
        DAP_UINT32          dap_max_links;
        DAP_UINT32          dap_max_name;
        DAP_BITMAP          dap_supported_attrs;
        DAP_BITMAP          dap_supported_fs_attrs;
        DAP_UINT64          dap_files_available;
        DAP_UINT64          dap_files_free;
        DAP_UINT64          dap_files_total;
        DAP_UINT64          dap_max_file_size;
        DAP_UINT64          dap_max_read;
        DAP_UINT64          dap_max_write;
        DAP_UINT64          dap_quota_hard;
        DAP_UINT64          dap_quota_soft;
        DAP_UINT64          dap_quota_used;
        DAP_UINT64          dap_space_avail;
        DAP_UINT64          dap_space_free;
        DAP_UINT64          dap_space_total;
        DAP_UINT64          dap_space_used;
        DAP_UINT64          dap_max_append;
        DAP_UINT64          dap_prefer_iosize;
        DAP_FSHANDLE        dap_filesys_handle;
} DAP_FILESYS_DESC;

/*
 * Locking
 */
typedef enum
dap_lock_type
{
        DAP_LOCK_TRY_READ,              /* Read Lock    */
        DAP_LOCK_TRY_WRITE,             /* Write Lock   */
        DAP_LOCK_READ,                  /* Blocking RL  */
        DAP_LOCK_WRITE,                 /* Blocking WL  */
        DAP_LOCK_ABORT                  /* roll back    */
} DAP_LOCK_TYPE;

#define DAP_LOCK_OPT_PERSIST    0x01    /* Persist      */
#define DAP_LOCK_OPT_AUTOREC    0x02    /* AutoRecover  */
#define DAP_LOCK_OPT_REPAIR     0x04    /* if broken    */


/*
 *  Function prototypes
 */

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Authentication and Credentials
 */
typedef unsigned int
        (*DAP_AUTH_FUNC) (
                        void *            /* context */ ,
                        unsigned int      /* opcode */ ,
                        DAP_AUTH_TYPE     /* auth_type */ ,
                        DAP_LENGTH      * /* auth_size */ ,
                        DAP_AUTH_DATA     /* auth_data */ );
typedef unsigned int
```

```
        (*DAP_CRED_FUNC) (
                        void *            /* context */ ,
                        DAP_CRED_HANDLE   /* cred_handle */ ,
                        unsigned int      /* opcode */ ,
                        DAP_CRED_TYPE   * /* cred_type */ ,
                        DAP_LENGTH      * /* cred_data_len */ ,
                        DAP_CRED_DATA   * /* cred_data */ );

extern
DAP_ERROR
dap_auth_callback(
                unsigned int              /* auth_type_mask */ ,
                void *                    /* handler_context */ ,
                DAP_AUTH_FUNC             /* dap_auth_handler */ );

extern
DAP_ERROR
dap_create_credential(
                void                    * /* handler_context */ ,
                DAP_CRED_HANDLE         * /* cred_handle */ );

extern
DAP_ERROR
dap_destroy_credential(
                DAP_CRED_HANDLE           /* cred_handle */ );

extern
DAP_ERROR
dap_cred_callback(
                DAP_CRED_FUNC   dap_cred_handler);

/*
 * Memory Management
 */
extern
DAP_ERROR
dap_register_mem(
                DAP_PVOID                 /* buffer */ ,
                DAP_LENGTH                /* length */ ,
                DAP_MEM_HANDLE          * /* mem_handle */ );

extern
DAP_ERROR
dap_register_shbuffer(
                DAP_PVOID                 /* buffer */ ,
                DAP_LENGTH                /* length */ ,
                DAP_SHBUFF_KEY            /* buff_key */ ,
                DAP_FLAGS                 /* flags */ ,
                DAP_MEM_HANDLE          * /* mem_handle */ );

extern
DAP_ERROR
dap_deregister_mem(
                DAP_MEM_HANDLE            /* mem_handle */ );


/*
 * Completion Group Management
 */
extern
DAP_ERROR
dap_create_cg(
                DAP_CG_HANDLE           * /* cg_handle */ ,
                DAP_COUNT                 /* cg_entries */ );
```

```
extern
DAP_ERROR
dap_destroy_cg(
                DAP_CG_HANDLE             /* cg_handle */ );


/*
 * File and Directory Management
 */
extern
DAP_ERROR
dap_open_file(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                const DAP_CHAR          * /* path */ ,
                DAP_FLAGS                 /* flags */ ,
                DAP_CREATE_MODE           /* mode */ ,
                DAP_SHARE_KEY             /* share_key */ ,
                DAP_FILE_HANDLE         * /* file_handle */);

extern
DAP_ERROR
dap_close_file(
                DAP_FILE_HANDLE           /* file_handle */ );

extern
DAP_ERROR
dap_make_dev(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                const DAP_CHAR          * /* path */ ,
                DAP_FILETYPE              /* type */ ,
                DAP_CREATE_MODE           /* mode */ ,
                const DAP_SPECDATA      * /* spec_data */ );


/*
 * Data Transfer and Completion
 */
extern
DAP_ERROR
dap_async_read(
                DAP_FILE_HANDLE           /* file_handle */ ,
                DAP_OFFSET                /* file_offset */ ,
                DAP_COUNT                 /* io_count */ ,
                const DAP_MEM_DESC      * /* mem_desc */ ,
                DAP_CG_HANDLE             /* cg_handle */ ,
                DAP_IO_RESULT           * /* io_desc */ );

extern
DAP_ERROR
dap_async_write(
                DAP_FILE_HANDLE           /* file_handle */ ,
                DAP_OFFSET                /* file_offset */ ,
                DAP_COUNT                 /* io_count */ ,
                const DAP_MEM_DESC      * /* mem_desc */ ,
                DAP_CG_HANDLE             /* cg_handle */ ,
                DAP_IO_RESULT           * /* io_desc */ );

extern
DAP_ERROR
dap_io_done(
                DAP_IO_RESULT           * /* io_desc */ );

extern
```

```
DAP_ERROR
dap_io_wait(
                DAP_TIMEOUT              /* timeout */ ,
                DAP_IO_RESULT           * /* io_desc */ );

extern
DAP_ERROR
dap_cg_done(
                DAP_CG_HANDLE            /* cg_handle */ ,
                DAP_IO_RESULT           ** /* io_desc */ );

extern
DAP_ERROR
dap_cg_batchwait(
                DAP_CG_HANDLE            /* cg_handle */ ,
                DAP_TIMEOUT              /* timeout */ ,
                DAP_COUNT               * /* n_results */ ,
                DAP_IO_RESULT           ** /* io_desc */ );

extern
DAP_ERROR
dap_cg_wait(
                DAP_CG_HANDLE            /* cg_handle */ ,
                DAP_TIMEOUT              /* timeout */ ,
                DAP_IO_RESULT           ** /* io_desc */ );

extern
DAP_ERROR
dap_expedite(
                DAP_IO_RESULT           * /* io_desc */ );

extern
DAP_ERROR
dap_cancel_async_op(
                DAP_IO_RESULT           * /* io_desc */ );

extern
DAP_ERROR
dap_read(
                DAP_FILE_HANDLE          /* file_handle */ ,
                DAP_OFFSET               /* file_offset */ ,
                DAP_COUNT                /* io_count */ ,
                const DAP_MEM_DESC      * /* mem_desc */ ,
                DAP_LENGTH              * /* done_count */ );

extern
DAP_ERROR
dap_write(
                DAP_FILE_HANDLE          /* file_handle */ ,
                DAP_OFFSET               /* file_offset */ ,
                DAP_COUNT                /* io_count */ ,
                const DAP_MEM_DESC      * /* mem_desc */ ,
                DAP_LENGTH              * /* done_count */ );

extern
DAP_ERROR
dap_async_listio(
                DAP_COUNT                /* io_count */ ,
                DAP_IO_REQUEST * const [] /* io_requests */ ,
                DAP_CG_HANDLE            /* cg_handle */ ,
                DAP_IO_RESULT  * const [] /* io_descs */ ,
                DAP_UINT32               /* usec_window */ ,
                DAP_UINT32               /* num_completions */  );
```

```
/*
 * Directory and File Management Operations
 */
extern
DAP_ERROR
dap_open_dir(
                DAP_DIRECTORY_HANDLE      /* base_dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                const DAP_CHAR          * /* path */ ,
                DAP_FLAGS                 /* flags */ ,
                DAP_CREATE_MODE           /* dap_mode */ ,
                DAP_DIRECTORY_HANDLE    * /* dir_handle */ );

extern
DAP_ERROR
dap_close_dir(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ );

extern
DAP_ERROR
dap_async_read_dir(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_OFFSET                /* cookie */ ,
                DAP_OPAQUE                /* cookie_verifier */ ,
                DAP_MEM_HANDLE            /* mem_handle */ ,
                DAP_LENGTH                /* size */ ,
                DAP_READDIR_RESULT      * /* resultp */ ,
                DAP_CG_HANDLE             /* cg_handle */ ,
                DAP_IO_RESULT           * /* io_desc */  );

extern
DAP_ERROR
dap_async_read_dir2(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_OFFSET                /* cookie */ ,
                DAP_OPAQUE                /* cookie_verifier */ ,
                DAP_BITMAP                /* attrs_requested */ ,
                DAP_MEM_HANDLE            /* mem_handle */ ,
                DAP_LENGTH                /* size */ ,
                DAP_READDIR_RESULT      * /* resultp */ ,
                DAP_CG_HANDLE             /* cg_handle */ ,
                DAP_IO_RESULT           * /* io_desc */  );

extern
DAP_ERROR
dap_read_dir(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_OFFSET                /* cookie */ ,
                DAP_OPAQUE                /* cookie_verifier */ ,
                DAP_MEM_HANDLE            /* mem_handle */ ,
                DAP_LENGTH                /* size */ ,
                DAP_READDIR_RESULT      * /* resultp */ );

extern
DAP_ERROR
dap_read_dir2(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_OFFSET                /* cookie */ ,
                DAP_OPAQUE                /* cookie_verifier */ ,
                DAP_BITMAP                /* attrs_requested */ ,
                DAP_MEM_HANDLE            /* mem_handle */ ,
                DAP_LENGTH                /* size */ ,
                DAP_READDIR_RESULT      * /* resultp */ );

extern
```

```
DAP_ERROR
dap_remove(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                const DAP_CHAR          * /* path */ );

extern
DAP_ERROR
dap_rename(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                const DAP_CHAR          * /* old_path */ ,
                const DAP_CHAR          * /* new_path */ );

extern
DAP_ERROR
dap_link(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                const DAP_CHAR          * /* old_path */ ,
                const DAP_CHAR          * /* new_name */ );

extern
DAP_ERROR
dap_flink(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                DAP_FILE_HANDLE           /* old_file */ ,
                const DAP_CHAR          * /* new_path */ );

extern
DAP_ERROR
dap_symlink(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                const DAP_CHAR          * /* old_path */ ,
                const DAP_CHAR          * /* new_path */ );

extern
DAP_ERROR
dap_read_link(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                const DAP_CHAR          * /* path */ ,
                DAP_COUNT                 /* buffer_size */ ,
                DAP_CHAR                * /* buffer */ );

extern
DAP_ERROR
dap_get_attr(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                const DAP_CHAR          * /* path */ ,
                DAP_FLAGS                 /* flags */ ,
                DAP_BITMAP                /* attrs_requested */ ,
                DAP_COUNT                 /* max_byte_count */ ,
                DAP_STAT_DESC           * /* descr_ptr */ );

extern
DAP_ERROR
dap_get_fattr(
                DAP_HANDLE                /* some_handle */ ,
                DAP_BITMAP                /* attrs_requested */ ,
                DAP_COUNT                 /* max_byte_count */ ,
                DAP_STAT_DESC           * /* descr_ptr */ );
```

```
extern
DAP_ERROR
dap_set_attr(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                const DAP_CHAR          * /* path */ ,
                DAP_FLAGS                 /* flags */ ,
                DAP_STAT_DESC           * /* descr_ptr */ ,
                DAP_BITMAP              * /* attrs_changed */ );

extern
DAP_ERROR
dap_set_fattr(
                DAP_HANDLE                /* some_handle */ ,
                DAP_STAT_DESC           * /* descr_ptr */ ,
                DAP_BITMAP              * /* attrs_changed */ );

extern
DAP_ERROR
dap_chmod(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                const DAP_CHAR          * /* path */ ,
                DAP_CREATE_MODE           /* mode */  );
extern
DAP_ERROR
dap_chown(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                const DAP_CHAR          * /* path */ ,
                const DAP_CHAR          * /* owner */ ,
                const DAP_CHAR          * /* group  */ );

extern
DAP_ERROR
dap_fchmod(
                DAP_HANDLE                /* some_handle */ ,
                DAP_CREATE_MODE           /* mode */ );

extern
DAP_ERROR
dap_fchown(
                DAP_HANDLE                /* some_handle */ ,
                const DAP_CHAR          * /* owner */ ,
                const DAP_CHAR          * /* group */ );

extern
DAP_ERROR
dap_fsync(
                DAP_FILE_HANDLE           /* file_handle */ );

extern
DAP_ERROR
dap_open_nattr(
                DAP_DIRECTORY_HANDLE      /* nattr_dir_handle */ ,
                DAP_CRED_HANDLE           /* cred_handle */ ,
                const DAP_CHAR          * /* attr_name */ ,
                DAP_FLAGS                 /* flags */ ,
                DAP_FILE_HANDLE         * /* file_handle */ );

extern
DAP_ERROR
dap_filesys_query(
                DAP_DIRECTORY_HANDLE      /* dir_handle */ ,
```

```
                DAP_CRED_HANDLE          /* cred_handle */ ,
                const DAP_CHAR         * /* path */ ,
                DAP_BITMAP               /* attrs_requested */ ,
                DAP_COUNT                /* max_byte_count */ ,
                DAP_FILESYS_DESC       * /* filesys_info */ );

extern
DAP_ERROR
dap_lock_range(
                DAP_FILE_HANDLE          /* file_handle */ ,
                DAP_OFFSET               /* byte_offset */ ,
                DAP_LENGTH               /* byte_length */ ,
                DAP_LOCK_TYPE            /* lock_type   */ ,
                unsigned int             /* lock_options */ ,
                DAP_TIMEOUT              /* how_long    */ );

extern
DAP_ERROR
dap_unlock_range(
                DAP_FILE_HANDLE          /* file_handle */ ,
                DAP_OFFSET               /* byte_offset */ ,
                DAP_LENGTH               /* byte_length */ );

extern
DAP_ERROR
dap_get_acl(
                DAP_DIRECTORY_HANDLE     /* dir_handle */ ,
                DAP_CRED_HANDLE          /* cred_handle */ ,
                const DAP_CHAR         * /* path */ ,
                DAP_FLAGS                /* flags */ ,
                DAP_COUNT                /* max_byte_count */ ,
                DAP_ACL_INFO           * /* aces_ptr */ ,
                DAP_COUNT              * /* num_aces */ );

extern
DAP_ERROR
dap_set_acl(
                DAP_DIRECTORY_HANDLE     /* dir_handle */ ,
                DAP_CRED_HANDLE          /* cred_handle */ ,
                const DAP_CHAR         * /* path */ ,
                DAP_FLAGS                /* flags */ ,
                DAP_COUNT                /* num_aces */ ,
                const DAP_ACL_INFO     * /* aces_ptr */ );

extern
DAP_ERROR
dap_set_fenceID(
                const DAP_CHAR         * /* fence_ID */ );

extern
DAP_ERROR
dap_set_fencelist(
                DAP_DIRECTORY_HANDLE     /* dir_handle */ ,
                DAP_CRED_HANDLE          /* cred_handle */ ,
                const DAP_CHAR         * /* path */ ,
                DAP_FLAGS                /* flags */ ,
                DAP_FENCELIST_UPDATE     /* action */ ,
                DAP_COUNT                /* num_fence_ids */ ,
                DAP_CHAR * const         /* fence_ids_ptr */ [] );

extern
DAP_ERROR
dap_get_fencelist(
                DAP_DIRECTORY_HANDLE     /* dir_handle */ ,
                DAP_CRED_HANDLE          /* cred_handle */ ,
```

```
                const DAP_CHAR          * /* path */ ,
                DAP_FLAGS                 /* flags */ ,
                DAP_COUNT                 /* max_byte_count */ ,
                DAP_CHAR                * /* fence_ids_ptr */ [],
                DAP_COUNT               * /* num_fence_ids */ );

extern
const char *
dap_strerror(
                DAP_ERROR                 /* error_code */ );


/*
 * Vendor-specific interface for extended functionality
 *
 *  The single pre-defined extension is one that returns the
 *  provider name as a NUL-terminated string and the major
 *  and minor version number.  All providers are encouraged
 *  to support this query, and must support the interface.
 */
typedef struct
dap_ext_version {
#   define          DAP_EXT_GETVERSION  0x0001
    unsigned int    dap_major;
    unsigned int    dap_minor;
    char            dap_provider[80];
} DAP_EXT_VERSION;

extern
DAP_ERROR
dap_extensions(
                unsigned int              /* request_token */ ,
                DAP_PVOID                 /* argument_ptr */ ,
                DAP_COUNT                 /* argument_size */ );

#ifdef  __cplusplus
}
#endif

#endif /*_DAFS_API_H_*/
```

```
/*
 *  Copyright (C) 2000, 2001  DAFS Collaborative
 *
 *  Direct Access File System Application Programming Interface (DAFS API)
 *  sample header file:  dafs_api_pd.h
 *
 *  Abstract:
 *
 *      Platform-dependent type definitions - an example.
 *
 *      N.B.: This file contains sample definitions, and is not
 *      necessarily correct for any particular hardware/software
 *      combination.
 */
#ifndef _DAFS_API_PD_H_
#define _DAFS_API_PD_H_

/*
 * Fundamental types
 */
typedef char                    DAP_CHAR;
typedef unsigned char           DAP_UCHAR;
typedef unsigned int            DAP_BOOLEAN;
typedef unsigned int            DAP_COUNT;
typedef unsigned long long      DAP_LENGTH;
typedef unsigned long long      DAP_OFFSET;
typedef unsigned long long      DAP_OPAQUE;
typedef unsigned long long      DAP_BITMAP;
typedef void *                  DAP_PVOID;

typedef unsigned long           DAP_CREATE_MODE;
typedef unsigned long           DAP_FLAGS;


/*
 * Machine-Dependent types for a Generic Nondescript Platform
 */
typedef unsigned long long      DAP_UINT64;
typedef unsigned long           DAP_UINT32;
typedef unsigned short          DAP_UINT16;
typedef unsigned char           DAP_UINT8;

#endif  /* _DAFS_API_PD_H_ */
```

```
/*
 *  Copyright (C) 2000, 2001  DAFS Collaborative
 *
 *  Direct Access File System Application Programming Interface (DAFS API)
 *  sample header file:  dafs_api_pd.h  --  BSD version
 *
 *  Abstract:
 *
 *      Platform-dependent type definitions - an example.
 *
 *      N.B.: This file contains sample definitions, and is not
 *      necessarily correct for any particular hardware/software
 *      combination.
 */
#ifndef _DAFS_API_PD_H_
#define _DAFS_API_PD_H_

/*
 * Fundamental types
 */
typedef char                    DAP_CHAR;
typedef unsigned char           DAP_UCHAR;
typedef unsigned int            DAP_BOOLEAN;
typedef unsigned int            DAP_COUNT;
typedef unsigned long long      DAP_LENGTH;
typedef unsigned long long      DAP_OFFSET;
typedef unsigned long long      DAP_OPAQUE;
typedef unsigned long long      DAP_BITMAP;
typedef void *                  DAP_PVOID;

typedef unsigned long           DAP_CREATE_MODE;
typedef unsigned long           DAP_FLAGS;


/*
 * Machine-Dependent types for BSDI/OS 4.0 on Intel32
 */
#include <sys/types.h>
typedef u_int64_t               DAP_UINT64;
typedef u_int32_t               DAP_UINT32;
typedef u_int16_t               DAP_UINT16;
typedef u_int8_t                DAP_UINT8;

typedef union {
        DAP_UINT64              AddressBits;  /* only the low 32 bits used */
        DAP_PVOID               Address;
} DAP_PVOID64;


#endif  /* _DAFS_API_PD_H_ */
```

```
/*
 *  Copyright (C) 2000, 2001  DAFS Collaborative
 *
 *  Direct Access File System Application Programming Interface (DAFS API)
 *  sample header file:  dafs_api_pd.h  --  Linux version
 *
 *  Abstract:
 *
 *      Platform-dependent type definitions - an example.
 *
 *      N.B.: This file contains sample definitions, and is not
 *      necessarily correct for any particular hardware/software
 *      combination.
 */
#ifndef _DAFS_API_PD_H_
#define _DAFS_API_PD_H_

/*
 * Fundamental types
 */
typedef char                    DAP_CHAR;
typedef unsigned char           DAP_UCHAR;
typedef unsigned int            DAP_BOOLEAN;
typedef unsigned int            DAP_COUNT;
typedef unsigned long long      DAP_LENGTH;
typedef unsigned long long      DAP_OFFSET;
typedef unsigned long long      DAP_OPAQUE;
typedef unsigned long long      DAP_BITMAP;
typedef void *                  DAP_PVOID;

typedef unsigned long           DAP_CREATE_MODE;
typedef unsigned long           DAP_FLAGS;


/*
 * Machine-Dependent types for Linux on x86
 */
#include <asm/types.h>
typedef __u64                   DAP_UINT64;
typedef __u32                   DAP_UINT32;
typedef __u16                   DAP_UINT16;
typedef __u8                    DAP_UINT8;

typedef union {
        DAP_UINT64              AddressBits;
        DAP_PVOID               Address;
} DAP_PVOID64;


#endif  /* _DAFS_API_PD_H_ */
```

```
/*
 *  Copyright (C) 2000, 2001  DAFS Collaborative
 *
 *  Direct Access File System Application Programming Interface (DAFS API)
 *  sample header file:  dafs_api_pd.h  --  Solaris version
 *
 *  Abstract:
 *
 *      Platform-dependent type definitions - an example.
 *
 *      N.B.: This file contains sample definitions, and is not
 *      necessarily correct for any particular hardware/software
 *      combination.
 */
#ifndef _DAFS_API_PD_H_
#define _DAFS_API_PD_H_

/*
 * Fundamental types
 */
typedef char                    DAP_CHAR;
typedef unsigned char           DAP_UCHAR;
typedef unsigned int            DAP_BOOLEAN;
typedef unsigned int            DAP_COUNT;
typedef unsigned long long      DAP_LENGTH;
typedef unsigned long long      DAP_OFFSET;
typedef unsigned long long      DAP_OPAQUE;
typedef unsigned long long      DAP_BITMAP;
typedef void *                  DAP_PVOID;

typedef unsigned long           DAP_CREATE_MODE;
typedef unsigned long           DAP_FLAGS;


/*
 * Machine-Dependent types for Solaris on Sparc
 */
#include <sys/types.h>
typedef uint64_t                DAP_UINT64;
typedef uint32_t                DAP_UINT32;
typedef uint16_t                DAP_UINT16;
typedef uint8_t                 DAP_UINT8;

/*
 * The DAP_PVOID64 is a special problem here!
 */
#ifdef _ILP32       /* _ILP32 32 bit int32/long32/ptr32 model */
/* Big-endian 32bit model: address must be right-justified in 64bit word */
typedef union {
        DAP_UINT64              AddressBits; /* only the right 32 bits used */
        DAP_UINT64              Address;     /* Note: not a DAP_PVOID type */
                                            /*  cast required */
} DAP_PVOID64;

#else               /* _LP64 64 bit int32/long64/ptr64 model */

typedef union {
     DAP_UINT64                 AddressBits;
     DAP_PVOID                  Address;
} DAP_PVOID64;
#endif

#endif  /* _DAFS_API_PD_H_ */
```

```
/*
 *  Copyright (C) 2000, 2001  DAFS Collaborative
 *
 *  Direct Access File System Application Programming Interface (DAFS API)
 *  sample header file:  dafs_api_pd.h  --  Windows version
 *
 *  Abstract:
 *
 *      Platform-dependent type definitions - an example.
 *
 *      N.B.: This file contains sample definitions, and is not
 *      necessarily correct for any particular hardware/software
 *      combination.
 */
#ifndef _DAFS_API_PD_H_
#define _DAFS_API_PD_H_

/*
 * Fundamental types
 */
typedef char                    DAP_CHAR;
typedef unsigned char           DAP_UCHAR;
typedef unsigned int            DAP_BOOLEAN;
typedef unsigned int            DAP_COUNT;
typedef unsigned long long      DAP_LENGTH;
typedef unsigned long long      DAP_OFFSET;
typedef unsigned long long      DAP_OPAQUE;
typedef unsigned long long      DAP_BITMAP;
typedef void *                  DAP_PVOID;

typedef unsigned long           DAP_CREATE_MODE;
typedef unsigned long           DAP_FLAGS;

/*
 * Machine-Dependent types for Windows32 and Windows64.
 */
#if Windows
typedef unsigned __int64        DAP_UINT64;
typedef unsigned __int32        DAP_UINT32;
typedef unsigned __int16        DAP_UINT16;
typedef unsigned __int8         DAP_UINT8;

typedef union {
        DAP_UINT64              AddressBits;
        DAP_PVOID               Address;
} DAP_PVOID64;

#endif  /* _DAFS_API_PD_H_ */
```

dap_async_listio

Initiate a list of asynchronous I/O operations.

```
DAP_ERROR
dap_async_listio(
                DAP_COUNT               io_count,
                DAP_IO_REQUEST          **io_requests,
                DAP_CG_HANDLE           cg_handle,
                DAP_IO_RESULT           **io_descs,
                DAP_UINT32              usec_window,
                DAP_UINT32              num_completions );
```

Description

Initiates a list of asynchronous read and/or write operations and returns control to the calling program. These operations continue concurrently with other activity of the process. This interface is unlike dap_async_read() and dap_async_write() in that it is fully general, supporting:

- scatter/gather to/from disjoint regions of a file
- scatter/gather to/from discontiguous application memory regions
- initiation of multiple I/O operations, possibly to a number of different files
- separate completion status for each I/O operation

The I/O requests are specified by the array of pointers to DAP_IO_REQUEST structures indicated by io_requests. The results are stored in the array of pointers to DAP_IO_RESULT structures indicated by io_descs. These arrays parallel each other and must contain the same number of entries, which is indicated by io_count. Each DAP_IO_RESULT corresponds to its counterpart DAP_IO_REQUEST, and is used to reap the results when the I/O has completed. The completion information can be reaped by the use of the dap_cg_wait() and dap_cg_batchwait() interfaces, which return pointer(s) to the DAP_IO_RESULT structures which completed (assuming that cg_handle was non-NULL). dap_cg_wait() returns the DAP_IO_RESULTs one at a time, while dap_cg_batchwait() returns pointers to a number of DAP_IO_RESULT structures. The order in which the completions will return is not specified. The application is responsible for remembering which DAP_IO_RESULT corresponds to which DAP_IO_REQUEST (for which the dap_app_private field of the DAP_IO_RESULT should be useful).

There are three ways to reap I/O completions if cg_handle was non-NULL:

- dap_cg_wait() awaits any single DAP_IO_RESULT
- dap_cg_done() polls for any single DAP_IO_RESULT
- dap_cg_batchwait() awaits a number of DAP_IO_RESULTs

There are two methods for reaping completions of I/Os that were issued using a NULL cg_handle:

- dap_io_wait() awaits a specific DAP_IO_RESULT
- dap_io_done() polls a specific DAP_IO_RESULT

Noto Bene: Both atomic append and non-blocking modes are silently ignored by dap_async_listio(). In other words, the use of the DAP_APPEND and DAP_NONBLOCK flags

when obtaining file handles from dap_open_file() has no effect on the behavior of dap_async_listio().

Arguments

io_count indicates the number of pointers to DAP_IO_REQUEST structures and pointers to DAP_IO_RESULT structures being passed in by the caller, and must be greater than zero.

io_requests is the address of an array of pointers to DAP_IO_REQUEST structures, providing the pertinent information for each I/O operation being requested. Each structure thus pointed to is a variable-sized structure containing elements describing scatter/gather to both the file and application memory, and contains:

dap_file_handle - a DAFS file handle as returned by the dap_open_file() or dap_open_nattr() calls.

dap_write_request - a boolean indicating whether this is a write request (zero indicates a read request).

dap_num_file_chunks - the number of DAP_FILE_DESC structures included in the array of file_chunks. This provides scatter or gather within the file indicated by file_handle.

dap_num_mem_chunks - the number of DAP_MEM_DESC structures included in the array of mem_chunks.

dap_file_chunks - a contiguous array of DAP_FILE_DESC structures, describing the file regions to (or from) which I/O should be done, each containing two fields describing the target range within the file: file_offset and byte_count. There is also a third field, dap_cache_hint, composed of bit flags that provide suggestions that may allow servers to optimize workload handling and aid cache management. Undefined bits should be zero. The bit definitions are enumerated in dafs_api.h, and indicate the likelihood of that a given chunk of a file will be read or written again in the near future.

dap_mem_chunks - a contiguous array of DAP_MEM_DESC structures, describing the application memory regions to (or from) which I/O should be done, each containing three fields: dap_mem_handle, dap_bufferp, and dap_buffer_len.

cg_handle is a completion group handle used to await the completion of the requested operation, and may be NULL.

io_descs is the address of an array of pointers to DAP_IO_RESULT structures, which upon successful return from dap_async_listio() can be used to await completion of this operation. In no case is the dap_app_private field of these structures modified by the provider. A non-zero dap_error field indicates that there was an error performing the requested write; otherwise, dap_length contains the number of bytes actually written to the file. Other fields are undefined and should not be referenced.

usec_window is a hint indicating that the application is willing to allow a certain amount of time to complete the I/O. This hint may enable the server to optimize the scheduling of data transfers. A value of zero provides default I/O scheduling behavior. The routine dap_expedite()

may be used to request that a previously issued I/O be processed immediately (in other words, that its usec_window be changed to zero).

num_completions is a hint indicating that the application is likely to want to reap its I/O completions in groups. This hint allows both server and client to optimize the handling of completions. Values less than or equal to one provide default behavior.

Returns

Returns zero on success. Otherwise, one of the error values below may be returned, either directly from this call or indirectly through the dap_error field of the DAP_IO_RESULT supplied by the application. Note that there are no partial failures; in other words, if an error is returned from dap_async_listio() either no I/Os have been initiated, or a catastrophic and unrecoverable transport failure has occurred. If there are problems with individual I/O requests, these will manifest as non-zero values in the dap_error field of the DAP_IO_RESULT when the I/O is reaped.

See Also

dap_cg_done(), dap_cg_wait(), dap_io_done(), dap_io_wait(), and dap_cg_batchwait(), dap_expedite().

Errors

DAP_ERROR_INVALID_FILE_HANDLE
: file_handle isn't a valid DAFS file handle.

DAP_ERROR_INVALID_MEM_HANDLE
: Some entry in the mem_desc has an invalid registered memory handle.

DAP_ERROR_INVALID_CG_HANDLE
: The completion group handle was invalid.

DAP_ERROR_BAD_ARG
: The io_count was less than or equal to zero.

DAP_ERROR_UNREGISTERED_MEM
: Some entry in the DAP_MEM_DESC is not valid. Either the dap_bufferp is not within a valid registered memory virtual region, or the end of the buffer extends beyond the memory region referred to by the memory handle, or a NULL dap_mem_handle was given and the Provider was unable to register the memory region on the fly.

DAP_ERROR_IO_OVERLAP
: This request attempts to write to an area that overlaps a pending write request, possibly leading to undefined results due to the lack of ordering guarantees among simultaneous pending I/O requests. Note that it may not be possible for the Provider to detect overlap with previously issued I/Os, and that this condition should be avoided by the application.

DAP_ERROR_LOCKED
: I/O attempt to a locked region.

DAP_ERROR_WRITE_TOOBIG

This operation is being done on a file opened for append mode, and the size of the write exceeds the maximum for atomic append operations.

DAP_ERROR_FBIG
This operation would exceed the maximum size supported, or would exceed the resources available on the server.

DAP_ERROR_DQUOT
This operation would exceed a resource (quota) limit.

DAP_ERROR_IO
There was a hard and unrecoverable media (disk) error.

DAP_ERROR_NXIO
There was no such device or address (perhaps hardware was taken off-line).

DAP_ERROR_NODEV
The operation is not supported by the device (such as writing to read-only media).

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_async_read

Asynchronous file read operation.

```
DAP_ERROR
dap_async_read(
                DAP_FILE_HANDLE         file_handle,
                DAP_OFFSET              file_offset,
                DAP_COUNT               io_count,
                const DAP_MEM_DESC      *mem_desc,
                DAP_CG_HANDLE           cg_handle,
                DAP_IO_RESULT           *io_desc );
```

Description

Initiates one asynchronous read and returns control to the calling program. The read operation continues concurrently with other activity of the process, attempting to read data from the file referenced by the file handle at an offset of file_offset into the buffer or buffers indicated by mem_desc.

The result of the asynchronous operation is stored in the structure pointed to by io_desc.

There are a variety of interfaces available to reap I/O completions if cg_handle was non-NULL:

- dap_cg_wait() awaits any single DAP_IO_RESULT
- dap_cg_done() polls for any single DAP_IO_RESULT
- dap_cg_batchwait() awaits a number of DAP_IO_RESULTs

There are two methods for reaping completions of I/Os that were issued using a NULL cg_handle:

- dap_io_wait() awaits a specific DAP_IO_RESULT
- dap_io_done() polls a specific DAP_IO_RESULT

Arguments

file_handle is a DAFS file handle as returned by the dap_open_file() or dap_open_nattr() calls.

file_offset is the offset in the file from which to read data.

io_count is the number of DAP_MEM_DESC structures in the array that mem_desc points to, and must be greater than zero.

mem_desc is pointer to a (vector of) descriptor(s) for the asynchronous I/O operations. Each entry in the vector contains:

dap_mem_handle - a DAFS memory handle that is associated with the buffer pointer and length. If DAP_NULL_MEM_HANDLE is supplied, the provider will register and bind the memory on the fly; it may cache these mappings to speed later operations.

dap_bufferp - a buffer pointer to somewhere within the registered memory region referred to by the DAFS memory handle.

dap_buffer_len - the length in bytes of the buffer.

cg_handle is a completion group handle used to await the completion of the requested operation, and may be NULL.

io_desc is a pointer to a DAP_IO_RESULT structure, which upon successful return from dap_async_read() can be used to await completion of this operation. In no case is dap_app_private modified by the provider. A non-zero dap_error field indicates that there was an error performing the requested read; otherwise, dap_length contains the number of bytes actually read from the file. Other fields are undefined and should not be referenced.

Returns

Returns zero on success. Otherwise, one of the error values below may be returned, either directly from this call or indirectly through the dap_error field of the DAP_IO_RESULT supplied by the application.

See Also

dap_cg_done(), dap_cg_wait(), dap_io_done(), dap_io_wait(), and dap_cg_batchwait().

Errors

DAP_ERROR_INVALID_FILE_HANDLE
: file_handle isn't a valid DAFS file object.

DAP_ERROR_INVALID_MEM_HANDLE
: Some entry in the mem_desc has an invalid registered memory handle.

DAP_ERROR_UNREGISTERED_MEM
: Some entry in the DAP_MEM_DESC is not valid. Either the dap_bufferp is not within a valid registered memory virtual region, or the end of the buffer extends beyond the memory region referred to by the memory handle, or a NULL dap_mem_handle was given and the Provider was unable to register the memory region on the fly.

DAP_ERROR_BAD_ARG
: The io_count was less than or equal to zero.

DAP_ERROR_WOULD_BLOCK
: The file being read was opened with the DAP_NONBLOCK flag, and this attempt to initiate asynchronous I/O would have to block due to either flow control or resource exhaustion.

DAP_ERROR_INVALID_CG_HANDLE
: The completion group handle was invalid.

DAP_ERROR_IO_OVERLAP
: This request attempts to write to an area that overlaps a pending write request, possibly leading to undefined results due to the lack of ordering guarantees among simultaneous pending I/O requests.

DAP_ERROR_IO
: There was a hard and unrecoverable media (disk) error.

DAP_ERROR_TRANSPORT_FAILURE

A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_async_read_dir

Read some number of directory entries.

```
DAP_ERROR
dap_async_read_dir(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_OFFSET              cookie,
                DAP_OPAQUE              cookie_verifier,
                DAP_MEM_HANDLE          mem_handle,
                DAP_LENGTH              size,
                DAP_READDIR_RESULT      *resultp,
                DAP_CG_HANDLE           cg_handle,
                DAP_IO_RESULT           *io_desc );
```

Description

This routine reads some number of entries from the directory indicated by dir_handle. The application provides some number of bytes of (registered) memory and a cookie of zero to begin reading a directory. The provider will return a DAP_READDIR_RESULT structure, which contains a cookie verified (see below), a flag indicating when the last item has been read, the number of items read, and a vector of DAP_DIRENTRY structures containing the actual directory data. The directory data consists of a number of fixed-size DAP_DIRENTRY structures, each containing the DAP_FILETYPE of the file system object, a pointer to its NUL-terminated name and an opaque cookie to be passed to a subsequent call to dap_async_read_dir() et al. to access the remaining directory entries.

There are no further entries to be read from the directory indicated by dir_handle when the dap_end_flag is set in the DAP_READDIR_RESULT structure.

There are a variety of interfaces available to reap I/O completions if cg_handle was non-NULL:

- dap_cg_wait() awaits any single DAP_IO_RESULT
- dap_cg_done() polls for any single DAP_IO_RESULT
- dap_cg_batchwait() awaits a number of DAP_IO_RESULTs

There are two methods for reaping completions of I/Os that were issued using a NULL cg_handle:

- dap_io_wait() awaits a specific DAP_IO_RESULT
- dap_io_done() polls a specific DAP_IO_RESULT

Arguments

dir_handle is a DAFS directory handle returned from the dap_open_dir() call, and indicates the directory which is to be read.

cookie is a value that represents where the operation should start within the directory. A value of 0 (zero) for the cookie is used to start reading at the beginning of the directory. For subsequent requests, the caller specifies a cookie value that is provided by the server in response to a previous request (dap_readdir_result.dap_entry[index].dap_direntry_cookie).

cookie_verifier should be set to 0 (zero) when the cookie value is 0 (zero) on the first directory read. On subsequent requests, it should be a cookieverf as returned by the server (dap_readdir_result.dap_cookiev). The cookie_verifier must match that returned by the read operation in which the cookie was acquired.

mem_handle is a DAFS memory handle that is associated with the application buffer pointed to by resultp, and may be NULL.

size is the length in bytes of the application buffer pointed to by resultp.

resultp points to the application buffer, a variable-size DAP_READDIR_RESULT structure. Upon successful completion, this contains:

- dap_cookiev - the cookie verifier returned by the server
- dap_end_flag - set to non-zero when the last entry in the directory has been read.
- dap_num_entries - the number of valid entires in the variable-size dap_entry array.
- dap_entry - the output array of DAP_DIRENTRY structures, containing dap_num_entries valid members. Each contains:
  - dap_direntry_type - indicating the DAP_FILETYPE of this entry.
  - dap_direntry_cookie - an opaque cookie to be handed to a subsequent call to any of the directory reading routines in order to obtain the next DAP_DIRENTRY.
  - dap_direntry_name - a pointer into this application-managed storage, to the NUL-terminated name of this file system object.
  - dap_direntry_attrp - a pointer into this application-managed storage, to the requested attributes of this file system object. This pointer will always be NULL when the attributes are obtained with this interface (see dap_async_read_dir2() and dap_read_dir2()).

cg_handle is a completion group handle used to await the completion of the requested operation, and may be NULL.

io_desc is a pointer to a DAP_IO_RESULT structure, which upon successful return from dap_async_read_dir() can be used to await completion of this operation. In no case is dap_app_private modified by the provider. A non-zero dap_error field indicates that there was an error performing the requested read. The contents of other fields is undefined.

Returns

Returns zero on success. Otherwise, one of the error values below may be returned, either directly from this call or indirectly through the dap_error field of the DAP_IO_RESULT supplied by the application.

See Also

dap_cg_done(), dap_cg_wait(), dap_io_done(), dap_io_wait(), and dap_cg_batchwait().

Errors

DAP_ERROR_INVALID_DIR_HANDLE
The base_dir_handle given was invalid.

DAP_ERROR_INVALID_CG_HANDLE
The completion group handle was invalid.

DAP_ERROR_BADCOOKIE
The cookier/cookie_verifier pair supplied was invalid.

DAP_ERROR_WOULD_BLOCK
The directory being read was opened with the DAP_NONBLOCK flag, and this attempt to initiate asynchronous I/O would have to block due to either flow control or resource exhaustion.

DAP_ERROR_BUFFER_TOO_SMALL
The number of bytes given was not sufficient to hold even a single DAP_DIRENTRY and its name. The application should try again using a larger buffer.

DAP_ERROR_IO
There was a hard and unrecoverable media (disk) error.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_async_read_dir2

Read some number of directory entries and their attributes.

```
DAP_ERROR
dap_async_read_dir2(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_OFFSET              cookie,
                DAP_OPAQUE              cookie_verifier,
                DAP_BITMAP              attrs_requested,
                DAP_MEM_HANDLE          mem_handle,
                DAP_LENGTH              size,
                DAP_READDIR_RESULT      *resultp,
                DAP_CG_HANDLE           cg_handle,
                DAP_IO_RESULT           *io_desc );
```

Description

This routine reads some number of entries, along with their attributes, from the directory indicated by dir_handle. The application provides some number of bytes of (registered) memory and a cookie of zero to begin reading a directory. The provider will return a DAP_READDIR_RESULT structure, which contains a cookie verified (see below), a flag indicating when the last item has been read, the number of items read, and a vector of DAP_DIRENTRY structures containing the actual directory data. The directory data consists of a number of fixed-size DAP_DIRENTRY structures, each containing the DAP_FILETYPE of the file system object, a pointer to its NUL-terminated name and an opaque cookie to be passed to a subsequent call to dap_async_read_dir() et al. to access the remaining directory entries.

The attributes desired are specified by setting bits of the attrs_requested parameter, and the valid_attrs field of the DAP_STAT_DESC indicates which attributes were actually returned (which may be fewer than requested).

There are no further entries to be read from the directory indicated by dir_handle when the dap_end_flag is set in the DAP_READDIR_RESULT structure.

There are a variety of interfaces available to reap I/O completions if cg_handle was non-NULL:

- dap_cg_wait() awaits any single DAP_IO_RESULT
- dap_cg_done() polls for any single DAP_IO_RESULT
- dap_cg_batchwait() awaits a number of DAP_IO_RESULTs

There are two methods for reaping completions of I/Os that were issued using a NULL cg_handle:

- dap_io_wait() awaits a specific DAP_IO_RESULT
- dap_io_done() polls a specific DAP_IO_RESULT

Arguments

dir_handle is a DAFS directory handle returned from the dap_open_dir() call, and indicates the directory which is to be read.

cookie is a value that represents where the operation should start within the directory. A value of 0 (zero) for the

cookie is used to start reading at the beginning of the directory. For subsequent requests, the caller specifies a cookie value that is provided by the server in response to a previous request (dap_readdir_result.dap_entry[index].dap_direntry_cookie).

cookie_verifier should be set to 0 (zero) when the cookie value is 0 (zero) on the first directory read. On subsequent requests, it should be a cookieverf as returned by the server (dap_readdir_result.dap_cookiev). The cookieverf must match that returned by the read operation in which the cookie was acquired.

attrs_requested indicates which attributes are desired; bits set to one indicate attributes that are requested.

mem_handle is a DAFS memory handle that is associated with the application buffer pointed to by resultp, and may be NULL.

size is the length in bytes of the application buffer pointed to by resultp.

resultp points to the application buffer, a variable-size DAP_READDIR_RESULT structure. Upon successful completion, this contains:

- dap_cookiev - the cookie verifier returned by the server
- dap_end_flag - set to non-zero when the last entry in the directory has been read.
- dap_num_entries - the number of valid entires in the variable-size dap_entry array.
- dap_entry - the output array of DAP_DIRENTRY structures, containing dap_num_entries valid members. Each contains:
  - dap_direntry_type - indicating the DAP_FILETYPE of this entry.
  - dap_direntry_cookie - an opaque cookie to be handed to a subsequent call to any of the directory reading routines in order to obtain the next DAP_DIRENTRY.
  - dap_direntry_name - a pointer into this application-managed storage, to the NUL-terminated name of this file system object.
  - dap_direntry_attrp - a pointer into this application-managed storage, to the requested attributes of this file system object.

cg_handle is a completion group handle used to await the completion of the requested operation, and may be NULL.

io_desc is a pointer to a DAP_IO_RESULT structure, which upon successful return from dap_async_read_dir2() can be used to await completion of this operation. In no case is dap_app_private modified by the provider. A non-zero dap_error field indicates that there was an error performing the requested read. The contents of other fields is undefined.

Returns

Returns zero on success. Otherwise, one of the error values below may be returned, either directly from this call or indirectly through the dap_error field of the DAP_IO_RESULT supplied by the application.

See Also

dap_cg_done(), dap_cg_wait(), dap_io_done(), dap_io_wait(), and dap_cg_batchwait().

Errors

DAP_ERROR_INVALID_DIR_HANDLE
: The base_dir_handle given was invalid.

DAP_ERROR_INVALID_CG_HANDLE
: The completion group handle was invalid.

DAP_ERROR_WOULD_BLOCK
: The directory being read was opened with the DAP_NONBLOCK flag, and this attempt to initiate asynchronous I/O would have to block due to either flow control or resource exhaustion.

DAP_ERROR_BUFFER_TOO_SMALL
: The number of bytes given was not sufficient to hold even a single DAP_DIRENTRY with attributes and its name. The application should try again using a larger buffer.

DAP_ERROR_IO
: There was a hard and unrecoverable media (disk) error.

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_async_write

Asynchronous file write operation.

```
DAP_ERROR
dap_async_write(
                DAP_FILE_HANDLE         file_handle,
                DAP_OFFSET              file_offset,
                DAP_COUNT               io_count,
                const DAP_MEM_DESC      *mem_desc,
                DAP_CG_HANDLE           cg_handle,
                DAP_IO_RESULT           *io_desc );
```

Description

Initiates one asynchronous write operation and returns control to the calling program. The write operation continues concurrently with other activity of the process, attempting to write data from the buffer or buffers pointed to by mem_desc to the file referenced by the file handle at an offset of file_offset.

The result of the asynchronous operation is stored in the structure pointed to by io_desc.

There are a variety of interfaces available to reap I/O completions if cg_handle was non-NULL:

- dap_cg_wait() awaits any single DAP_IO_RESULT
- dap_cg_done() polls for any single DAP_IO_RESULT
- dap_cg_batchwait() awaits a number of DAP_IO_RESULTs

There are two methods for reaping completions of I/Os that were issued using a NULL cg_handle:

- dap_io_wait() awaits a specific DAP_IO_RESULT
- dap_io_done() polls a specific DAP_IO_RESULT

Arguments

file_handle is a DAFS file handle as returned by the dap_open_file() or dap_open_nattr() calls.

file_offset is the offset in the file to write the data. This parameter is ignored if file_handle was opened with the DAP_APPEND option.

io_count is the number of DAP_MEM_DESC structures in the array that mem_desc points to, and must be greater than zero.

mem_desc is pointer to a (vector of) descriptor(s) for the asynchronous I/O operations. Each entry in the vector contains:

dap_mem_handle - a DAFS memory handle that is associated with the buffer pointer and length. If DAP_NULL_MEM_HANDLE is supplied, the provider will register and bind the memory on the fly; it may cache these mappings to speed later operations.

dap_bufferp - a buffer pointer to somewhere within the registered memory region referred to by the DAFS memory handle.

dap_buffer_len - the length in bytes of the buffer.

cg_handle is a completion group handle used to await the completion of the requested operation, and may be NULL.

io_desc is a pointer to a DAP_IO_RESULT structure, which upon successful return from dap_async_write() can be used to await completion of this operation. In no case is dap_app_private modified by the provider. A non-zero dap_error field indicates that there was an error performing the requested write; otherwise, dap_length contains the number of bytes actually written to the file. Other fields are undefined and should not be referenced.

Returns

Returns zero on success. Otherwise, one of the error values below may be returned, either directly from this call or indirectly through the dap_error field of the DAP_IO_RESULT supplied by the application.

See Also

dap_cg_done(), dap_cg_wait(), dap_io_done(), dap_io_wait(), and dap_cg_batchwait().

Errors

DAP_ERROR_INVALID_FILE_HANDLE
file_handle isn't a valid DAFS file handle.

DAP_ERROR_INVALID_MEM_HANDLE
Some entry in the mem_desc has an invalid registered memory handle.

DAP_ERROR_INVALID_CG_HANDLE
The completion group handle was invalid.

DAP_ERROR_BAD_ARG
The io_count was less than or equal to zero.

DAP_ERROR_WOULD_BLOCK
The file being written was opened with the DAP_NONBLOCK flag, and this attempt to initiate asynchronous I/O would have to block due to either flow control or resource exhaustion.

DAP_ERROR_UNREGISTERED_MEM
Some entry in the DAP_MEM_DESC is not valid. Either the dap_bufferp is not within a valid registered memory virtual region, or the end of the buffer extends beyond the memory region referred to by the memory handle, or a NULL dap_mem_handle was given and the Provider was unable to register the memory region on the fly.

DAP_ERROR_IO_OVERLAP
This request attempts to write to an area that overlaps a pending write request, possibly leading to undefined results due to the lack of ordering guarantees among simultaneous pending I/O requests.

DAP_ERROR_LOCKED
I/O attempt to a locked region.

DAP_ERROR_WRITE_TOOBIG
This operation is being done on a file opened for append mode, and the size of the write exceeds the maximum for atomic append operations.

DAP_ERROR_FBIG
This operation would exceed the maximum size supported, or would exceed the resources available on the server.

DAP_ERROR_DQUOT
This operation would exceed a resource (quota) limit.

DAP_ERROR_IO
There was a hard and unrecoverable media (disk) error.

DAP_ERROR_NXIO
There was no such device or address (perhaps hardware was taken off-line).

DAP_ERROR_NODEV
The operation is not supported by the device (such as writing to read-only media).

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_auth_callback

Register an authentication callback with the DAFS Provider.

```
DAP_ERROR
dap_auth_callback(
                unsigned int            auth_type_mask,
                void *                  handler_context,
                unsigned int (*dap_auth_handler) (
                        void *          context,
                        unsigned int    opcode,
                        unsigned int    auth_type,
                        unsigned int    * auth_size,
                        unsigned int    * auth_data ) );
```

Description

dap_auth_callback() is used by the application to register an authentication handling function with the DAFS Provider. This handler will be called by the Provider whenever it is necessary to authenticate while initiating contact with a new server. The handler indicates the method and supplies the data to be used in authenticating. A subsequent registration will overwrite, or change, the registered handler.

Arguments

auth_type_mask indicates the type of authentication which this callback routine knows how to handle. A separate handler may be registered for each of the supported authentication type (DAP_AUTH_NONE, DAP_AUTH_TEXT, DAP_AUTH_GSS) though since there is no data to be supplied in the case of DAP_AUTH_NONE, the handler will never be invoked.

handler_context is a datum which is passed without modification to the handler when it is invoked.

dap_auth_handler is the address of the application-defined function to be called when authentication must be done (presumably due to a new server being contacted). This function is not guaranteed to run in the context of the thread which called dap_auth_callback() to register dap_auth_handler, and must return zero after supplying the authentication data. A non-zero return indicates to the Provider that the handler could not supply the requested data. The handler may return DAP_ERROR_TOO_SMALL when invoked to indicate that the supplied buffer (auth_data) was insufficient, after setting auth_size to the size desired to allow the Provider to re-try the handler. A NULL may be used to unregister (though unregistering the handler for DAP_AUTH_NONE makes very little sense).

context is the value passed in as handler_context when the handler was registered with the Provider.

opcode indicates what operation is being performed, and the value will change (in an auth_type-specific way) when a multi-step protocol is being executed by multiple sequential callback invocations. For DAP_AUTH_TEXT handler invocations, opcode is always zero.

auth_type indicates the type of authentication being attempted.

auth_size and auth_data are pointers indicating where the application should deposit the authentication data, and how big the buffer (supplied by the Provider) is. auth_size should be treated as an IN/OUT parameter, while only the buffer pointed to by auth_data, not the pointer itself, should be modified. An error of DAP_ERROR_TOO_SMALL may be returned if the Provider supplies a buffer of insufficient size; auth_size should be set to the desired number of bytes prior to returning from the handler.

Returns

Returns zero on success. Otherwise returns one of the DAFS errors listed below.

Bugs

The details of the various authentication mechanisms supported using DAP_AUTH_GSS need to be elaborated, explained, and documented.

Errors

DAP_ERROR_AUTH_TYPE
Invalid DAFS authentication type

DAP_ERROR_AUTH_DENIED
The authentication information was incorrect or insufficient.

Other DAP_ERROR values may also be returned.

dap_cancel_async_op

Attempts to cancel a previously initiated I/O operation.

```
DAP_ERROR
dap_cancel_async_op(
                DAP_IO_RESULT           *io_desc );
```

Description

This call attempts to cancel an outstanding asynchronous I/O operation. This may not be possible. In all cases the application must wait for the I/O to complete using the usual methods, prior to attempting to re-use or free any underlying buffers (the typical motivation for I/O cancellation).

Arguments

io_desc is a pointer to the DAP_IO_RESULT structure which is being used to retrieve completion information for the operation which is being canceled. It is possible that the operation in question is being completed as the attempt at cancellation is being made, so the caller must await completion using the normal means, and check the dap_error field to determine the actual fate of the operation.

Returns

Returns zero on success. Otherwise, one of the error values below may be returned. In all cases the caller must await completion of the operation and check the dap_error field of the DAP_IO_RESULT supplied by the application.

Errors

DAP_ERROR_INVALID_IO_RESULT
: The operation associated with this DAP_IO_RESULT is no longer pending, and may have already completed.

Other DAP_ERROR values may also be returned.

dap_cg_batchwait

Block on a completion group, awaiting some number of I/O completions

```
DAP_ERROR
dap_cg_batchwait(
                DAP_CG_HANDLE           cg_handle,
                DAP_TIMEOUT             timeout,
                DAP_COUNT               *n_results,
                DAP_IO_RESULT           **io_desc );
```

Description

Blocks for up to a duration of "timeout" awaiting a number (n_results) of I/O completions from the specified completion group. If timeout is set to DAP_WAIT_NOWAIT, it checks and returns whatever is ready, without blocking. If timeout is set to DAP_WAIT_FOREVER, it will block until the requested number of I/Os complete before returning. Note however that (given a timeout other than DAP_WAIT_FOREVER) it is not an error for this routine to return fewer completions than requested; callers must be prepared to handle this situation correctly.

If I/Os have completed, this routine returns zero and updates the DAP_COUNT pointed to by n_results to indicate the number of pointers (in the array of pointers indicated by io_desc) which are valid. Each of the pointers thus returned points to the DAP_IO_RESULT structure that was originally used to initiate the corresponding I/O operation, filled in with the results for the completed operation.

Note that if n_results points to a count of one (1), this this function is equivalent to dap_cg_wait().

Arguments

cg_handle the completion group handle on which to check for I/O completions.

timeout indicates how long to wait for an I/O completion before returning DAP_ERROR_PENDING_IO. DAP_WAIT_NOWAIT is a polling operation and DAP_WAIT_FOREVER will block until n_results completions are reaped.

n_results is a pointer to a DAP_COUNT variable indicating the desired number of completions to reap, which is also the number of entries in the array of pointers to DAP_IO_RESULT structures whose address is supplied in io_desc. Upon return it is set to the number of completions that were actually reaped.

io_desc is the address of an array of pointers to DAP_IO_RESULT structures, which will be filled in to point to the actual structures that were supplied to the call originating the asynchronous operation (dap_async_read(), dap_async_write(), dap_async_listio(), etc.).

Returns

Returns zero on success. Otherwise returns one of the error values listed below. An error from the asynchronous operation may be contained in the dap_error field of the DAP_IO_RESULT.

Errors

DAP_ERROR_PENDING_IO
: The timeout expired before an I/O operation in the completion group completed.

DAP_ERROR_INVALID_CG_HANDLE
: The completion group handle was invalid.

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_cg_done

Checks for any I/O completion with completion group.

```
DAP_ERROR
dap_cg_done(
                DAP_CG_HANDLE           cg_handle,
                DAP_IO_RESULT           **io_desc );
```

Description

Checks for an I/O completion in a completion group. This is functionally equivalent to a call to dap_cg_wait() with an instantaneous timeout (DAP_WAIT_NOWAIT). If an I/O has completed, this routine returns zero and the pointer referenced by io_desc has been updated to point to the DAP_IO_RESULT used to initiate the I/O, which has been filled in with the results for the completed operation.

Arguments

cg_handle the completion group handle on which to check for I/O completions.

io_desc is the address of a pointer to a DAP_IO_RESULT structure, which will be filled in to point to an actual structure that was supplied to the call originating the asynchronous operation (dap_async_read(), dap_async_write(), dap_async_listio(), etc.).

Returns

Returns zero on success. Otherwise returns one of the error values listed below. An error from the asynchronous operation may be contained in the dap_error field of the DAP_IO_RESULT.

Errors

DAP_ERROR_INVALID_CG_HANDLE
The completion group handle was invalid.

DAP_ERROR_PENDING_IO
There have been no I/O completions for this completion group. Try again later.

DAP_ERROR_NO_IO_PENDING
There are no I/O operations attached to the completion group.

DAP_ERROR_IO_CANCELLATION
The operation associated with this DAP_IO_RESULT was successfully cancelled. This error will only appear in io_desc->dap_error.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_cg_wait

Block for an I/O completion using a completion group.

```
DAP_ERROR
dap_cg_wait(
                DAP_CG_HANDLE           cg_handle,
                DAP_TIMEOUT             timeout,
                DAP_IO_RESULT           **io_desc );
```

Description

Blocks for up to a duration of "timeout" awaiting an I/O completion in the requested completion group. If timeout is set to DAP_WAIT_NOWAIT, it checks and returns without blocking. If timeout is set to DAP_WAIT_FOREVER, it will block until an I/O completes before returning. If an I/O has completed, this routine returns zero and the pointer referenced by io_desc has been updated to point to the DAP_IO_RESULT used to initiate the I/O, which has been filled in with the results for the completed operation.

Arguments

cg_handle the completion group handle on which to check for I/O completions.

timeout indicates how long to wait for an I/O completion before returning DAP_ERROR_PENDING_IO. If timeout is equal to DAP_WAIT_NOWAIT this call is equivalent to dap_cg_done(). If timeout is equal to DAP_WAIT_FOREVER it will block until an I/O completes.

io_desc is the address of a pointer to a DAP_IO_RESULT structure, which will be filled in to point to an actual structure that was supplied to the call originating the asynchronous operation (dap_async_read(), dap_async_write(), dap_async_listio(), etc.).

Returns

Returns zero on success. Otherwise returns one of the error values listed below. An error from the asynchronous operation may be contained in the dap_error field of the DAP_IO_RESULT.

Errors

DAP_ERROR_PENDING_IO
: The timeout expired before an I/O operation in the completion group completed.

DAP_ERROR_INVALID_CG_HANDLE
: The completion group handle was invalid.

DAP_ERROR_IO_CANCELLATION
: The operation associated with this DAP_IO_RESULT was successfully cancelled. This error will only appear in io_desc->dap_error.

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_chmod

Change the mode of a file or directory.

```
DAP_ERROR
dap_chmod(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path,
                DAP_CREATE_MODE         mode );
```

Description

This routine sets the permission bits of the file or directory specified by path and dir_handle.

The directory handle and path together indicate the target of this call. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the file. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a DAFS directory handle returned from the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is interpreted relative to dir_handle, and must lead to a valid file or directory.

mode specifies the permission bits to be set; these are POSIX-style permission bits.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
: The base_dir_handle given was invalid.

DAP_ERROR_INVALID_CRED_HANDLE
: The credential handle was invalid.

DAP_ERROR_PERM
: The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_ACCESS
: The supplied credential does not allow access to the path requested.

DAP_ERROR_LOOP
: Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

Other DAP_ERROR values may also be returned.

dap_chown

Change the owner and group attributes of a file or directory.

```
DAP_ERROR
dap_chown(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path,
                const DAP_CHAR          *owner,
                const DAP_CHAR          *group );
```

Description

This routine sets the owner and group attributes of a file or directory.

The directory handle and path together indicate the target of this call. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the file. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a DAFS directory handle returned from the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is interpreted relative to dir_handle, and must lead to a valid file or directory.

owner and group point to representations of the desired owner and group attributes of the file or directory indicated by dir_handle and path. Either may be NULL, causing that parameter to be ignored.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
: The base_dir_handle given was invalid.

DAP_ERROR_INVALID_CRED_HANDLE
: The credential handle was invalid.

DAP_ERROR_PERM
: The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_ACCESS
: The supplied credential does not allow access to the path requested.

DAP_ERROR_LOOP
Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

Other DAP_ERROR values may also be returned.

dap_close_dir

Closes a directory object.

```
DAP_ERROR
dap_close_dir(
                DAP_DIRECTORY_HANDLE    dir_handle );
```

Description

Upon successful return, the directory referred to by dir_handle is no longer accessible. References in progress, whether reads or modifications, may be aborted. The client should wait for any outstanding operations and shut down gracefully.

Arguments

dir_handle is a DAFS directory handle (or named attribute directory handle) as returned by the dap_open_dir() call.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
: dir_handle is not a valid directory handle.

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_close_file

Closes a file (or named attribute).

```
DAP_ERROR
dap_close_file(
                DAP_FILE_HANDLE         file_handle );
```

Description

The file indicated by file_handle is closed. Upon successful completion, the file object referred to by file_handle is no longer accessible. Any outstanding operations on this file may be aborted. The application should wait for any outstanding operations and shut down gracefully.

Arguments

file_handle is a DAFS file handle as returned by the dap_open_file() or dap_open_nattr() calls.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_FILE_HANDLE
: file_handle isn’t a valid DAFS file handle.

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_create_cg

Creates a DAFS I/O completion group.

```
DAP_ERROR
dap_create_cg(
                DAP_CG_HANDLE           *cg_handle,
                DAP_COUNT               cg_entries );
```

Description

Creates a handle to which DAFS async I/O operations may be attached for the purpose of I/O completion notification.

Arguments

cg_handle is a pointer to a DAP_CG_HANDLE to be returned.

cg_entries is a hint from the application to the provider that the completion group being created ought to be able to handle the indicated number of entries without overflowing or blocking. This call may fail if the provider cannot create a sufficiently large completion group. This parameter is advisory; if a zero is supplied, the provider will attempt to use a reasonable size.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

The returned handle represents a completion group to which I/O operations can be attached at I/O initiation time (e.g.: dap_async_read() and dap_async_write()).

Errors

DAP_ERROR_NO_RESOURCES
: No resources available to create the completion group construct.

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_create_credential

Create a credential.

```
DAP_ERROR
dap_create_credential(
                void                    *handler_context,
                DAP_CRED_HANDLE         *cred_handle );
```

Description

DAFS applications can pre-register some number of credentials to be used in subsequent operations. This routine creates a single credential, represented by the cred_handle, which is supplied to other DAFS calls to indicate the caller's intended identification. This call simply returns an empty credential handle, which must be initialized using the callback registered using dap_cred_callback(). The callback is triggered when first contact to a server is made, typically upon first use of the empty credential handle, but possibly also due to server fail-over or data set migration.

Arguments

handler_context is a datum which is passed without modification to the handler registered with dap_cred_callback() when it is called to register cred_handle.

cred_handle is a pointer to a credential handle, which on successful return can be used with subsequent I/O operations.

Returns

Returns zero on success. Otherwise return one of the errors listed below.

Errors

DAP_ERROR_NO_RESOURCES
: The credential cannot be created due to some system-imposed limit (for example, the server's limit on credentials per client).

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_cred_callback

Register a credential callback with the DAFS Provider.

```
DAP_ERROR
dap_cred_callback(
                unsigned int (*dap_cred_handler) (
                        void *          context,
                        DAP_CRED_HANDLE cred_handle,
                        unsigned int    opcode,
                        DAP_CRED_TYPE   *cred_type,
                        DAP_LENGTH      *cred_data_len,
                        DAP_CRED_DATA   *cred_data ) );
```

Description

This routine is used by the application to register a credential handling function with the DAFS Provider. This handler will be called by the Provider whenever it is necessary to supply data to register a credential (originally obtained from dap_create_credential()) with a server, either on first access with that server or due to server fail-over or data set migration.

Arguments

dap_cred_handler is the address of the application-defined function to be called when the credential must be registered. This function is not guaranteed to run in the context of the thread which called dap_cred_callback() to register the handler, and must return zero after supplying the credential data. A non-zero return indicates to the Provider that the handler could not supply the requested data. The handler may return DAP_ERROR_TOO_SMALL when invoked to indicate that the supplied buffer (auth_data) was insufficient, after setting auth_size to the size desired, to allow the provider to re-try the handler.

context is the value passed in as handler_context when cred_handle was first created by dap_create_credential().

cred_handle is the credential handle obtained from dap_create_credential(), which now requires identification data in order to be registered.

opcode indicates which part of a multi-stage operation is being performed (see GSS details).

cred_type is the type of credential to be registered, and should be set by the handler.

cred_data_len is the length in bytes of the identification data. It is an IN/OUT parameter, since the buffer is allocated by the Provider, and should be set to the number of bytes actually used.

cred_data is a pointer to the buffer to be filled in with the identification data.

Returns

Returns zero on success. Otherwise return one of the errors listed below.

Errors

DAP_ERROR_NO_AUTH
: The application must successfully authenticate prior to attempting to create credentials.

DAP_ERROR_PERM
: No permission to obtain a credential handle using the supplied data.

DAP_ERROR_INVALID_CRED_TYPE
: The type of credential requested is invalid or not supported.

DAP_ERROR_NO_RESOURCES
: The credential cannot be created due to some system-imposed limit (for example, the server’s limit on credentials per client).

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_deregister_mem

Deregisters memory associated with the DAFS memory handle.

```
DAP_ERROR
dap_deregister_mem(
                DAP_MEM_HANDLE          mem_handle );
```

Description

This routine is the inverse of dap_register_mem() and dap_register_shbuffer(). Memory registrations are per-process, so deregistering a shared memory buffer effects only the caller.

Arguments

mem_handle is a memory handle used to identify the memory registration as returned by dap_register_mem() or dap_register_shbuffer().

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

See Also

dap_register_mem(), dap_register_shbuffer().

Errors

DAP_ERROR_INVALID_MEM_HANDLE
: The mem_handle argument is not a valid registered memory handle.

DAP_ERROR_BOUND_MEMORY
: This memory is in use by an outstanding I/O request.

Other DAP_ERROR values may also be returned.

dap_destroy_cg

Destroys a completion group.

```
DAP_ERROR
dap_destroy_cg(
                DAP_CG_HANDLE           cg_handle );
```

Description

Destroys a completion group. If successful, the completion group handle is destroyed and any associated resources are released. Subsequent use of this handle will produce an error. Pending operations may be aborted. The client should wait for all pending operations and clean up gracefully.

Arguments

cg_handle is a completion group handle as returned by dap_create_cg().

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_CG_HANDLE
The completion group handle was invalid.

DAP_ERROR_PENDING_IO
There is pending I/O on this DAP_CG_HANDLE.

Other DAP_ERROR values may also be returned.

dap_destroy_credential

Destroy a credential.

```
DAP_ERROR
dap_destroy_credential(
                DAP_CRED_HANDLE         cred_handle );
```

Description

This routine destroys a credential created by dap_create_credential(), once it is no longer needed.

Arguments

cred_handle is a valid credential handle that is no longer needed, which is to be destroyed.

Returns

Returns zero on success. Otherwise return one of the errors listed below.

Errors

DAP_ERROR_INVALID_CRED_HANDLE
The cred_handle supplied is not valid.

Other DAP_ERROR values may also be returned.

dap_expedite

Request immediate processing for a previously initiated I/O request.

```
DAP_ERROR
dap_expedite(
                DAP_IO_REQUEST          *io_request );
```

Description

This call attempts to cause an outstanding asynchronous I/O operation to be processed immediately. This may not be possible, as the I/O may have already been processed. Caveat: This call is inherently race-prone, so an error return is not necessarily reason for concern. In all cases the application must wait for the I/O to complete using the usual methods.

This call is useful when used in conjunction with dap_async_listio() requests that were issued with a non-zero usec_window parameter.

Arguments

io_desc is a pointer to the DAP_IO_RESULT structure which is being used to retrieve completion information for the operation which is being canceled.

See Also

dap_async_listio().

Returns

Returns zero on success. Otherwise, one of the error values below may be returned.

Errors

DAP_ERROR_INVALID_IO_RESULT
The operation associated with this DAP_IO_RESULT is no longer pending, and may have already completed.

Other DAP_ERROR values may also be returned.

dap_extensions

Interface for non-standard DAFS API extensions

```
DAP_ERROR
dap_extensions(
               unsigned int            request_token,
               DAP_PVOID               argument_ptr,
               DAP_COUNT               argument_size);
```

Description

This interface allows the addition of non-standard and vendor-unique extensions to the DAFS API in a way that does not result in binary compatibility. By implementing all non-standard extensions behind this interface, the support or lack of support for a given extension can be determined at run-time rather than being evidenced by a failure during compilation or linkage, which would prevent the application from being run at all.

Applications using this interface should always be prepared to handle error returns, since any support hidden within this interface is likely to be non-portable and specific to a particular vendor or host. Inasmuch as there is no central registry for the request_token bit patterns, carefully written applications will verify that the provider version is the one expected, prior to attempting to access other extended functionality.

All providers must support this interface, though returning DAP_ERROR_NOT_IMPLEMENTED is always adequate. It is nonetheless recommended that each provider implement the sole predefined request, which simply identifies the provider and its current version.

Arguments

request_token is an opaque bit pattern which identifies the specific extended function being requested.

argument_ptr points to the structure containing the parameters being supplied to this procedure.

argument_size indicates the size in bytes of the structure pointed to by argument_ptr.

Example

```
DAP_ERROR       rval;
DAP_EXT_VERSION dafs_version;

rval = dap_extensions(  DAP_EXT_GETVERSION,
                        (DAP_PVOID) &dafs_version,
                        sizeof(dafs_version) );
if (rval == DAP_SUCCESS)
        printf("Current DAFS Provider: %s Version %2u.%02u\n",
                        dafs_version.dap_provider,
                        dafs_version.dap_major,
                        dafs_version.dap_minor);
```

Returns

Returns zero on success. Otherwise, any of the DAFS errors may be returned, including those mentioned below.

Errors

DAP_ERROR_NOT_IMPLEMENTED
: The requested function is not implemented.

DAP_ERROR_INVALID_ADDRESS
: The address of the arguments, or an address with them, was not valid.

DAP_ERROR_NO_RESOURCES
: There were insufficient resources to perform the requested action.

DAP_ERROR_INVALID_FILE_HANDLE
DAP_ERROR_INVALID_DIR_HANDLE
DAP_ERROR_INVALID_CG_HANDLE
DAP_ERROR_INVALID_MEM_HANDLE
: A handle supplied in the arguments was not valid.

DAP_ERROR_PERM
: The caller does not have permission to perform the requested action.

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_fchmod

Change the mode of a file or directory.

```
DAP_ERROR
dap_fchmod(
                DAP_HANDLE              some_handle,
                DAP_CREATE_MODE         mode );
```

Description

This routine sets the permission bits of a file or directory, given a valid handle.

Arguments

some_handle is a valid handle to a file or directory, obtained from, for example, dap_open_dir(), or dap_open_file().

mode specifies the permission bits to be set; these are POSIX-style permission bits.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
The directory handle some_handle was invalid.

DAP_ERROR_INVALID_FILE_HANDLE
The file handle some_handle was invalid.

DAP_ERROR_PERM
The credentials established were not sufficient to allow the requested operation.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_fchown

Change the owner and group attributes of a file or directory, given an open handle to it.

```
DAP_ERROR
dap_fchown(
                DAP_HANDLE              some_handle,
                const DAP_CHAR          *owner,
                const DAP_CHAR          *group );
```

Description

This routine sets the owner and group attributes of a file or directory, given a valid handle.

Arguments

some_handle is a valid handle to a file or directory, obtained from, for example, dap_open_dir() or dap_open_file().

owner and group point to representations of the desired owner and group attributes of the file or directory indicated by some_handle. Either may be NULL, causing that parameter to be ignored.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
: The directory handle some_handle was invalid.

DAP_ERROR_INVALID_FILE_HANDLE
: The file handle some_handle was invalid.

DAP_ERROR_PERM
: The credentials established were not sufficient to allow the requested operation.

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_filesys_query

Query the attributes of a DAFS file system.

```
DAP_ERROR
dap_filesys_query(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path,
                DAP_BITMAP              attrs_requested,
                DAP_COUNT               max_byte_count,
                DAP_FILESYS_DESC        *filesys_info);
```

Description

This routine fetches the attributes of the file system indicated by path and dir_handle. The attributes desired are specified by setting bits of the attrs_requested parameter, and the dap_valid_attrs field of the DAP_FILESYS_DESC indicates which attributes were actually returned (which may be fewer than requested).

The directory handle and path together indicate the target file system to be queried. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the file system. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a valid directory handle to a directory residing somewhere within the target file system, as returned by dap_open_dir().

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is interpreted relative to dir_handle, and is a path to any valid file system object within the target file system.

attrs_requested indicates which attributes are desired; bits set to one indicate attributes that are requested.

max_byte_count is the number of bytes (octets) pointed to by filesys_info. The caller is responsible for managing this storage. This size is included to allow for future growth of the DAP_FILESYS_DESC, possibly including variably-sized fields.

filesys_info is a pointer to a DAP_FILESYS_DESC structure, to be filled in with the requested information.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE

The dir_handle supplied is not valid.

DAP_ERROR_INVALID_CRED_HANDLE
The cred_handle supplied is not valid.

DAP_ERROR_ACCESS
The supplied credential does not allow access to the path requested.

DAP_ERROR_LOOP
Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_flink

Establish a link to a file, given a handle to an open file.

```
DAP_ERROR
dap_flink(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                DAP_FILE_HANDLE         old_file,
                const DAP_CHAR          *new_path );
```

Description

The dap_flink() routine establishes a new directory entry (new_path) which points to the existing file system object indicated by old_file, an open handle. This is the only way to make visible a file created in 'unlinked' state (see dap_open_file()). The new_path must not exist, and old_file must be a valid handle to an open file, and both must lie within the same file system, as defined by the underlying server.

The old_file handle indicates the existing file which is the target of this operation. The directory handle and path together indicate the target's new name. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the target object. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a DAFS directory handle returned by the dap_open_dir() call, or NULL.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

old_file, as returned by dap_open_file(), indicates the target of the link operation. It may not reside on a different file system (as defined by the server) from new_path.

new_path indicates the new path which will refer to the target of this operation. It is also interpreted relative to dir_handle.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

See Also

See dap_open_file() and dap_open_file2() for details on opening a file in 'unlinked' state. An open file which is never linked to a pathname becomes irretrievably lost should the application close the handle, whether through deliberate action or error.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
dir_handle isn't a valid directory handle.

DAP_ERROR_INVALID_FILE_HANDLE
old_file is not a valid file handle.

DAP_ERROR_INVALID_CRED_HANDLE
The credential handle was invalid.

DAP_ERROR_PERM
The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_ACCESS
The supplied credential does not allow access to one fo the paths requested.

DAP_ERROR_INVALID_NATTR
The dir_handle indicates a named attribute directory.

DAP_ERROR_LOOP
Too many symbolic links were encountered in translating the path.

DAP_ERROR_MLINK
There are too may hard links to the target.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_fsync

Ensure that file data has reached stable storage.

```
DAP_ERROR
dap_fsync(
                DAP_FILE_HANDLE         file_handle,
                DAP_OFFSET              file_offset,
                DAP_COUNT               byte_count);
```

Description

This routine ensures that a range of previously written data to the handle "file_handle" is forced into non-volatile storage on the server(s), including the metadata describing the committed operation, if any. It is undefined whether in-flight asynchronous writes are affected by this call.

This is particularly useful in conjunction with files opened with the DAP_ASYNC flag, which allows the server latitude in scheduling disk operations by allowing the server to buffer write data and other modifications. See dap_open_file() for details.

Arguments

file_handle is a DAFS file object handle as returned by the dap_open_file() or dap_open_nattr() calls.

file_offset and byte_count indicate the range of I/O, previously completed, that must be committed to stable storage. If both are zero, the entire file indicated by file_handle is committed. It is not an error for byte_count to exceed the current size of the file.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_FILE_HANDLE
: The file handle file_handle was invalid.

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_get_acl

Fetch the access control list of a file system object.

```
DAP_ERROR
dap_get_acl(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path,
                DAP_FLAGS               flags,
                DAP_COUNT               max_byte_count,
                DAP_ACL_INFO            *aces_ptr,
                DAP_COUNT               *num_aces);
```

Description

This routine fetches the access control list associated with a file system object.

Since the returned attributes may contain a variable amount of data, the application is responsible for allocating sufficient storage, indicating the amount supplied in the max_byte_count parameter. The returned data consists of an array of fixed-size structures (of type DAP_ACL_INFO) along with the NUL-terminated strings pointed to by fields in the DAP_ACL_INFO structures.

The directory handle and path together indicate the target of this call. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the target object. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a DAFS directory handle returned from the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is interpreted relative to dir_handle, and must lead to a valid file system object.

flags consists of bit flags used to modify the behavior of this routine. Undefined bits must be zero.
The following flag bits are defined:

DAP_NO_FOLLOW
: If the final component of path is a symbolic link, this flag indicates that the link itself is the target of this operation, rather than whatever it might point to. If the final component of path is not a symbolic link, this flag is ignored.

max_byte_count is the number of bytes (octets) pointed to by aces_ptr. The caller is responsible for managing this storage, and is responsible for providing sufficient storage to hold the results.

aces_ptr points to a block of the caller's memory to be filled in with the array of DAP_ACL_INFO structures associated with the file system object indicated by path.

num_aces points to a variable that, on successful return, will indicate the number of valid DAP_ACL_INFO structures fetched. Zero indicates that there are no access control entries associated with the indicated file system object.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
: The base_dir_handle given was invalid.

DAP_ERROR_INVALID_CRED_HANDLE
: The credential handle was invalid.

DAP_ERROR_BUFFER_TOO_SMALL
: The number of bytes given was not sufficient to hold the DAP_STAT_DESC and variable-length fields. The application should try again using a larger buffer.

DAP_ERROR_PERM
: The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_ACCESS
: The supplied credential does not allow access to the path requested.

DAP_ERROR_LOOP
: Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
: The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
: The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
: The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
: The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
: One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
: The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
: A pathname component is not a directory.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_get_attr

Fetches the attributes of a file system object.

```
DAP_ERROR
dap_get_attr(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path,
                DAP_FLAGS               flags,
                DAP_BITMAP              attrs_requested,
                DAP_COUNT               max_byte_count,
                DAP_STAT_DESC           *descr_ptr);
```

Description

This routine fetches the attributes of a file system object. The attributes desired are specified by setting bits of the attrs_requested parameter, and the dap_valid_attrs field of the DAP_STAT_DESC indicates which attributes were actually returned (which may be fewer than requested).

Since the returned attributes may contain a variable amount of data, the application is responsible for allocating sufficient storage, indicating the amount supplied in the max_byte_count parameter. The returned data consists of a fixed-size structure (the DAP_STAT_DESC) and the NUL-terminated strings whose contents are pointed to by fields in the DAP_STAT_DESC.

The directory handle and path together indicate the target of this call. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the target object. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a DAFS directory handle returned from the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is interpreted relative to dir_handle, and must lead to a valid file system object.

flags consists of bit flags used to modify the behavior of this routine. Undefined bits must be zero.
The following flag bits are defined:

DAP_NO_FOLLOW
: If the final component of path is a symbolic link, this flag indicates that the link itself is the target of this operation, rather than whatever it might point to. If the final component of path is not a symbolic link, this flag is ignored.

attrs_requested indicates which attributes are desired; bits set to one indicate attributes that are requested.

max_byte_count is the number of bytes (octets) pointed to by descr_ptr. The caller is responsible for managing this storage.

descr_ptr points to a DAP_STAT_DESC structure to be filled in with the attribute information for the path given. The attributes that were actually fetched are indicated by bits in descr_ptr->dap_valid_attrs being set. Note well that a successful invocation may return a subset of those requested, if some are not supported.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
The base_dir_handle given was invalid.

DAP_ERROR_INVALID_CRED_HANDLE
The credential handle was invalid.

DAP_ERROR_BUFFER_TOO_SMALL
The number of bytes given was not sufficient to hold the DAP_STAT_DESC and variable-length fields. The application should try again using a larger buffer.

DAP_ERROR_PERM
The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_ACCESS
The supplied credential does not allow access to the path requested.

DAP_ERROR_LOOP
Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

DAP_ERROR_INVALID_ATTR
An invalid or unsupported attribute was specified.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_get_fattr

Fetches the attributes of a file system object, given an open handle to it.

```
DAP_ERROR
dap_get_fattr(
                DAP_HANDLE              some_handle,
                DAP_BITMAP              attrs_requested,
                DAP_COUNT               max_byte_count,
                DAP_STAT_DESC           *descr_ptr);
```

Description

This routine fetches the attributes of a file system object, given a valid handle to that object. The attributes desired are specified by setting bits of the attrs_requested parameter, and the dap_valid_attrs field of the DAP_STAT_DESC indicates which attributes were actually returned (which may be fewer than requested).

Since the returned attributes may contain a variable amount of data, the application is responsible for allocating sufficient storage, indicating the amount supplied in the max_byte_count parameter. The returned data consists of a fixed-size structure (the DAP_STAT_DESC) and the NUL-terminated strings whose contents are pointed to by fields in the DAP_STAT_DESC.

The directory handle and path together indicate the target of this call. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the target object. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

some_handle is a valid handle to a file or directory, obtained from, for example, dap_open_dir() or dap_open_file().

attrs_requested indicates which attributes are desired; bits set to one indicate attributes that are requested.

max_byte_count is the number of bytes (octets) pointed to by descr_ptr. The caller is responsible for managing this storage.

descr_ptr points to a DAP_STAT_DESC structure to be filled in with the attribute information for the path given. The attributes that were actually fetched are indicated by bits in descr_ptr->dap_valid_attrs being set. Note well that a successful invocation may return a subset of those requested, if some are not supported.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
The base_dir_handle given was invalid.

DAP_ERROR_INVALID_FILE_HANDLE
The base_dir_handle given was invalid.

DAP_ERROR_BUFFER_TOO_SMALL
The number of bytes given was not sufficient to hold even a single DAP_DIRENTRY with attributes and its name. The application should try again using a larger buffer.

DAP_ERROR_PERM
The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_INVALID_ATTR
An invalid or unsupported attribute was specified.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_get_fencelist

Fetch the fencing list of a file or file system.

```
DAP_ERROR
dap_get_fencelist(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path,
                DAP_FLAGS               flags,
                DAP_COUNT               max_byte_count,
                DAP_CHAR                *fence_ids_ptr[],
                DAP_COUNT               *num_fence_ids);
```

Description

This routine fetches the fencing list associated with a file system or a file system object.

Cooperating clients begin by registering the single fencing ID (an arbitrary string) which identifies them. Fencing lists are attached to file systems and file system objects, and indicate those clients that are to be allowed access. Manipulating those fencing lists then provides cooperating clients the ability to revoke a particular client’s access.

Since the returned attributes may contain a variable amount of data, the application is responsible for allocating sufficient storage, indicating the amount supplied in the max_byte_count parameter. The returned data consists of an array of fixed-size pointers (of type DAP_CHAR *) and the NUL-terminated strings pointed to by the array of pointers.

The directory handle and path together indicate the target of this call. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the target object. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a DAFS directory handle returned from the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is interpreted relative to dir_handle, and must lead to a valid file system object.

flags consists of bit flags used to modify the behavior of this routine. Undefined bits must be zero.
The following flag bits are defined:

DAP_FILESYSTEM
: If this bit is set, the fencelist for the file system underlying path is to be fetched.

DAP_NO_FOLLOW

If the final component of path is a symbolic link, this flag indicates that the link itself is the target of this operation, rather than whatever it might point to. If the final component of path is not a symbolic link, this flag is ignored.

max_byte_count is the number of bytes (octets) pointed to by fence_ids_ptr. The caller is responsible for managing this storage, and is responsible for providing sufficient storage to hold the results.

fence_ids_ptr points to a block of the caller's memory to be filled in with the array of pointers to fence IDs (which are arbitrary NUL-terminated strings) associated with the file system or file system object indicated by path.

num_fence_ids points to a variable that, on successful return, will indicate the number of valid fence IDs fetched. Zero indicates that there are no access control entries associated with the indicated file system object.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

See Also

dap_set_fenceID(), dap_set_fencelist()

Errors

DAP_ERROR_INVALID_DIR_HANDLE
The base_dir_handle given was invalid.

DAP_ERROR_INVALID_CRED_HANDLE
The credential handle was invalid.

DAP_ERROR_BUFFER_TOO_SMALL
The number of bytes given was not sufficient to hold the DAP_STAT_DESC and variable-length fields. The application should try again using a larger buffer.

DAP_ERROR_PERM
The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_ACCESS
The supplied credential does not allow access to the path requested.

DAP_ERROR_LOOP
Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE

The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

DAP_ERROR_NOT_SUPPORTED
Fencing is not supported.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_io_done

Checks the status of an async I/O operation.

```
DAP_ERROR
dap_io_done(
                DAP_IO_RESULT           *io_desc );
```

Description

Checks the status of an asynchronous I/O operation and returns immediately without waiting. Functionally equivalent to a call to dap_io_wait() with an instantaneous timeout (DAP_WAIT_NOWAIT). If the I/O has completed, this routine returns zero and the structure pointed to by io_desc has been filled in with the results for the completed operation.

This routine may be used to reap completions only from those I/O operations that were initiated using a completion group handle of NULL.

Arguments

io_desc is the pointer to the DAP_IO_RESULT for the asynchronous I/O operation being checked for completion. These are the structure(s) that were supplied to the call originating the asynchronous operations (dap_async_read(), dap_async_write(), dap_async_listio(), etc.).

Returns

Returns zero on success. Otherwise returns one of the error values listed below. An error from the asynchronous operation may be contained in the dap_error field of the DAP_IO_RESULT.

Errors

DAP_ERROR_PENDING_IO
The asynchronous I/O operation is still in progress. Try again later.

DAP_ERROR_NO_IO_PENDING
There have been no I/O operations initiated using this DAP_IO_RESULT.

DAP_ERROR_CG_INVALID
It is invalid to attempt to wait on a DAP_IO_RESULT that has had completions directed to a completion group.

DAP_ERROR_IO_CANCELLATION
The operation associated with this DAP_IO_RESULT was successfully cancelled. This error will only appear in io_desc->dap_error.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_io_wait

Block for an asynchronous I/O operation.

```
DAP_ERROR
dap_io_wait(
                DAP_TIMEOUT             timeout,
                DAP_IO_RESULT           *io_desc );
```

Description

Blocks for up to a duration of "timeout" awaiting the I/O completion indicated by io_desc. If timeout is set to DAP_WAIT_NOWAIT, it checks but does not block. If timeout is set to DAP_WAIT_FOREVER, it will block until an I/O completes before returning. If the I/O has completed, this routine returns zero and the structure pointed to by io_desc has been filled in with the results for the completed operation.

This routine may be used to reap completions only from those I/O operations that were initiated using a completion group handle of NULL.

Arguments

timeout indicates how long to wait for an I/O completion before returning DAP_ERROR_PENDING_IO. If timeout is equal to DAP_WAIT_NOWAIT this call is equivalent to dap_io_done(). If timeout is equal to DAP_WAIT_FOREVER it will block until an I/O completes.

io_desc is the pointer to the DAP_IO_RESULT for the asynchronous I/O operation being checked for completion. These are the structure(s) that were supplied to the call originating the asynchronous operations (dap_async_read(), dap_async_write(), dap_async_listio(), etc.).

Returns

Returns zero on success. Otherwise returns one of the error values listed below. An error from the asynchronous operation may be contained in the dap_error field of the DAP_IO_RESULT.

Errors

DAP_ERROR_PENDING_IO
: The timeout expired before I/O completion.

DAP_ERROR_NO_IO_PENDING
: There have been no I/O operations initiated using this DAP_IO_RESULT.

DAP_ERROR_CG_INVALID
: It is invalid to attempt to wait on a DAP_IO_RESULT that has had completions directed to a completion group.

DAP_ERROR_IO_CANCELLATION
: The operation associated with this DAP_IO_RESULT was successfully cancelled. This error will only appear in io_desc->dap_error.

DAP_ERROR_TRANSPORT_FAILURE

A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_link

Establish a link to a file

```
DAP_ERROR
dap_link(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *old_path,
                const DAP_CHAR          *new_path );
```

Description

The dap_link() routine establishes a new directory entry (new_path) which points to the existing file system object indicated by old_path. The new_path must not exist, and old_path must exist, and both must lie within the same file system, as defined by the underlying server.

The directory handle and paths together indicate the targets of the dap_link() call. If absolute paths are given, then the directory handle may be NULL; the DAFS name service will be used to locate the target objects. If the paths are relative, they are interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a DAFS directory handle returned by the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

old_path indicates the target of the link operation. It is interpreted relative to dir_handle, and must lead to a valid file system object. It may not be a directory, and may not reside on a different file system (as defined by the server) from new_path.

new_path indicates the new path which will refer to the target of this operation. It is also interpreted relative to dir_handle.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
: dir_handle isn't a valid directory handle.

DAP_ERROR_INVALID_CRED_HANDLE
: The credential handle was invalid.

DAP_ERROR_PERM
: The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_ACCESS
The supplied credential does not allow access to one of the paths requested.

DAP_ERROR_INVALID_NATTR
The dir_handle indicates a named attribute directory.

DAP_ERROR_LOOP
Too many symbolic links were encountered in translating the path.

DAP_ERROR_MLINK
There are too may hard links to the target.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_lock_range

Attempt an advisory read/write lock on a range of bytes.

```
DAP_ERROR
dap_lock_range(
                DAP_FILE_HANDLE         file_handle,
                DAP_OFFSET              byte_offset,
                DAP_LENGTH              byte_length,
                DAP_LOCK_TYPE           lock_type,
                unsigned int            lock_options,
                DAP_TIMEOUT             how_long );
```

Description

This routine attempts to take a record lock for the range of bytes requested, against the file indicated by file_handle. The locking model is one of advisory read/write locks. In other words, there may be multiple readers, but only one writer, and not both at the same time. That the locks are advisory means that a successful locking attempt inhibits further locking attempts, but does not prevent I/O operations from taking place.

Bytes in a file may be locked even if those bytes are not currently allocated to the file. To lock the file from a specific offset through the end-of-file (no matter how long the file actually is) use a byte_length field will all bits set to 1 (one). To lock the entire file, use a byte_offset of 0 (zero) and a byte_length with all bits set to 1. A byte_length of zero should not be used.

Locks may be upgraded (from read-lock to write-lock) and downgraded (from write-lock to read-lock) by performing an additional dap_lock_range() operation of the appropriate type on the same byte range. Locking is done on a per-DAP_FILE_HANDLE basis. This implies that multiple locking operations may be performed successfully by the current lock holder, but does not imply that multiple dap_unlock_range() operations are necessary. At most one lock per DAP_FILE_HANDLE is held on a given byte range, so that a single dap_unlock_range() undoes all previous dap_lock_range() operations for that range. Thus multiple clients threads operating on the same DAP_FILE_HANDLE cannot use this locking mechanism to provide mutual exclusion, though they may obtain distinct DAP_FILE_HANDLEs if necessary.

It is implementation dependent whether a client may request a lock with one byte range and then either upgrade or unlock a sub-range of the initial lock. Likewise, it is implementation dependent whether a client may lock two adjacent byte ranges or two overlapping byte ranges and then upgrade or unlock the entire range or a subset spanning parts of both prior locking operations. The caller must be prepared for DAP_ERROR_LOCK_RANGE to be returned in these cases.

For a more detailed explanation of auto-recover and persistent lock behavior, see the protocol specification.

Arguments

file_handle is a DAFS file object as returned by the dap_open_file() call.

byte_offset indicates the offset of the first byte of the range to be locked. Zero indicates the initial byte.

byte_length is the number of bytes to be locked, with a value of all one bits indicating "everything."

lock_type indicates the type of locking desired. Possible values are:

- o DAP_LOCK_TRY_READ - try lock for reading once
- o DAP_LOCK_TRY_WRITE - try lock for writing once
- o DAP_LOCK_READ - blocking read lock attempt
- o DAP_LOCK_WRITE - blocking write lock attempt
- o DAP_LOCK_ABORT - roll-back auto-recover lock

lock_options indicates the type of locking requested, and provides for handling of these extra features. Support for these is optional, and other lock_options bits should be zero.

- o DAP_LOCK_OPT_PERSIST - get a persistent lock
- o DAP_LOCK_OPT_AUTOREC - get an autorecover lock
- o DAP_LOCK_OPT_REPAIR - re-take broken persist lock

how_long indicates how long the caller is willing to wait for the lock to become available. If the lock_type does not indicate a blocking lock attempt, this parameter is ignored.

## Returns

Returns zero on success. Otherwise returns one of the error values listed below.

## Errors

DAP_ERROR_INVALID_FILE_HANDLE
: file_handle isn't a valid DAFS file handle.

DAP_ERROR_LOCK_DENIED
: this lock attempt conflicts with a held lock

DAP_ERROR_TIMED_OUT
: The lock request was not satisfied in the time period specified.

DAP_ERROR_LOCK_BROKEN
: This locking attempt was made to a persist lock whose lease has expired.

DAP_ERROR_LOCK_RANGE
: Locking of sub-ranges or overlapping ranges is not supported.

DAP_ERROR_NOT_SUPPORTED
: An unsupported feature was requested.

DAP_ERROR_BAD_ARG
: An argument was invalid (for example, byte_length of zero).

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_make_dev

Create a special file

```
DAP_ERROR
dap_make_dev(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path,
                DAP_FILETYPE            type,
                DAP_CREATE_MODE         mode,
                const DAP_SPECDATA      *spec_data );
```

Description

The dap_make_dev() routine creates a special file of the type and mode specified. Only block devices, character devices, FIFOs, and sockets may be created using this interface; regular files are created with dap_open_file(), directories with dap_open_dir(), symbolic links with dap_symlink(), and named attributes with dap_open_nattr().

The directory handle and path together indicate the target of this call. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the target object. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a DAFS directory handle returned by the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is an arbitrary path name. Each component except the last must be a directory, and the last component must not exist.

type specifies the type of the special file to be created, and must be one of DAP_BLOCK_DEV, DAP_CHAR_DEV, DAP_SOCKET, or DAP_FIFO.

mode specifies the mode of the special file to be created.

spec_data points to the special data to be associated with the created special file. This parameter is ignored for sockets and FIFOs, and may be NULL. For character and block special files, spec_data encodes the major/minor numbers; it may be examined with dap_get_attr().

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

See Also

dap_get_attr(), dap_get_fattr(), dap_open_file(), dap_open_dir(), dap_open_nattr(), dap_symlink(), dap_remove().

Errors

DAP_ERROR_INVALID_DIR_HANDLE
: dir_handle isn't a valid directory handle.

DAP_ERROR_INVALID_CRED_HANDLE
: The credential handle was invalid.

DAP_ERROR_PERM
: The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_BAD_ARG
: The type or spec_data or mode specified is invalid or not supported.

DAP_ERROR_ACCESS
: The supplied credential does not allow access to the path requested.

DAP_ERROR_FILE_EXISTS
: The target name already exists.

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

DAP_ERROR_LOOP
: Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
: The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
: The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
: The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
: The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
: One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
: The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
: A pathname component is not a directory.

Other DAP_ERROR values may also be returned.

dap_open_dir

Opens a directory, possibly creating it.

```
DAP_ERROR
dap_open_dir(
                DAP_DIRECTORY_HANDLE    base_dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path,
                DAP_FLAGS               flags,
                DAP_CREATE_MODE         dap_mode,
                DAP_DIRECTORY_HANDLE    *dir_handle);
```

Description

This routine will open the directory indicated by path. If DAP_CREATE is specified in the flags, the directory will be created if it does not already exist, otherwise an error will be returned.

The directory handle and path together indicate the target directory to be opened. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the server(s) containing the directory. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

base_dir_handle is a DAFS directory handle returned from the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is interpreted relative to base_dir_handle, and must lead to a file system object that is either a valid directory or does not exist. All non-terminal components of the path must be directories.

flags consists of bit flags used to modify the behavior of this routine. Undefined bits must be zero. The following flag bits are defined:

DAP_CREATE
: If DAP_CREATE is specified, and "path" does not already exist, the directory will be created; otherwise an error will be returned.

DAP_NATTR_DIR
: If this flag is used, the named attribute directory associated with path (which in this case may indicate any file or directory) is opened. The resulting directory handle may then used in conjunction with any of the directory reading routines to determine the list of named attributes associated with path. dap_open_nattr() may then be used to access the contents of the attribute.

DAP_NONBLOCK
: If this flag is set, attempts to initiate asynchronous

I/O that would delay the calling thread return DAP_ERROR_WOULD_BLOCK; if this flag is not set, attempts to initiate asynchronous I/O may block as necessary due to issues of resource exhaustion or flow control.

dap_mode is ignored unless the directory is being created, in which case it governs the mode of the new directory. Mode details are platform-dependent.

dir_handle is a pointer to a DAP_DIRECTORY_HANDLE, which upon successful completion will contain a handle to the requested directory.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
The base_dir_handle given was invalid.

DAP_ERROR_INVALID_CRED_HANDLE
The credential handle was invalid.

DAP_ERROR_PERM
The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_INVALID_FLAGS
The flags bits were invalid.

DAP_ERROR_FILE_EXISTS
The path refers to a existing file, not a directory.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

DAP_ERROR_DIRECTORY
Directory creation was requested, but the directory already exists.

DAP_ERROR_ACCESS
The supplied credential does not allow access to the path requested.

DAP_ERROR_LOOP
Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_open_file

Opens a file, obtaining a handle to it.

```
DAP_ERROR
dap_open_file(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path,
                DAP_FLAGS               flags,
                DAP_CREATE_MODE         mode,
                DAP_SHARE_KEY           share_key ,
                DAP_FILE_HANDLE         *file_handle);
```

Description

This routine opens the file indicated by path, which is interpreted relative to dir_handle.

The directory handle and path together indicate the target file object to be opened. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the file. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

This function always opens the indicated file; in other words, if the path given is a symbolic link, "follow" behavior is implemented. The dap_close_file() routine is the inverse of this function.

Arguments

dir_handle is a DAFS directory object returned by the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is interpreted relative to dir_handle, and must lead to a file system object that is either a valid file or does not exist. All non-terminal components of the path must be directories.

flags specifies the type of access to the file. An application can obtain read access, write access, or read/write access qualified by other flags bitwise or-ed together. Any unsupported flag bits must be zero, and this parameter must include at least one of the following values:

DAP_READ
: Open file for reading.

DAP_WRITE
: Open file for writing.

DAP_READ_WRITE
: Open file for both reading and writing (shorthand for 'DAP_READ | DAP_WRITE').

Qualifier flags which are bitwise or-ed to the above are:

DAP_CREATE
If the file already exists, this flag has no effect unless the DAP_EXCLUSIVE bit is also set, in which case the open will fail with DAP_ERROR_FILE_EXISTS. Otherwise the file is created.

DAP_UNLINKED
Valid only in combination with the DAP_CREATE flag, this bit indicates that the file should be created in 'unlinked' mode, meaning that it will not be immediately made visible in the DAFS name space. The resulting file handle must be used with dap_flink() to provide a name and force visibility, else the file will be silently deleted when the handle is closed.

DAP_EXCLUSIVE
Valid only in combination with the DAP_CREATE flag, this bit forces an error if the requested file already exists (avoiding the possibility of inadvertently sharing this file with another DAFS client due to dueling creation attempts).

DAP_APPEND
If set, all write I/O is appended to the end of the file; read operations are not affected. Note that append-mode writes larger than dap_filesys_desc.dap_max_append bytes may fail without any data having been written. The minimum supported append size will be at least 64KB. NB: this flag has no effect upon dap_async_listio().

DAP_TRUNCATE
If the file exists, and the file is successfully opened with the DAP_WRITE bit set its length is truncated to 0.

DAP_NONBLOCK
If this flag is set, attempts to initiate asynchronous I/O that would delay the calling thread return DAP_ERROR_WOULD_BLOCK; if this flag is not set, attempts to initiate asynchronous I/O may block as necessary due to issues of resource exhaustion or transport flow control. NB: this flag has no effect upon dap_async_listio().

DAP_SEQUENTIAL
This bit flag is a hint to the DAFS provider that the application expects its access to the file being opened to be sequential.

DAP_SEQ_REVERSE
This hint is like DAP_SEQUENTIAL, but in reverse order.

DAP_RANDOM
This bit flag is a hint to the DAFS provider that the application expects its access to the file being opened to be random.

DAP_BUFFERED
This bit flag allows the DAFS server latitude in scheduling disk operations by allowing the server to buffer write data and other modifications. It may decrease response latency, but necessitates

the use of dap_fsync() by applications. The default behavior is that writes are unbuffered (so no use of dap_fsync() is necessary).

DAP_SHAREKEY
This bit flag indicates that the share_key parameter is intended to be used to control access to the target file. If this bit is off, the share_key parameter will be ignored.

DAP_SHARE_DENY_RD
DAP_SHARE_DENY_WR
DAP_SHARE_DENY_BOTH
These bits flags indicate that values used to control shared access by denying other clients the ability to read, write, or both. See 'Extended Sharing Semantics' below for the behavioral details. Not all combinations are valid; each of the parameters may be set to RD, WR, or BOTH.

DAP_NO_DELETE
This bit indicates that deletions are to be denied while this file handle is held open.

mode is the creation mode of the file. These bits specify the permissions of the file being created, POSIX style. On non-POSIX-compatible systems these bits may be ignored or treated differently.

file_handle is a pointer to a DAP_FILE_HANDLE which upon successful return, will contain a handle to the requested file, useful for performing I/O.

share_key is the shared key bit pattern used to control clustered access to the file being opened, iff DAP_SHAREKEY is set in flags. See 'Extended Sharing Semantics' below for the behavioral details.

Returns

Returns zero on success. Otherwise returns one of the error values listed below. The file handle resulting from this call is contained in the handle pointed to by file_handle.

Extended Sharing Semantics

The share_key reservation is provided to aid a clustered application to detect rogue instances that are trying to perform conflicting access to a file. This reservation allows a clustered application to have all components of a cluster instance share a reservation. This is in addition to the NFSv4-style share semantics, which provide the ability to deny other clients read access, write access, or both.

The following pseudo-code describes the algorithm implemented:

```
    if ((request.access & file_state.share_deny) ||
        (request.share_deny & file_state.access)) {
            /*
             * NFS-style failure
             */
            return( DAP_ERROR_DENIED );

    } else if ((flags & DAP_SHAREKEY) &&
                file_state.flags & DAP_SHAREKEY) {
            /*
```

```
                     * Neither sharekey bit is cleared, so
                     * check the sharekeys
                     */
                    if (share_key != file_state.share_key)
                        return(DAP_ERROR_KEY_MISMATCH);
                    else
                        file_state.share_key_reference_count++;
                }
```

The dap_close_file() interface decrements file_state.share_key_reference_count if a share key is held. When the count goes to zero, the DAP_SHAREKEY bit is cleared, and the shared key, access, and deny fields are set to zero.

See Also

Use dap_async_read_link() to open and read a symbolic link.

Use dap_flink() to link a file opened in 'unlinked' state to a pathname so that it will be visible to other clients. An open file which is never linked to a pathname becomes irretrievably lost should the application close the handle, whether through deliberate action or error.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
: dir_handle isn't a valid directory handle.

DAP_ERROR_INVALID_CRED_HANDLE
: The credential handle was invalid.

DAP_ERROR_PERM
: The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_FILE_EXISTS
: Request to exclusively create file was made (flags DAP_EXCLUSIVE and DAP_CREATE), but file already exists.

DAP_ERROR_DIRECTORY
: The path refers to a directory, not a file.

DAP_ERROR_ACCESS
: The supplied credential does not allow access to the path requested.

DAP_ERROR_INVALID_FLAGS
: An invalid combination of flag options was requested, or the feature is not supported.

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

DAP_ERROR_NOT_SUPPORTED
: extended sharing semantics are not supported by the server holding the indicated file.

DAP_ERROR_DENIED
: extended sharing failure due to NFSv4-style share_access and share_deny conflict.

DAP_ERROR_LOOP
: Too many symbolic links were encountered in

translating the path.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

DAP_ERROR_KEY_MISMATCH
extended sharing failure due to share_key conflict.

Other DAP_ERROR values may also be returned.

dap_open_nattr

Open the named attribute associated with a file system object.

```
DAP_ERROR
dap_open_nattr(
                DAP_DIRECTORY_HANDLE    nattr_dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *attr_name,
                DAP_FLAGS               flags,
                DAP_FILE_HANDLE         *file_handle );
```

Description

This routine opens the named attribute indicated by attr_name, contained in the named attribute directory indicated by nattr_dir_handle, which was obtained from a successful call to dap_open_dir() with the DAP_NATTR_DIR flag.

The contents of the named attribute may then be read using dap_async_dap_async_read() upon the handle returned by this routine.

The dap_close_file() routine is the inverse of this function.

Arguments

nattr_dir_handle is a non-NULL directory handle indicating a named attribute directory, obtained from a successful call to dap_open_dir() using the DAP_NATTR_DIR flag.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

attr_name is the name of the attribute to be opened and read or written.

flags specifies the type of access to the attribute. An application can obtain read access, write access, or read/write access qualified by other flags bitwise or-ed together. Undefined flag bits must be zero. This parameter must include at least one of the following values:

DAP_READ
: Open the named attribute for reading.

DAP_WRITE
: Open the named attribute for writing.

DAP_READ_WRITE
: Open the named attribute for both reading and writing (shorthand for 'DAP_READ | DAP_WRITE').

Qualifier flags which are bitwise or-ed to the above are:

DAP_CREATE
: If the named attribute already exists, this flag has no effect unless the DAP_EXCLUSIVE bit is also set, in which case the open will fail with DAP_ERROR_FILE_EXISTS. Otherwise the named attribute is created.

DAP_EXCLUSIVE
If the DAP_CREATE bit is not also set, this flag has no effect

DAP_APPEND
If set, all write I/O is appended to the end of the named attribute; read operations are not affected. Note that append-mode writes larger than dap_filesys_desc.dap_max_append bytes may fail without any data having been written. The minimum supported append size will be at least 64KB. NB: this flag has no effect upon dap_async_listio().

DAP_TRUNCATE
If the attribute exists, and it is successfully opened with the DAP_WRITE bit set its contents are truncated to be zero length.

DAP_NONBLOCK
If this flag is set, attempts to initiate asynchronous I/O that would delay the calling thread return DAP_ERROR_WOULD_BLOCK; if this flag is not set, attempts to initiate asynchronous I/O may block as necessary due to issues of resource exhaustion or flow control. NB: this flag has no effect upon dap_async_listio().

file_handle is a pointer to a DAP_FILE_HANDLE which upon successful return, will contain a handle to the requested named attribute, useful for reading (or writing) its contents.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
nattr_dir_handle isn't a valid named attribute directory handle.

DAP_ERROR_INVALID_CRED_HANDLE
The credential handle was invalid.

DAP_ERROR_PERM
The credential supplied was not sufficient to allow the requested operation.

Other errors are reflected in the io_error field of the DAP_IO_RESULT structure:

DAP_ERROR_FILE_EXISTS
Request to exclusively create the attribute was made (flags DAP_EXCLUSIVE and DAP_CREATE), but it already exists.

DAP_ERROR_ACCESS
The supplied credential does not allow access to the attribute requested.

DAP_ERROR_INVALID_FLAGS
The server does not support atomic append mode, and DAP_APPEND was specified in the flags.

DAP_ERROR_INVALID_NATTR

The named attribute does not exist, or the name is invalid, possibly due to a server-specific limit in length or character set.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_read

Synchronous file read operation.

```
DAP_ERROR
dap_read(
                DAP_FILE_HANDLE         file_handle,
                DAP_OFFSET              file_offset,
                DAP_COUNT               io_count,
                const DAP_MEM_DESC      *mem_desc,
                DAP_LENGTH              *done_count );
```

Description

Initiates one read operation, returning control upon completion. An attempt is made to read data from the file referenced by the file handle at an offset of file_offset into the buffer or buffers indicated by mem_desc.

The number of bytes read is returned in the variable pointed to by done_count.

Arguments

file_handle is a DAFS file handle as returned by the dap_open_file() or dap_open_nattr() calls.

file_offset is the offset in the file from which to read data.

io_count is the number of sequential DAP_MEM_DESC structures, and must be greater than zero.

mem_desc is pointer to a (vector of) descriptor(s) for the I/O operation. Each entry in the vector contains:

dap_mem_handle - a DAFS memory handle that is associated with the buffer pointer and length. If DAP_NULL_MEM_HANDLE is supplied, the provider will register and bind the memory on the fly; it may cache these mappings to speed later operations.

dap_bufferp - a buffer pointer to somewhere within the registered memory region referred to by the DAFS memory handle.

dap_buffer_len - the length in bytes of the buffer.

done_count points to a variable which upon successful return contains the number of bytes transferred.

Returns

Returns zero on success, with the number of bytes read being returned in the variable pointed to by done_count. Otherwise, one of the error values below may be returned.

Errors

DAP_ERROR_INVALID_FILE_HANDLE
file_handle isn’t a valid DAFS file object.

DAP_ERROR_INVALID_MEM_HANDLE
Some entry in the mem_desc has an invalid registered

memory handle.

DAP_ERROR_BAD_ARG
The io_count was less than or equal to zero.

DAP_ERROR_UNREGISTERED_MEM
Some entry in the DAP_MEM_DESC is not valid. Either the dap_bufferp is not within a valid registered memory virtual region, or the end of the buffer extends beyond the memory region referred to by the memory handle, or a NULL dap_mem_handle was given and the Provider was unable to register the memory region on the fly.

DAP_ERROR_IO_OVERLAP
This request attempts to write to an area that overlaps a pending write request, possibly leading to undefined results due to the lack of ordering guarantees among simultaneous pending I/O requests.

DAP_ERROR_IO
There was a hard and unrecoverable media (disk) error.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_read_dir

Read some number of directory entries, synchronously.

```
DAP_ERROR
dap_read_dir(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_OFFSET              cookie,
                DAP_OPAQUE              cookie_verifier,
                DAP_MEM_HANDLE          mem_handle,
                DAP_LENGTH              size,
                DAP_READDIR_RESULT      *resultp);
```

Description

This routine reads some number of entries from the directory indicated by dir_handle, synchronously. The application provides some number of bytes of (registered) memory and a cookie of zero to begin reading a directory. The provider will return a DAP_READDIR_RESULT structure, which contains a cookie verified (see below), a flag indicating when the last item has been read, the number of items read, and a vector of DAP_DIRENTRY structures containing the actual directory data. The directory data consists of a number of fixed-size DAP_DIRENTRY structures, each containing the DAP_FILETYPE of the file system object, a pointer to its NUL-terminated name and an opaque cookie to be passed to a subsequent call to dap_async_read_dir() et al. to access the remaining directory entries.

There are no further entries to be read from the directory indicated by dir_handle when the dap_end_flag is set in the DAP_READDIR_RESULT structure.

Arguments

dir_handle is a DAFS directory handle returned from the dap_open_dir() call, and indicates the directory which is to be read.

cookie is a value that represents where the operation should start within the directory. A value of 0 (zero) for the cookie is used to start reading at the beginning of the directory. For subsequent requests, the caller specifies a cookie value that is provided by the server in response to a previous request (dap_readdir_result.dap_entry[index].dap_direntry_cookie).

cookie_verifier should be set to 0 (zero) when the cookie value is 0 (zero) on the first directory read. On subsequent requests, it should be a cookieverf as returned by the server (dap_readdir_result.dap_cookiev). The cookieverf must match that returned by the read operation in which the cookie was acquired.

mem_handle is a DAFS memory handle that is associated with the application buffer pointed to by resultp, and may be NULL.

size is the length in bytes of the application buffer pointed to by resultp.

resultp points to the application buffer, a variable-size

DAP_READDIR_RESULT structure. Upon successful completion, this contains:

dap_cookiev - the cookie verifier returned by the server

dap_end_flag - set to non-zero when the last entry in the directory has been read.

dap_num_entries - the number of valid entries in the variable-size dap_entry array.

dap_entry - the output array of DAP_DIRENTRY structures, containing dap_num_entries valid members. Each contains:

dap_direntry_type - indicating the DAP_FILETYPE of this entry.

dap_direntry_cookie - an opaque cookie to be handed to a subsequent call to any of the directory reading routines in order to obtain the next DAP_DIRENTRY.

dap_direntry_name - a pointer into this application-managed storage, to the NUL-terminated name of this file system object.

dap_direntry_attrp - a pointer into this application-managed storage, to the requested attributes of this file system object. This pointer will always be NULL when the attributes are obtained with this interface (see dap_async_read_dir2() and dap_read_dir2()).

Returns

Returns zero on success, else one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
The base_dir_handle given was invalid.

DAP_ERROR_BADCOOKIE
The cookier/cookie_verifier pair supplied was invalid.

DAP_ERROR_BUFFER_TOO_SMALL
The number of bytes given was not sufficient to hold even a single DAP_DIRENTRY and its name. The application should try again using a larger buffer.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_read_dir2

Read some number of directory entries and their attributes, synchronously.

```
DAP_ERROR
dap_read_dir2(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_OFFSET              cookie,
                DAP_OPAQUE              cookie_verifier,
                DAP_BITMAP              attrs_requested,
                DAP_MEM_HANDLE          mem_handle,
                DAP_LENGTH              size,
                DAP_READDIR_RESULT      *resultp,
                DAP_LENGTH              *done_count);
```

Description

This routine reads some number of entries and their attributes from the directory indicated by dir_handle. The application provides some number of bytes of (registered) memory and a cookie of zero to begin reading a directory. The provider will return a DAP_READDIR_RESULT structure, which contains a cookie verified (see below), a flag indicating when the last item has been read, the number of items read, and a vector of DAP_DIRENTRY structures containing the actual directory data. The directory data consists of a number of fixed-size DAP_DIRENTRY structures, each containing the DAP_FILETYPE of the file system object, a pointer to its NUL-terminated name and an opaque cookie to be passed to a subsequent call to dap_async_read_dir() et al. to access the remaining directory entries.

There are no further entries to be read from the directory indicated by dir_handle when the dap_end_flag is set in the DAP_READDIR_RESULT structure.

Arguments

dir_handle is a DAFS directory handle returned from the dap_open_dir() call, and indicates the directory which is to be read.

cookie is a value that represents where the operation should start within the directory. A value of 0 (zero) for the cookie is used to start reading at the beginning of the directory. For subsequent requests, the caller specifies a cookie value that is provided by the server in response to a previous request (dap_readdir_result.dap_entry[index].dap_direntry_cookie).

cookie_verifier should be set to 0 (zero) when the cookie value is 0 (zero) on the first directory read. On subsequent requests, it should be a cookieverf as returned by the server (dap_readdir_result.dap_cookiev). The cookieverf must match that returned by the read operation in which the cookie was acquired.

attrs_requested indicates which attributes are desired; bits set to one indicate attributes that are requested.

mem_handle is a DAFS memory handle that is associated with the application buffer pointed to by resultp, and may be

NULL.

size is the length in bytes of the application buffer pointed to by resultp.

resultp points to the application buffer, a variable-size DAP_READDIR_RESULT structure. Upon successful completion, this contains:

dap_cookiev - the cookie verifier returned by the server

dap_end_flag - set to non-zero when the last entry in the directory has been read.

dap_num_entries - the number of valid entries in the variable-size dap_entry array.

dap_entry - the output array of DAP_DIRENTRY structures, containing dap_num_entries valid members. Each contains:

dap_direntry_type - indicating the DAP_FILETYPE of this entry.

dap_direntry_cookie - an opaque cookie to be handed to a subsequent call to any of the directory reading routines in order to obtain the next DAP_DIRENTRY.

dap_direntry_name - a pointer into this application-managed storage, to the NUL-terminated name of this file system object.

dap_direntry_attrp - a pointer into this application-managed storage, to the requested attributes of this file system object.

Returns

Returns zero on success, else one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
The base_dir_handle given was invalid.

DAP_ERROR_BADCOOKIE
The cookier/cookie_verifier pair supplied was invalid.

DAP_ERROR_BUFFER_TOO_SMALL
The number of bytes given was not sufficient to hold even a single DAP_DIRENTRY with attributes and its name. The application should try again using a larger buffer.

DAP_ERROR_IO
There was a hard and unrecoverable media (disk) error.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_read_link

Read a symbolic link.

```
DAP_ERROR
dap_read_link(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path,
                DAP_COUNT               buffer_size,
                DAP_CHAR                *buffer );
```

Description

This routine reads the contents of a symbolic link into a caller-supplied buffer. The contents will be NUL-terminated, and an error is returned if the supplied buffer is too small, so buffer_size should be at least one plus the size of the link.

The directory handle and path together indicate the target of this call. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the target object. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a DAFS directory object returned by the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is interpreted relative to dir_handle, and must lead to a symbolic link (in other words, the final component of path must be a symbolic link - if a preceding component is a link than it is followed). The contents of the link (rather than whatever object it might point to) are read.

buffer_size indicates the size in bytes of the buffer supplied by the caller.

buffer is a pointer to buffer_size bytes in the caller’s memory, where the contents of the link will be placed. The bytes read will be NUL-terminated.

Returns

Returns zero on success, with a NUL-terminated series of bytes placed in the memory pointed to by buffer. Otherwise, one of the error values below may be returned.

See Also

dap_get_attr( flags=DAP_STAT_OBJECT_SIZE ) may be used to determine the number of bytes to be read from the link.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
The directory handle was invalid.

DAP_ERROR_BUFFER_TOO_SMALL,
The buffer supplied was insufficient to hold the contents of the link. Use dap_get_attr() to determine the necessary size and try again.

DAP_ERROR_ACCESS
The supplied credential does not allow access to the path requested.

DAP_ERROR_LOOP
Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

DAP_ERROR_IO
There was a hard and unrecoverable media (disk) error.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_register_mem

Registers a memory region with the DAFS library.

```
DAP_ERROR
dap_register_mem(
                DAP_PVOID               buffer,
                DAP_LENGTH              length,
                DAP_MEM_HANDLE          *mem_handle );
```

Description

This routine registers a region of the caller's address space with the DAFS provider returning a memory handle to be used when initiating I/O operations.

Arguments

buffer is a pointer to the memory buffer.

length is the size in bytes of the buffer.

mem_handle is a pointer to a DAP_MEM_HANDLE to be returned on successful return. This handle is used to identify the underlying memory when initiating I/O operations.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

See Also

dap_deregister_mem() is the inverse of this operation.

Errors

DAP_ERROR_NO_RESOURCES
Insufficient resources exist to complete this operation.

DAP_ERROR_INVALID_ADDRESS
The {buffer, length} pair is invalid, possibly because it is not contained within the caller's accessible virtual address space.

Other DAP_ERROR values may also be returned.

dap_register_shbuffer

Register a shared memory region with the DAFS library.

```
DAP_ERROR
dap_register_shbuffer(
                DAP_PVOID               buffer,
                DAP_LENGTH              length,
                DAP_SHBUFF_KEY          key,
                DAP_FLAGS               flags,
                DAP_MEM_HANDLE          *mem_handle );
```

Description

This routine registers a region of the caller's address space with the DAFS provider, associating the specified key with that region of memory, and returning a memory handle to be used when initiating I/O operations. This region is presumed to be shared (or shareable) between client processes. The key is used as a tag for the memory region, permitting the DAFS provider to optimize use of both provider and transport resources during registrations subsequent to the first. The flags allow the caller to exclusively associate the specified key with the memory region, receiving an error if the key is already in use.

As with dap_register_mem(), the {buffer, length} pair need not be aligned, but subsequent callers of dap_register_shbuffer() providing the same key (hence registering the same shared region) must provide a buffer that is congruent with that given by the first caller. The exact method used to cause sharing of the buffer among DAFS clients is dependent on the details of the host system, and is the responsibility of the caller (e.g.: shmget() and shmat() on UNIX systems).

Arguments

buffer is a pointer to the memory buffer.

length is the size in bytes of the buffer.

key is a DAP_SHBUFF_KEY used to tag the memory buffer.

flags consists of bit flags used to modify the behavior of this routine. Undefined bits must be zero.
The following flag bits are defined:

DAP_CREATE
: If the specified key has not been associated with a (shared) memory region, this association is made. If the key is already in use, this flag has no effect unless the DAP_EXCLUSIVE bit is also set, in which case the operation will fail with DAP_ERROR_FILE_EXISTS.

DAP_EXCLUSIVE
: Valid only in combination with the DAP_CREATE flag, this option forces an error if the requested key is already in use (avoiding the possibility of inadvertently sharing the memory of another DAFS client due to dueling key registration attempts).

mem_handle is a pointer to a DAP_MEM_HANDLE to be returned

on successful return. This handle is used to identify the underlying memory when initiating I/O operations.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

See Also

dap_deregister_mem() is the inverse of dap_register_shbuffer(); each process must deregister its own memory, and that operation will have no effect upon other processes.

dap_register_mem() performs the same task albeit without the potential for optimizing global resource usage.

Errors

DAP_ERROR_NO_RESOURCES
: Insufficient resources exist to complete this operation.

DAP_ERROR_INVALID_ADDRESS
: The {buffer, length} pair is invalid, possibly because it is not contained within the caller's virtual address space, or is of an unsupported type (not obtained through one of the supported host-specific methods) or is not congruent with the region already associated with the key provided.

DAP_ERROR_INVALID_KEY
: The key provided is invalid (NULL or not currently in use and DAP_CREATE was not specified).

DAP_ERROR_FILE_EXISTS
: The key is already in use and the caller specified flags DAP_EXCLUSIVE and DAP_CREATE, requesting exclusive creation of the association between the key and the memory region.

DAP_ERROR_INVALID_FLAGS
: An invalid combination of flag options was requested.

Other DAP_ERROR values may also be returned.

dap_remove

Remove a directory entry.

```
DAP_ERROR
dap_remove(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path );
```

Description

This routine removes the directory entry specified by path.

Arguments

dir_handle is a DAFS directory handle returned by the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is interpreted relative to dir_handle, and must lead to a valid file system object. That object must not be a non-empty directory. If the final component of path is a symbolic link, the link itself is removed, not the object that it points to. Otherwise, the directory entry indicated by path is removed, and the link count of the underlying object is decremented; if the count goes to zero the underlying object is removed. If there are any existing references to the open file (or directory), the removal is delayed until all references to it have been closed.

This routine will remove a symlink (and not the object that the symlink might point to). If dir_handle indicates an open named attribute directory, and if path indicates a named attribute within that directory, this routine will remove the named attribute indicated.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_DENIED
: Deletion of this object is denied because of an existing open reference made with DAP_NO_DELETE set. See dap_open_file() for DAP_NO_DELETE details.

DAP_ERROR_INVALID_DIR_HANDLE
: dir_handle isn't a valid directory handle.

DAP_ERROR_INVALID_CRED_HANDLE
: The credential handle was invalid.

DAP_ERROR_PERM
: The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_NOTEMPTY

The path refers to a non-empty directory.

DAP_ERROR_ACCESS
The supplied credential does not allow access to the path requested.

DAP_ERROR_INVALID_NATTR
The dir_handle indicates a named attribute directory, and path does not indicate a named attribute within that directory.

DAP_ERROR_LOOP
Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_rename

Rename a file system object.

```
DAP_ERROR
dap_rename(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *old_path,
                const DAP_CHAR          *new_path );
```

Description

This routine renames the object specified by old_path, to new_path. Both paths must lie within the same file system, as defined by the server.

The directory handle and paths together indicate the targets of this call. If absolute paths are given, then the directory handle may be NULL; the DAFS name service will be used to locate the target objects. If the paths are relative, they are interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a DAFS directory handle returned by the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

old_path indicates the target of this operation, and is interpreted relative to dir_handle. If the final component of old_path is a symbolic link, the link itself is renamed, not the object that it points to.

new_path indicates the destination path of this renaming, and is also interpreted relative to dir_handle. It must lie within the same file system as the target, as defined by the server, and must exist up to the final component, which must be a directory. The final component of new_path must not exist.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_DENIED
: Renaming would result in the deletion of the existing file system object at new_path, and deletion of this object is denied because of an existing open reference made with DAP_NO_DELETE set. See dap_open_file() for DAP_NO_DELETE details.

DAP_ERROR_INVALID_DIR_HANDLE
: dir_handle isn’t a valid directory handle.

DAP_ERROR_INVALID_CRED_HANDLE
The credential handle was invalid.

DAP_ERROR_PERM
The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_ACCESS
The supplied credential does not allow access to the path requested.

DAP_ERROR_INVALID_NATTR
The dir_handle indicates a named attribute directory, and old_path does not indicate a valid named attribute, or the server does not implement renaming of named attributes.

DAP_ERROR_LOOP
Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_set_acl

Set the access control list of a file system object.

```
DAP_ERROR
dap_set_acl(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path,
                DAP_FLAGS               flags,
                DAP_COUNT               num_aces,
                const DAP_ACL_INFO      *aces_ptr);
```

Description

This routine sets the access control list associated with the file system object specified by path. The number of access control entries supplied is indicated by num_aces.

The directory handle and path together indicate the target of this call. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the target object. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a DAFS directory handle returned from the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is interpreted relative to dir_handle, and must lead to a valid file system object.

flags consists of bit flags used to modify the behavior of this routine. Undefined bits must be zero.
The following flag bits are defined:

DAP_NO_FOLLOW
: If the final component of path is a symbolic link, this flag indicates that the link itself is the target of this operation, rather than whatever it might point to. If the final component of path is not a symbolic link, this flag is ignored.

num_aces indicates the number of access control entries (of type DAP_ACL_INFO) supplied by the caller. Supplying zero truncates the list of access control entries associated with the file system object.

aces_ptr points to an array of DAP_ACL_INFO structures containing the access control entries to be set. If num_aces is zero, this pointer may be NULL.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
: The base_dir_handle given was invalid.

DAP_ERROR_INVALID_CRED_HANDLE
: The credential handle was invalid.

DAP_ERROR_INVALID_ACE
: One of the DAP_ACL_INFO entries contained an invalid type, flag, mask, or identity. The ACL has not been changed.

DAP_ERROR_PERM
: The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_ACCESS
: The supplied credential does not allow access to the path requested.

DAP_ERROR_LOOP
: Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
: The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
: The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
: The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
: The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
: One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
: The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
: A pathname component is not a directory.

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_set_attr

Set the attributes of a file system object.

```
DAP_ERROR
dap_set_attr(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path,
                DAP_FLAGS               flags,
                DAP_STAT_DESC           *descr_ptr,
                DAP_BITMAP              *attrs_changed);
```

Description

This routine sets the attribute information for an object, specified by path. The attribute information must be consistent with the type of the target object. The attributes to be changed are specified by setting bits of the valid_attrs field of the DAP_STAT_DESC input parameter, and those that were actually set are indicated, upon successful return, by the bits set in the attrs_changed output parameter (which may be fewer than requested).

The directory handle and path together indicate the target of this call. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the target object. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a DAFS directory handle returned from the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is interpreted relative to dir_handle, and must lead to a valid file system object.

flags consists of bit flags used to modify the behavior of this routine. Undefined bits must be zero.
The following flag bits are defined:

DAP_NO_FOLLOW
: If the final component of path is a symbolic link, this flag indicates that the link itself is the target of this operation, rather than whatever it might point to. If the final component of path is not a symbolic link, this flag is ignored.

descr_ptr points to a DAP_STAT_DESC structure containing the attribute information to be set. The valid_attrs field indicates which attributes are to be changed. Indirect fields, which point to storage that may not be contiguous with the DAP_STAT_DESC structure, must be correctly initialized if their corresponding bits in descr_ptr->valid_attrs are set.

attrs_changed indicates which attributes were actually changed. Note well that this may be a subset of those requested, if some are not supported.

Returns

Returns zero on success, and attrs_changed indicates which attributes were actually modified. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
The base_dir_handle given was invalid.

DAP_ERROR_INVALID_CRED_HANDLE
The credential handle was invalid.

DAP_ERROR_PERM
The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_ACCESS
The supplied credential does not allow access to the path requested.

DAP_ERROR_LOOP
Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

DAP_ERROR_SYMLINK
Attempt to perform an unsupported operation on a symbolic link.

DAP_ERROR_INVALID_ATTR
An invalid or unsupported attribute was specified.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure

has occurred.

Other DAP_ERROR values may also be returned.

dap_set_fattr

Set the attributes of a file system object,
given an open handle to it.

```
DAP_ERROR
dap_set_fattr(
                DAP_HANDLE              some_handle,
                DAP_STAT_DESC           *descr_ptr,
                DAP_BITMAP              *attrs_changed);
```

Description

This routine sets the attribute information for an object, specified by path. The attribute information must be consistent with the type of the target object. The attributes to be changed are specified by setting bits of the valid_attrs field of the DAP_STAT_DESC input parameter, and those that were actually set are indicated, upon successful return, by the bits set in the attrs_changed output parameter (which may be fewer than requested).

Arguments

some_handle is a valid handle to a file or directory, obtained from, for example, dap_open_dir() or dap_open_file().

descr_ptr points to a DAP_STAT_DESC structure containing the attribute information to be set. The valid_attrs field indicates which attributes are to be changed. Indirect fields, which point to storage that may not be contiguous with the DAP_STAT_DESC structure, must be correctly initialized if their corresponding bits in descr_ptr->valid_attrs are set.

attrs_changed indicates which attributes were actually changed. Note well that this may be a subset of those requested, if some are not supported.

Returns

Returns zero on success, and attrs_changed indicates which attributes were actually modified. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
: The base_dir_handle given was invalid.

DAP_ERROR_INVALID_CRED_HANDLE
: The credential handle was invalid.

DAP_ERROR_PERM
: The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_ACCESS
: The supplied credential does not allow access to the path requested.

DAP_ERROR_SYMLINK
: Attempt to perform an unsupported operation on

a symbolic link.

DAP_ERROR_INVALID_ATTR
An invalid or unsupported attribute was specified.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_set_fenceID

Set the caller's fencing ID.

```
DAP_ERROR
dap_set_fenceID(
        const DAP_CHAR          *fence_ID);
```

Description

This routine sets the fencing ID associated with the caller, and must be called prior to opening any files (similarly to setting up the authentication callback functions). The fencing ID, once set, may not be changed.

Cooperating clients begin by registering the single fencing ID (an arbitrary string) which identifies them. Fencing lists are attached to file systems and file system objects, and indicate those clients that are to be allowed access. Manipulating those fencing lists then provides cooperating clients the ability to revoke a particular client's access.

Arguments

fence_ID points to the fencing ID (an arbitrary NUL-terminated string) used to identify the caller.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

See Also

dap_get_fencelist(), dap_set_fencelist()

Errors

DAP_ERROR_BAD_ARG
: The fencing ID supplied is not NULL and is not valid (perhaps containing invalid characters or being of zero length).

DAP_ERROR_NOT_SUPPORTED
: Fencing is not supported.

Other DAP_ERROR values may also be returned.

dap_set_fencelist

Set the fencing list of a file or file system.

```
DAP_ERROR
dap_set_fencelist(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *path,
                DAP_FLAGS               flags,
                DAP_FENCELIST_UPDATE    action,
                DAP_COUNT               num_fence_ids,
                DAP_CHAR * const        fence_ids_ptr[] );
```

Description

This routine sets the access control list associated with the file system object specified by path. The number of fencing IDs supplied is indicated by num_fence_ids.

Cooperating clients begin by registering the single fencing ID (an arbitrary string) which identifies them. Fencing lists are attached to file systems and file system objects, and indicate those clients that are to be allowed access. Manipulating those fencing lists then provides cooperating clients the ability to revoke an errant client’s access.

The directory handle and path together indicate the target of this call. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the target object. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

The ability to set the fencing list for a file system object is reserved to the owner of the object or a trusted client, while the ability to set the fencing list for a file system is reserved to trusted clients.

Arguments

dir_handle is a DAFS directory handle returned from the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

path is interpreted relative to dir_handle, and must lead to a valid file system object.

flags consists of bit flags used to modify the behavior of this routine. Undefined bits must be zero.
The following flag bits are defined:

DAP_FILESYSTEM
: If this bit is set, the fencelist for the file system underlying path is to be set.

DAP_NO_FOLLOW
: If the final component of path is a symbolic link, this flag indicates that the link itself is the

target of this operation, rather than whatever it might point to. If the final component of path is not a symbolic link, this flag is ignored.

action indicates the operation intended, and may be one of:

DAP_FENCE_APPEND
This action indicates that the supplied list of fencing IDs is to be added (appended) to the existing list, if any.

DAP_FENCE_REPLACE
This action indicates that the supplied list of fencing IDs is to be used to overwrite, or replace, the existing list, if any, associating a completely new list with the indicated file or file system.

DAP_FENCE_REMOVE
This action indicates that the supplied list of fencing IDs is to be removed from the existing list.

When fencing IDs are removed from a file system or file system object, whether by DAP_FENCE_REPLACE or DAP_FENCE_REMOVE operations, all in-progress I/O to that file system or object from the clients associated with the just-denied fencing IDs will be drained (either aborted or completed) and further attempts at access by those clients will fail.

num_fence_ids indicates the number of pointers to fencing IDs (which are arbitrary NUL-terminated strings) being supplied by the caller. Supplying zero truncates the list of fencing IDs associated with the file system object.

fence_ids_ptr points to an array of pointers to fencing IDs (arbitrary NUL-terminated strings) to be associated with the file system or file system object indicated by path. If num_fence_ids is zero, this pointer may be NULL.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

See Also

dap_set_fenceID(), dap_get_fencelist()

Errors

DAP_ERROR_INVALID_DIR_HANDLE
The base_dir_handle given was invalid.

DAP_ERROR_INVALID_CRED_HANDLE
The credential handle was invalid.

DAP_ERROR_PERM
The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_ACCESS
The supplied credential does not allow access to the path requested.

DAP_ERROR_BAD_ARG
The opcode is not one of the known values.

DAP_ERROR_LOOP
Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

DAP_ERROR_NOT_SUPPORTED
Fencing is not supported.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_strerror

Convert a DAFS error return to a printable string

```
const char *
dap_strerror(
               DAP_ERROR               error_code );
```

Description

The dap_strerror() function accepts a DAFS API error number and returns a pointer to the corresponding message string. The returned string is not to be modified by the caller.

If error_code is not a recognized error number, the error message string will contain "unknown DAP_ERROR".

Arguments

error_code is an error return from a DAFS API function.

Returns

Returns a printable error string in all cases.

Errors

N/A

dap_symlink

Create a symbolic link to another file system object.

```
DAP_ERROR
dap_symlink(
                DAP_DIRECTORY_HANDLE    dir_handle,
                DAP_CRED_HANDLE         cred_handle,
                const DAP_CHAR          *any_path,
                const DAP_CHAR          *link_name );
```

Description

The dap_symlink() routine creates a symbolic link (link_name) to any_path. In other words, link_name is the name of the file created, and any_path is the string used in creating the symbolic link. The link_name must not exist.

The directory handle and path (link_name) together indicate the target of this call. If an absolute path is given, then the directory handle may be NULL; the DAFS name service will be used to locate the target object. If the path is relative, it is interpreted in relation to the directory handle. The DAFS API has no concept of "current working directory" since that is not thread-safe.

Arguments

dir_handle is a DAFS directory handle returned by the dap_open_dir() call.

cred_handle is an optional credential handle obtained with the dap_create_credential() routine. If NULL is supplied, the provider will use default credentials if such exist.

any_path is an arbitrary path name.

link_name is interpreted relative to dir_handle. It must not exist, and the caller must have credentials sufficient to create it.

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_DIR_HANDLE
: dir_handle isn’t a valid directory handle.

DAP_ERROR_INVALID_CRED_HANDLE
: The credential handle was invalid.

DAP_ERROR_PERM
: The credential supplied was not sufficient to allow the requested operation.

DAP_ERROR_ACCESS
: The supplied credential does not allow access to the path requested.

DAP_ERROR_LOOP

Too many symbolic links were encountered in translating the path.

DAP_ERROR_UNKNOWN_LOCATION
The location for the target file cannot be found in the DAFS name service.

DAP_ERROR_UNKNOWN_SERVER
The server containing the target file was located by the DAFS name service, but cannot be resolved to a transport address.

DAP_ERROR_UNREACHABLE
The server containing the target file cannot be reached. This could be temporary (a broken cable) or permanent (a configuration error).

DAP_ERROR_UNKNOWN_PATH
The path provided does not resolve to a DAFS server.

DAP_ERROR_PATH
One of the components in the path does not exist.

DAP_ERROR_NAMETOOLONG
The path name exceeds the maximum length supported.

DAP_ERROR_NOT_DIRECTORY
A pathname component is not a directory.

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_unlock_range

Release an advisory read/write lock on a range of bytes.

```
DAP_ERROR
dap_unlock_range(
                DAP_FILE_HANDLE         file_handle,
                DAP_OFFSET              byte_offset,
                DAP_LENGTH              byte_length );
```

Description

This routine releases a currently held record lock for the range of bytes indicated.

It is implementation dependent whether a client may request a lock with one byte range and then unlock a sub-range of the initial lock. Likewise, it is implementation dependent whether a client may lock two adjacent byte ranges or two overlapping byte ranges and then unlock the entire range or a subset spanning parts of both prior locking operations. The caller must be prepared for DAP_ERROR_LOCK_RANGE to be returned in these cases.

Arguments

file_handle is a DAFS file handle as returned by the dap_open_file() or dap_open_nattr() calls.

byte_offset indicates the first byte of the range to be locked. Zero indicates the initial byte.

byte_length is the number of bytes to be locked, with a value of all one bits indicating "everything."

Returns

Returns zero on success. Otherwise returns one of the error values listed below.

Errors

DAP_ERROR_INVALID_FILE_HANDLE
: file_handle isn't a valid DAFS file handle.

DAP_ERROR_LOCK_RANGE
: Unlocking of sub-ranges is not supported.

DAP_ERROR_BAD_ARG
: An argument was invalid (for example, byte_length of zero).

DAP_ERROR_LEASE_EXPIRED
: The client's lease on the lock had expired.

DAP_ERROR_TRANSPORT_FAILURE
: A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.

dap_write

File write operation.

```
DAP_ERROR
dap_write(
                DAP_FILE_HANDLE         file_handle,
                DAP_OFFSET              file_offset,
                DAP_COUNT               io_count,
                const DAP_MEM_DESC      *mem_desc,
                DAP_LENGTH              *done_count );
```

Description

Initiates one write operation, returning control upon completion. An attempt is made to write data from the buffer or buffers pointed to by mem_desc to the file referenced by the file handle at an offset of file_offset.

The number of bytes read is returned in the variable pointed to by done_count.

Arguments

file_handle is a DAFS file handle as returned by the dap_open_file() or dap_open_nattr() calls.

file_offset is the offset in the file to write the data. This parameter is ignored if file_handle was opened with the DAP_APPEND option.

io_count is the number of sequential DAP_MEM_DESC structures, and must be greater than zero.

mem_desc is pointer to a (vector of) descriptor(s) for the I/O operation. Each entry in the vector contains:

dap_mem_handle - a DAFS memory handle that is associated with the buffer pointer and length. If DAP_NULL_MEM_HANDLE is supplied, the provider will register and bind the memory on the fly; it may cache these mappings to speed later operations.

dap_bufferp - a buffer pointer to somewhere within the registered memory region referred to by the DAFS memory handle.

dap_buffer_len - the length in bytes of the buffer.

done_count points to a variable which upon successful return contains the number of bytes transferred.

Returns

Returns zero on success, with the number of bytes read being returned in the variable pointed to by done_count. Otherwise, one of the error values below may be returned.

Errors

DAP_ERROR_INVALID_FILE_HANDLE
file_handle isn’t a valid DAFS file object.

DAP_ERROR_INVALID_MEM_HANDLE
Some entry in the mem_desc has an invalid registered memory handle.

DAP_ERROR_BAD_ARG
The io_count was less than or equal to zero.

DAP_ERROR_UNREGISTERED_MEM
Some entry in the DAP_MEM_DESC is not valid. Either the dap_bufferp is not within a valid registered memory virtual region, or the end of the buffer extends beyond the memory region referred to by the memory handle, or a NULL dap_mem_handle was given and the Provider was unable to register the memory region on the fly.

DAP_ERROR_IO_OVERLAP
This request attempts to write to an area that overlaps a pending write request, possibly leading to undefined results due to the lack of ordering guarantees among simultaneous pending I/O requests.

DAP_ERROR_LOCKED
I/O attempt to a locked region.

DAP_ERROR_WRITE_TOOBIG
This operation is being done on a file opened for append mode, and the size of the write exceeds the maximum for atomic append operations.

DAP_ERROR_FBIG
This operation would exceed the maximum size supported, or would exceed the resources available on the server.

DAP_ERROR_DQUOT
This operation would exceed a resource (quota) limit.

DAP_ERROR_IO
There was a hard and unrecoverable media (disk) error.

DAP_ERROR_NXIO
There was no such device or address (perhaps hardware was taken off-line).

DAP_ERROR_NODEV
The operation is not supported by the device (such as writing to read-only media).

DAP_ERROR_TRANSPORT_FAILURE
A catastrophic and unrecoverable transport failure has occurred.

Other DAP_ERROR values may also be returned.