Managing Persistent Data with Mneme: User's Guide to the Client Interface*

J. Eliot B. Moss *Tony Hosking* *Eric Brown*†

February 17, 1992

Object Oriented Systems Laboratory
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

We present the client interface of the prototype Mneme persistent object store. Mneme stores *objects*, preserving their identity and structural relationships. Its goals include portability, low overhead and extensibility. This document is intended to be an introduction to Mneme for applications programmers. We discuss the basic concepts of Mneme and the interface from the programmer's point of view. An example application program is given in the appendix.

*This project is supported by National Science Foundation Grants CCR-8658074 and DCR-8500332, and by Digital Equipment Corporation, Apple Computer, Inc., GTE Laboratories, and the Eastman Kodak Company.

†Authors' present address: Department of Computer Science, Lederle Graduate Research Center, University of Massachusetts, Amherst, MA, 01003; telephone (413) 545-4206; Internet addresses Moss@cs.umass.edu, Hosking@cs.umass.edu, and Brown@cs.umass.edu.

1 Introduction

The Mneme persistent object store is part of an overall effort to integrate programming language and database features so as to provide better support for cooperative, information intensive tasks. Such tasks include computer-aided design (CAD), computer aided software engineering (CASE), document preparation/publishing and office automation applications, as well as hypertext and other advanced information systems and tools to support group work. These applications commonly need to store and retrieve considerable amounts of highly structured information in a distributed system context, where many people may be cooperating on large tasks.

The approach that Mneme takes to supporting these tasks is to provide the illusion of a large shared heap of objects, directly accessible from the programming language used to build the applications.

This document serves as an introduction to the interface presented to Mneme clients. We discuss Mneme from the point of view of programmers wishing to use Mneme to provide persistent objects to their own applications. Readers interested in a deeper discussion of the design and implementation of Mneme, and the goals and rationales that influenced it, are referred to [Moss, 1990].

2 Basic Concepts

Mneme presents a number of abstractions to its clients: objects, handles, object pointers, files and pools. Briefly, an *object* is a collection of bytes and references to other Mneme objects. Each object is uniquely referenced by an *object identifier*. Acquiring a *handle* on, or a pointer to, an object guarantees a client access to the internals of that object until such time as the handle is destroyed, or the pointer is released. Objects are grouped together into units called *files*. Each file has a distinguished object called the *root object*, from which all other objects within the file may be reached. Objects may reference other objects in the same file, or objects in other files. Every Mneme object is also associated with exactly one *pool*, which determines the policy under which the object is managed. There may be many object pools within a file, each with their own management policy. We now describe these concepts in more detail.

2.1 Objects

A Mneme object consists of three parts: *slots*, *bytes* and *attribute bits*. When an object is created its size is specified by indicating the number of slots, s, and the number of bytes, b, that it should contain. The slots part of an object is a vector of 32-bit signed integers, indexed from 0 to $s-1$. Each slot contains one of three things: a distinguished *empty* value, an immediate integer value, or an *object identifier* (id). Every Mneme object has a distinct id, allowing the object to be located and accessed. Ids and immediates are distinguished by their sign: ids are positive, immediates are negative. The empty slot has the value 0.

Clients are responsible for converting immediates to slots, and vice versa. This is easily accomplished by negating the value of the immediate or slot.

The bytes part is an uninterpreted vector of 8-bit bytes, indexed from 0 to $b-1$. Bytes must not be used to store object ids. This separation of slots from bytes permits garbage collection of the store, and automatic and transparent id interpretation.

Attribute bits (on the order of 8) are made available to clients to mark special properties of objects, such as being read-only. Mneme specifies no particular use for them but they are provided as a hook for extensions.

Mneme considers objects to be typeless. If any type is to be imposed on an object it is done so by the client (e.g., type information might be stored in the first slot of an object). Mneme objects embody the desired structure and minimal object semantics necessary for a large class of applications, but no particular interpretation is imposed on these objects. Every object has an id, and the object's slots describe its pointer relationships to other objects. This structure is simple, general and efficient in storage and access.

2.2 Handles

A handle is a data structure that provides efficient access to the internals of an object. To manipulate an object in the Mneme store using the Mneme call interface, one must first acquire a handle on it, by presenting its id. When access is no longer desired, the handle may be destroyed. The actions of creating and destroying handles serve several purposes. Creating a handle is the time at which the corresponding object may be fetched from the store, causing what we call an *object fault*, unless of course the object is already resident. So long as the handle is held, the object is known to be available, and further checks are not required.

While handles permit faster access to an object, their main drawback is that they consume substantial space. For this reason, clients must allocate and deallocate the space for all handles. Creating a handle simply initializes the handle's contents, and makes available its corresponding object. Destroying a handle does not deallocate the space for the handle, but informs Mneme that guaranteed access to the object is being relinquished.

2.3 Pointers

The Mneme call interface, using handles, was designed to provide safe client access to objects, with maximal protection from arbitrary modification. Sophisticated clients may wish to acquire a direct pointer to an object, allowing them to modify its internals directly. We provide routines that return a direct pointer to an object, so that clients can bypass the overhead of manipulating objects through the call interface. Acquiring a pointer to an object guarantees access to it, until such time as a *release* operation is performed on it. Clients must indicate if they have modified an object, any time prior to releasing it.

2.4 Files

Mneme groups objects together into units called files. A file of objects can be separately named and located within the overall distributed store. Every Mneme object resides in a file, and there is no provision for moving objects from one file to another.

Files provide a convenient unit of storage and modularity. A Mneme file, or groups of related files, are intended to be reasonable units of backup, recovery, garbage collection, and transfer between different Mneme stores. A file may contain on the order of one million objects. Although individual files are constrained in the number of objects they may contain, there is no constraint on the total space of objects, since there is no limit on the number of files that may be constructed.

An object id is valid only while the file in which the object resides is open. This means that clients should not retain ids for use after the file has been closed. Nor should clients synthesize their own ids. This means that clients need some way to begin accessing objects from a newly opened file. To accomplish this each file has a distinguished *root* object from which all other objects in the file may directly, or indirectly (via other objects), be reached. Any object may be made the root of a file. Objects not accessible from the root may become targets for garbage collection (since there is no way for them to be accessed anyway).

Files are created and opened using the file's name in the form of an `FNAME`. Once a file has been opened, further operations are performed on the file using its file id (`FILEID`). A new file id is assigned to a file each time the file is opened. Therefore, a file id is valid only for as long as the file remains open. File ids should be treated like object ids in that they should not be retained for use after the file has been closed and clients should not synthesize their own file ids.

2.5 Pools and Strategies

In addition to grouping objects physically in files, Mneme also allows objects to be grouped logically, according to the policy under which they are managed. Such logical groupings of objects by management policy are called pools. Each Mneme object is a member of exactly one pool. A management *strategy* is associated with each pool when it is created. A strategy is a vector of routines for making individual policy decisions.

Mneme defines a default pool strategy that should be acceptable to most clients, however we expect to provide further strategies to clients as we determine their needs. Sophisticated users are free to develop their own strategies, and Mneme provides means by which strategy routine vectors may be filled in by an application.

Mneme currently supports two strategy routines: one determines where a new object is placed on disk, thus controlling physical clustering of objects; the other decides how a physical clustering of objects is to be grown when the cluster of objects is written back to disk. In the future we expect to support further pool policy decisions, allowing flexible approaches to object clustering, storage allocation, prefetch, concurrency, buffering/caching, and perhaps even security and versioning.

Within a file a pool has a unique identifier called its pool number (`POOLNUM`). Pool numbers do not change once they have been assigned and users may store them for future reference to a pool. In order to distinguish a pool in one file from a pool in another file with the same pool number, a pool id (`POOLID`) is assigned to each pool when its file is opened. In general, pool ids are required for any operation involving a pool. Like object ids and file ids, pool ids are valid only for as long as the file containing the pool remains open. Routines are provided that convert back and forth between pool numbers and pool ids.

3 The Client Interface

The interface is designed with several programming conventions in mind:

- Every routine returns a standard *result code*. A negative return code indicates an error, zero (`MN_SUCCESS`) indicates unqualified success, and a positive code indicates a specific success condition. Errors are trapped by Mneme's exception handling mechanism. The default exception handler simply prints a message and returns the error code.
- Any routine that must return data of unknown size allocates the space for that data itself, setting a pointer variable supplied by the client. Clients must free the storage when it is no longer needed using the memory allocation and deallocation functions supplied by Mneme. Fixed size result items are always allocated by the client, to allow efficient static or stack allocation. The client passes the address of these items to the Mneme routines as required.
- While C does not support data abstraction, several types are considered abstract in the Mneme interface, and client code should not make assumptions about how they are represented. These types include `FNAME`, `POOLID`, `POOLNUM`, `STRATID`, `ID`, `HANDLE`, `SLOT`, `FILEID` and `HANDLER` .

In the following presentation arguments to routines are labelled to indicate their role more clearly. `IN` indicates an argument that is passed by value, `INOUT` indicates an argument that is passed by reference, and `OUT` indicates an argument that specifies the address of a location in which a result is to be returned. Each operation is assumed to return `MN_SUCCESS` if it completes successfully, unless it returns a more specific success code. A list of possible return codes (other than `MN_SUCCESS`) is given with the description of each operation.

3.1 General and File Operations

The initialization routine establishes a Mneme *session*, performing initialization of system data structures. This routine should be invoked once, before any other Mneme operation, to establish a context of interaction with the Mneme store.

- `MnInit ()`
  Initiate a Mneme session.

Mneme files are identified either by a name or an id. Data type `FNAME` is the Mneme abstraction for a file name. The following macros are provided to set up and examine variables of type `FNAME`:

- **fname_setStr (string, fnamePtr)**
  Set up the given `FNAME` to refer to the file named by the string. Argument **string** must be a null-terminated C string, while **fnamePtr** is a pointer to a variable of type `FNAME`.

- **fname_isEqual (fname1, fname2)**
  Determine if the given `FNAME`s refer to the same file.

- **fname_isNil (fname)**
  Determine if the given `FNAME` refers to a file.

The following file operations expect a file name (in the form of an `FNAME`) as an argument.

- **MnFileExists (IN FNAME *fname)**

MNPTRISNULL	**fname** is null
MNFILEXIST	The existence of the file cannot be determined

  Returns `MN_TRUE` (1) if the named file exists and can be accessed, and `MN_FALSE` (0) if the file definitely does not exist.

- **MnFileCreate (IN FNAME *fname, OUT FILEID *fid)**

MNPTRISNULL	One of the pointer arguments is null
MNFILTABFUL	Unable to open any more files
MNFILCREAT	Unable to create the file
MNFILOPEN	Unable to open the file
MNSEGCREAT	Unable to allocate the file's internal tables

  Create and open a new file as named, assigning it a file id.

- **MnFileDestroy (IN FNAME *fname)**

MN_FAILURE	File does not exist, or is currently open
MNPTRISNULL	**fname** is null

  Destroy the named file.

- **MnFileRename (IN FNAME *oldName, FNAME *newName)**

MN_FAILURE	File does not exist, is currently open, or another file having the new name already exists
MNPTRISNULL	One of the pointer arguments is null

  Rename the file.

- **MnFileOpen (IN FNAME *fname, OUT FILEID *fid)**

MNPTRISNULL	One of the pointer arguments is null
MNFILTABFUL	Unable to open any more files
MNFILOPEN	Unable to open the file
MNSEGREAD	Unable to read the file's internal tables

  Open the named file, assigning it a file id.

- **MnFileId (IN FNAME *fname, OUT FILEID *fid)**

MN_FAILURE	File does not exist, or is not currently open
MNPTRISNULL	One of the pointer arguments is null

  Determine the file id of the named file.

The remaining file operations are oriented towards file ids.

- **MnFileClose (INOUT FILEID *fid)**

MNPTRISNULL	`fid` is null
MNBADFILEID	`fid` is invalid
MNSEGWRITE	Unable to write the file
MNFILCLOSE	Unable to close the file

  Close the identified file. If the operation succeeds then the given FILEID will be set to `EMPTY_FILEID`.

- **MnFileName (IN FILEID fid, OUT FNAME *fname)**

MNBADFILEID	`fid` is invalid
MNPTRISNULL	`fname` is null

  Determine the name of the identified file.

- **MnFileGetPools (IN FILEID fid, OUT POOLID *(poolIds[]), int *count)**

MNPTRISNULL	One of the pointer arguments is null
MNBADFILEID	`fid` is invalid

  Allocate an array consisting of the pool ids of all the pools in the indicated file. The size of the array is returned in `count`. When the array is no longer needed the client must free its space with a call to `mn_free`. Note that there is *no* correlation between a pool id's index in the array and the pool number associated with that pool.

- **MnFileSetRoot (IN FILEID fid, HANDLE *h, ID id)**

MNBADFILEID	`fid` is invalid
MNBADOBJECT	Either the given `HANDLE` is invalid, or if `h` is null then `id` is invalid

  Set the root object of the file from the handle, or, if the handle pointer is null, the object id.

- **MnFileGetRoot (IN FILEID fid, HANDLE *h, ID *id)**

MNSEGFETCH	The object was faulted from disk (this is a success code)
MNBADFILEID	`fid` is invalid
MNPTRISNULL	`id` is null

  Determine the root object of the file. If the handle pointer is not null then create a handle for the object.

3.2 Object and Handle Operations

A few operations related to objects use ids, but most use handles. We first discuss the operations that use ids.

- **`MnObjectCreate (IN POOLID p, LONG s, LONG b, BYTE attr, ID nearId,`**
  `OUT HANDLE *h, ID *id)`

`MNBADPOOLID`	`p` is invalid
`MNEMPTYSTRAT`	Pool `p` has no associated strategy
`MNBADSLOTNUM`	`s` is out of range
`MNBADBYTENUM`	`b` is out of range
`MNPOOLCONFLICT`	Pool of **`nearId`** object conflicts with given pool
`MNPTRISNULL`	`id` is null

  Create a new object having the given number of slots, `s`, and bytes, `b`, and initial attribute bits `attr`. If the **`nearId`** argument is non-zero, then it hints that the new object should be allocated "close to" the object it indicates. The operation returns the id of the newly created object, and, if `h` is not the null pointer, a handle for the object as well.

- **`MnObjectCompare (IN ID id1, ID id2)`**
  Returns `MN_TRUE` (1) if the arguments `id1` and `id2` refer to the same object, and `MN_FALSE` (0) if they do not.

The first group of handle oriented routines are simple inquiries.

- **`MnObjectPool (IN HANDLE *h, OUT POOLID *p)`**
  `MnObjectFile (IN HANDLE *h, OUT FILEID *f)`
  `MNPTRISNULL` One of the pointer arguments is null

  Determine the pool/file that contains the indicated object.

- **`MnObjectNumBytes (IN HANDLE *h, OUT LONG *b)`**
  `MnObjectNumSlots (IN HANDLE *h, OUT LONG *s)`
  `MNPTRISNULL` One of the pointer arguments is null

  Determine the number of bytes/slots in the indicated object.

The second group allows access to the three parts of an object.

- **`MnObjectAttrs (IN HANDLE *h, BYTE andMask, BYTE xorMask, OUT BYTE *attr)`**
  `MNPTRISNULL` One of the pointer arguments is null

  Suppose that the current value of the attributes of the specified object is v. The result of this operation (which is also stored in the object as the new value for its attributes) is[1] ($v \wedge$ **`andMask`**) $\oplus$ **`xorMask`**. This allows any of the four Boolean operations on a single bit value (set, clear, invert, nothing) to be performed individually to each attribute bit, all at once, requiring only one attribute routine.

[1] $\oplus$ is the logical exclusive-or operator.

- `MnObjectGetBytes (IN HANDLE *h, LONG first, LONG count,`
  `OUT BYTE *bytes)`
  `MnObjectPutBytes (IN HANDLE *h, LONG first, LONG count, BYTE *bytes)`

`MNPTRISNULL`	One of the pointer arguments is null
`MNBADBYTENUM`	`first` is out of range
`MNBADCOUNT`	`count` is out of range

  Read/write `count` bytes from/to the object, beginning at the byte indexed by `first`.

- `MnObjectGetSlot (IN HANDLE *h, LONG which, OUT SLOT *slot)`
  `MnObjectPutSlot (IN HANDLE *h, LONG which, SLOT slot)`

`MNPTRISNULL`	One of the pointer arguments is null
`MNBADSLOTNUM`	`which` is out of range

  Read/write the slot indexed by `which` from the object.

- `MnObjectGetSlots (IN HANDLE *h, LONG first, LONG count,`
  `OUT SLOT *slots)`
  `MnObjectPutSlots (IN HANDLE *h, LONG first, LONG count, SLOT *slots)`

`MNPTRISNULL`	One of the pointer arguments is null
`MNBADSLOTNUM`	`first` is out of range
`MNBADCOUNT`	`count` is out of range

  Read/write `count` slots from/to the object, beginning at the slot indexed by `first`.

- `MnCopyBytes (IN HANDLE *from, HANDLE *to,`
  `LONG fromFirst, LONG toFirst, LONG count)`

`MNPTRISNULL`	One of the pointer arguments is null
`MNBADBYTENUM`	One of `fromFirst` or `toFirst` is out of range
`MNBADCOUNT`	`count` is out of range

  Copy a range of bytes from one object to another. If the destination object is the same as the source object and the ranges overlap then the copy will be performed so that source bytes are not overwritten before they have been copied.

- `MnCopySlots (IN HANDLE *from, HANDLE *to,`
  `LONG fromFirst, LONG toFirst, LONG count)`

`MNPTRISNULL`	One of the pointer arguments is null
`MNBADSLOTNUM`	One of `fromFirst` or `toFirst` is out of range
`MNBADCOUNT`	`count` is out of range

  Copy a range of slots from one object to another. If the destination object is the same as the source object and the ranges overlap then the copy will be performed so that source slots are not overwritten before they have been copied.

- `MnObjectFillBytes (IN HANDLE *h, LONG first, LONG count, BYTE byte)`

`MNPTRISNULL`	`h` is null
`MNBADBYTENUM`	`first` is out of range
`MNBADCOUNT`	`count` is out of range

  Fill a range of bytes with a given byte value.

- `MnObjectFillSlots (IN HANDLE *h, LONG first, LONG count, SLOT slot)`

`MNPTRISNULL`	`h` is null
`MNBADSLOTNUM`	`first` is out of range
`MNBADCOUNT`	`count` is out of range

  Fill a range of slots with a given slot value.

Recall that a slot can be empty, contain an id, or contain an immediate value. The following operations allow these cases to be distinguished.

- `MnSlotIsEmpty (SLOT slot)`
  `MnSlotIsData  (SLOT slot)`
  `MnSlotIsId    (SLOT slot)`

  Return `MN_TRUE` (1) if the given slot is the empty slot, data, or an id, respectively. Return `MN_FALSE` (0) otherwise.

Finally, there are three operations on handles themselves.

- `MnHandleCreate (IN ID id, OUT HANDLE *h)`

`MNSEGFETCH`	The object was faulted from disk (this is a success code)
`MNPTRISNULL`	`h` is null
`MNBADID`	`id` is invalid

  Create a handle for the object corresponding to the given id. This operation initializes the contents of the handle, possibly faulting in the object, and permits the client to access the internals of the object. The client must allocate the space for the handle.

- `MnHandleDestroy (INOUT HANDLE *h)`

`MNPTRISNULL`	`h` is null

  Destroy the given handle. This operation signals that the client is relinquishing access to the object, nulling out the handle's contents. It does not deallocate the space for the handle, so that the handle may be reinitialized for a different object in a later call to `MnHandleCreate`.

- `MnHandleId (IN HANDLE *h, OUT ID *idPtr)`

`MNPTRISNULL`	One of the pointer arguments is null

  Determine the id of the object corresponding to the given handle.

3.3 Direct Pointer Operations

The following routines are provided to allow sophisticated clients direct access to objects in the Mneme buffers, bypassing the call interface described in the previous section. Clients wishing to use these routines should be aware that direct access permits arbitrarily bad modification of objects, and that appropriate care should be taken.

Acquiring a direct pointer to a Mneme object exposes its internals, which are slightly different to the abstraction presented by the call interface. The principal difference is the use of *persistent* object identifiers (pids). Object identifiers, as stored in the slots of objects within the Mneme store, always name objects within the same file as the object containing the identifier. Because clients may have many files open at the same time during a Mneme session, a pid, which is unique only within one file, must be converted into a client id for use by the client. The call interface takes care of this conversion automatically. However, clients using the direct pointer interface must perform this conversion for themselves. To assist in this, we include a number of routines for converting identifiers.

The following operations name an object using a client id:

- **`MnIdPtr (IN ID id, int mod, OUT PTR *ptr)`**

  `MNSEGFETCH` The object was faulted from disk (this is a success code)
  `MNPTRISNULL` `ptr` is null
  `MNBADID` `id` is invalid

  Acquire a pointer to the object identified by the client id, indicating if it is to be modified (`mod` non-zero).

- **`MnIdMod (IN ID id)`**

  `MNBADID` `id` is invalid
  `MNRELEASE` object is not resident

  Indicate the object is modified.

- **`MnIdRelease (IN ID id, int mod)`**

  `MNBADID` `id` is invalid
  `MNRELEASE` object is not resident

  Release the object, indicating if it has been modified (`mod` non-zero).

The corresponding operations for pids are as follows:

- **`MnPidPtr (IN FILEID fid, PID pid, int mod, OUT PTR *ptr)`**

  `MNSEGFETCH` The object was faulted from disk (this is a success code)
  `MNPTRISNULL` `ptr` is null
  `MNBADFILEID` `fid` is invalid
  `MNBADPID` `pid` is invalid

  Acquire a pointer on the identified object in the given file, indicating if it is to be modified (`mod` non-zero).

- **`MnPidMod (IN FILEID fid, PID pid)`**

  `MNBADFILEID` `fid` is invalid
  `MNBADPID` `pid` is invalid
  `MNRELEASE` object is not resident

  Indicate the object is modified.

- `MnPidRelease (IN FILEID fid, PID pid, int mod)`

`MNBADFILEID`	`fid` is invalid
`MNBADPID`	`pid` is invalid
`MNRELEASE`	object is not resident

  Release the object, indicating if it has been modified (`mod` non-zero).

For clients who desire higher performance at the cost of minimal error checking, the following two routines are provided. The routines do not follow the Mneme convention of returning an `RCODE`. Rather, the desired pointer is returned by the functions. If the returned value is `NULL`, the operation has failed and a global status variable, `MnStatus`, will contain the error code. It is assumed that the arguments to the functions are valid.

- `PTR MnPidPtrNoMod (IN FILEID fileId, PID pid)`

  Return a pointer to the identified object in the given file, assuming modifications *will not* be made.

- `PTR MnPidPtrMod (IN FILEID fileId, PID pid)`

  Return a pointer to the identified object in the given file, assuming modifications *will* be made.

The object creation routine is similar to `MnObjectCreate`. However, it always returns a direct object pointer rather than optionally returning a handle:

- `MnObjectCreatePtr (IN POOLID p, LONG s, LONG b, BYTE attr, ID nearId,`
  `OUT PTR *ptr, ID *id)`

`MNBADPOOLID`	`p` is invalid
`MNEMPTYSTRAT`	Pool `p` has no associated strategy
`MNBADSLOTNUM`	`s` is out of range
`MNBADBYTENUM`	`b` is out of range
`MNPOOLCONFLICT`	Pool of `nearId` object conflicts with given pool
`MNPTRISNULL`	One of `ptr` or `id` is null

  Create a new object having the given number of slots, `s`, and bytes, `b`, and initial attribute bits `attr`. If the `nearId` argument is non-zero, then it hints that the new object should be allocated "close to" the object it indicates. The operation returns the id of the newly created object, and a direct pointer to its first slot.

We also provide routines for determining information about an object, given a direct object pointer. Notice that there is just one routine, `MnPtrInfo`, corresponding to the two routines, `MnObjectNumBytes` and `MnObjectNumSlots`:

- `MnPtrInfo (IN PTR ptr, OUT LONG *s, LONG *b)`

`MNPTRISNULL`	One of the pointer arguments is null
`MNBADPTR`	`ptr` does not point to the first slot of a Mneme object

  Determine the number of slots and bytes in the object.

- `MnPtrAttrs (IN PTR ptr, BYTE andMask, BYTE xorMask, OUT BYTE *attr)`
  `MNPTRISNULL` One of `ptr` or `result` is null

  This is the direct pointer routine corresponding to `MnObjectAttrs`.

Finally, there are conversion routines for converting a client id to a file and pid, and vice versa:

- `MnIdPid (IN ID id, OUT PID *pid)`
  `MNBADID` `id` is invalid
  `MNPTRISNULL` `pid` is null

  Convert a client id into a pid.

- `MnIdFile (IN ID id, OUT FILEID *fid)`
  `MNBADID` `id` is invalid
  `MNPTRISNULL` `fid` is null

  Determine what file the object is in.

- `MnPidId (IN FILEID fid, PID pid, OUT ID *id)`
  `MNBADFILEID` `fid` is invalid
  `MNBADPID` `pid` is invalid
  `MNPTRISNULL` `id` is null

  Convert a pid into a client id.

3.4 Pool and Strategy Operations

Recall that an object's pool determines its management policy. In future releases of Mneme pools will have *attributes* associated with them, allowing them to have arbitrary, client-specified (and strategy-specific) parameters, and routines will be provided for setting and retrieving these attributes. For now, the following routines permit the creation of pools, and tailoring of the strategy routines.

- `MnPoolCreate (IN FILEID f, STRATID s, OUT POOLID *p)`
  `MNBADFILEID` `f` is invalid
  `MNPTRISNULL` `p` is null
  `MNBADSTRATID` `s` is invalid
  `MNTOOMANYPOOLS` Unable to allocate space for a new pool

  Create a new pool in the indicated file, having the given strategy. The default strategy has `STRATID` 1 (`DEFAULT_STRAT`).

- `MnStrategySetRoutine (IN STRATID s, STRAT_EVENT e, RCODE (*routine)())`
  `MNBADSTRATID` `s` is invalid
  `MNBADEVENT` `e` is invalid
  `MNPTRISNULL` `routine` is null

This routine is for the use of sophisticated users wishing to define pool policies that use their own strategy routines. Currently, Mneme allows for a maximum of 16 possible strategies, reserving strategies 0 (`SYSTEM_STRAT`) and 1 (`DEFAULT_STRAT`) for itself. This means that clients may use `STRATID`s 2 through 15 to identify their own strategies, however there are no guarantees that any of these strategy identifiers will be available to clients in the future.

- `MnPoolNumId (IN FILEID fileId, POOLNUM poolNum, OUT POOLID *poolId)`
  `MNBADFILEID` `fileId` is invalid
  `MNBADPOOLID` `poolNum` is invalid
  `MNPTRISNULL` `poolId` is null

  Obtain the pool id for the pool with number `poolNum` in the file identified by `fileId`.

- `MnPoolIdNum (IN POOLID poolId, OUT POOLNUM *poolNum)`
  `MNBADPOOLID` `poolId` is invalid
  `MNPTRISNULL` `poolNum` is null

  Obtain the pool number for the pool identified by `poolId`.

3.5 Profiling and File Information Operations

The interface offers several operations for acquiring statistics and counts. Two data types are provided: `FILEDATA` is a structure that holds allocation information about a Mneme file, and `PROFDATA` is a structure that contains counts of various operations. For each structure operations are provided to print the structure on the standard output, sample data into a client-supplied structure, and reset the profiling counts.

- `MnFileInfoPrint (IN FILEID fid)`
  `MNBADFILEID` `fid` is invalid

  Print allocation information about the file.

- `MnFileInfoCopy (IN FILEID fid, OUT FILEDATA *fdata)`
  `MNBADFILEID` `fid` is invalid
  `MNPTRISNULL` `fdata` is null

  Determine file allocation information.

Since the collection of profiling statistics affects the performance of Mneme, there is a compiler flag `PROFILE` that enables or disables collection (see the Mneme makefile for details). The following routines all return an error code if profiling has been disabled.

- `MnProfileReset ()`
  `MNNOPROFILE` Profiling is disabled

  Reset the profile counters to initial values. Clients should call this when profiling should begin.

- `MnProfilePrint ()`
  `MNNOPROFILE` Profiling is disabled

  Print the current profiling counts.

- `MnProfileCopy (PROFDATA *pdata)`
  `MNNOPROFILE` Profiling is disabled
  `MNPTRISNULL` `pdata` is null

  Sample the current profiling counts.

3.6 Exception Handling

The client interface also includes an exception handling facility that traps all errors as they occur. A stack of exception handlers is maintained, and Mneme provides routines that permit clients to push and pop their own exception handlers on this stack. The stack is initialized with a default exception handler that simply prints an error message and returns the error code. Whenever an exception occurs the handler that is on the top of the stack is invoked to deal with it. Exception handlers may defer action to the next handler below them on the stack using the following macro.

- `deferToNextHandler (rcode)`

All exception handlers must conform to the signature:
`RCODE handler (IN HANDLER_NODE *nextNode, RCODE rcode)`
The routines for pushing and popping handlers are as follows.

- `MnHandlerPush (HANDLER *handler)`
  `MNPTRISNULL` `handler` is null

  Push a handler on the stack, where `handler` points to a routine having the mandated signature for exception handlers as defined above.

- `MnHandlerPop ()`
  `MNPOPHANDLER` Attempted to pop the default handler from the stack

  Pop a handler from the stack.

References

[Moss, 1990] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Trans. Inf. Syst. 8*, 2 (Apr. 1990), 103–139.

A. An Example

In this section we present an annotated example Mneme application that builds a complete tree of arbitrary height and degree, and then performs an in-order traversal on the tree. Each node in the tree is represented as a Mneme object, and the root node of the tree is the root object of the Mneme file. Each node may have a number of bytes of data associated with it. Each internal (i.e., non-leaf) node also requires as many slots as the degree of the tree, to hold the ids of its subtrees. The program interactively requests the following parameters from the user:

- Branching factor (i.e., degree) of the tree;
- Height of tree;
- Number of data bytes to be allocated per node;
- Whether the data bytes should be *read* from each node as it is traversed;
- Whether the data bytes should be *written* as each node is traversed;
- Whether the tree file should be closed (forcing the tree nodes to disk) and then reopened before the tree is traversed. If the file is closed then the traversal will cause the nodes of the tree to be faulted in from disk as they are needed.

We list the program, and intersperse it with comments.

```
# include <mneme/mneme.h>
static POOLID poolId;
static BYTE data[256];
```

Include the Mneme header file and declare static variables: `poolId` will be used to identify the pool in which each of the nodes are to be created.

```
void createTree (height, branches, bytes, this)
     int height;
     int branches;
     int bytes;
     ID *this;
{
  HANDLE handle;

  if (height == 0) { /* height of leaves is 0 */
    int slots = 0;

    MnObjectCreate (poolId, slots, bytes, 0, EMPTY_ID, &handle, this);
    MnObjectPutBytes (&handle, 0, bytes, data);
  } else {
```

```
    int child;
    ID  subtree;
    int slots = branches;

    MnObjectCreate (poolId, slots, bytes, 0, EMPTY_ID, &handle, this);
    MnObjectPutBytes (&handle, 0, bytes, data);
    for (child = 0; child < slots; child++) {
      createTree (height - 1, branches, bytes, &subtree);
      MnObjectPutSlot (&handle, child, subtree);
    }
  }
  MnHandleDestroy (&handle);
}
```

`createTree` builds a tree of height `height` and degree `branches`. The Mneme ID of the root node of this tree is returned via `this`. The routine recursively calls itself to build the subtrees of this node.

```
void traverseTree (this, read, write)
     ID this;
     int read;
     int write;
{
  HANDLE handle;
  int slots;
  int bytes;
  int child;
  ID subtree;

  MnHandleCreate (this, &handle);
  MnObjectNumSlots (&handle, &slots);
  MnObjectNumBytes (&handle, &bytes);

  if (read)
    MnObjectGetBytes (&handle, 0, bytes, data);
  if (write)
    MnObjectPutBytes (&handle, 0, bytes, data);
  for (child = 0; child < slots; child++) {
    MnObjectGetSlot (&handle, child, &subtree);
    traverseTree (subtree, read, write);
  }

  MnHandleDestroy (&handle);
}
```

traverseTree does an in-order traversal of the tree whose root node has Mneme id **this**. The parameters **read** and **write** determine whether the data bytes should be read or written as the node is traversed.

```
RCODE myHandler (nextNode, rcode)
     HANDLER_NODE *nextNode;
     RCODE rcode;
{
  printf ("Defer to next handler...\n");
  return(deferToNextHandler(rcode));
}
```

To illustrate the use of the exception handling facilities we define our own handler routine that just prints a message and defers to the next handler. Function **main** queries for the necessary parameters, initializes the Mneme session, illustrates the use of exception handlers, creates the Mneme file and the tree, and traverses it, printing out various profiling statistics along the way.

```
main () {
  int branches;
  int bytes;
  int height;
  char str[32];
  int read, write, close;
  FNAME fname;
  FILEID fileId;
  ID root;

  do {
    printf ("Enter branching factor: ");
  } while (scanf ("%d", &branches) != 1);
  do {
    printf ("Enter height: ");
  } while (scanf ("%d", &height) != 1);
  do {
    printf ("Enter bytes stored/retrieved per node (up to 256): ");
  } while (scanf ("%d", &bytes) != 1 || bytes > 256);
  do {
    printf ("Should the traversal read data from the nodes (y/n): ");
  } while (scanf ("%s", str) != 1 || (str[0] != 'y' && str[0] != 'n'));
  read = str[0] == 'y';
  do {
    printf ("Should the traversal write data to the nodes (y/n): ");
  } while (scanf ("%s", str) != 1 || (str[0] != 'y' && str[0] != 'n'));
  write = str[0] == 'y';
  do {
```

```
  printf ("Should file be closed before tree traversal (y/n): ");
} while (scanf ("%s", str) != 1 || (str[0] != 'y' && str[0] != 'n'));
close = str[0] == 'y';

printf ("\n");

/* Initialise the Mneme session */
MnInit ();

/* Illustrate exception handlers */
MnHandlerPush (myHandler); /* Push my handler */
MnHandlerPush ((HANDLER *)NULL); /* Generate an exception */
MnHandlerPop (); /* Pop my handler */

fname_setStr ("Tree.mn", &fname);
if (MnFileExists (&fname))
  MnFileDestroy (&fname);

MnProfileReset ();

printf ("Creating file...\n");
MnFileCreate (&fname, &fileId);
MnPoolCreate (fileId, DEFAULT_STRAT, &poolId);
printf ("done\n");
MnProfilePrint ();

printf ("\nCreating tree...\n");
createTree (height, branches, bytes, &root);
MnFileSetRoot (fileId, (HANDLE *)NULL, root);
printf ("done\n");
MnProfilePrint ();
MnFileInfoPrint (fileId);

if (close) {
  printf ("\nClosing file...\n");
  MnFileClose (&fileId);
  printf ("done\n");
  MnProfilePrint ();

  MnFileOpen (&fname, &fileId);
  MnFileGetRoot (fileId, (HANDLE *)NULL, &root);
}

printf ("\nTraversing tree...\n");
traverseTree (root, read, write);
```

```
  printf ("done\n");
  MnProfilePrint ();
  MnFileInfoPrint (fileId);

  printf ("\nClosing file...\n");
  MnFileClose (&fileId);
  printf ("done\n");
  MnProfilePrint ();
}
```

B. Using the Direct Pointer Interface

We recode routines `createTree` and `traverseTree` using direct pointers.

```
void createTree (height, branches, bytes, this)
     int height;
     int branches;
     int bytes;
     ID *this;
{
  PTR ptr;

  if (height == 0) { /* height of leaves is 0 */
    int slots = 0;
    int i;

    MnObjectCreatePtr (poolId, slots, bytes, 0, EMPTY_ID, &ptr, this);
    memcpy (ptr, data, bytes);
  } else {
    int child;
    ID  subtree;
    int slots = branches;

    MnObjectCreatePtr (poolId, slots, bytes, 0, EMPTY_ID, &ptr, this);
    memcpy ((SLOT *)ptr + slots, data, bytes);
    for (child = 0; child < slots; child++) {
      createTree (height - 1, branches, bytes, &subtree);
      MnIdPid (subtree, &((PID *)ptr)[child]);
    }
  }
  MnIdRelease (*this, 0);
  /*
   * There is no need to indicate ``this'' as modified, since
   * it was just created, and Mneme knows it is modified.
```

```
   */
}

void traverseTree (this, read, write)
     ID this;
     int read;
     int write;
{
  PTR ptr;
  LONG slots;
  LONG bytes;
  int child;
  ID subtree;
  FILEID fileId;

  MnIdFile (this, &fileId);
  MnIdPtr (this, write, &ptr); /* Indicate intention to modify */
  MnPtrInfo (ptr, &slots, &bytes);

  if (read)
    memcpy (data, (SLOT *)ptr + slots, bytes);
  if (write)
    memcpy ((SLOT *)ptr + slots, data, bytes);
  for (child = 0; child < slots; child++) {
    MnPidId (fileId, ((PID *)ptr)[child], &subtree);
    traverseTree (subtree, read, write);
  }

  MnIdRelease (this, 0); /* Intention to modify already noted */
}
```