Mneme V3.x User's Guide

J. Eliot B. Moss *Tony Hosking* *Eric Brown**
Object Systems Laboratory
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

May 4, 1994

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 is meant to serve as a general overview of the Mneme system, as well as a user's guide to the library routines provided by the system. Section 2 gives a high level descriptions of the basic concepts and components of the system. Section 4 details each of the functions in the user interface to the system.

2 Basic Concepts

Mneme presents a number of abstractions to its clients: objects, object pointers, files, object pools, and buffer 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 pointer to an object guarantees a client access to the internals of that object until such time as 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 (indirectly) objects in other files. Every Mneme object is also associated with exactly one *object pool*, which determines the policy under which the object is managed. There may be many object pools within a file, each with its own management policy. Resident objects are placed in a *buffer pool*, which has associated allocation and replacement policies. The *Resident Object Table* keeps track of which objects are resident and maps object identifiers to object pointers. Figure 1 gives an overview of the architecture. We now describe these concepts in more detail.

2.1 Objects

A Mneme object is a chunk of contiguous bytes that has been assigned a unique identifier. Objects may contain inter-object references, which are represented on disk by 32-bit object identifiers. The client determines the layout of objects and is responsible for any translation required between the disk and main memory formats of objects. Clients

*Authors' present address: Department of Computer Science, Lederle Graduate Research Center, University of Massachusetts, Amherst, MA, 01003; telephone: (413) 545-0256; Email: {moss, hosking, brown}@cs.umass.edu

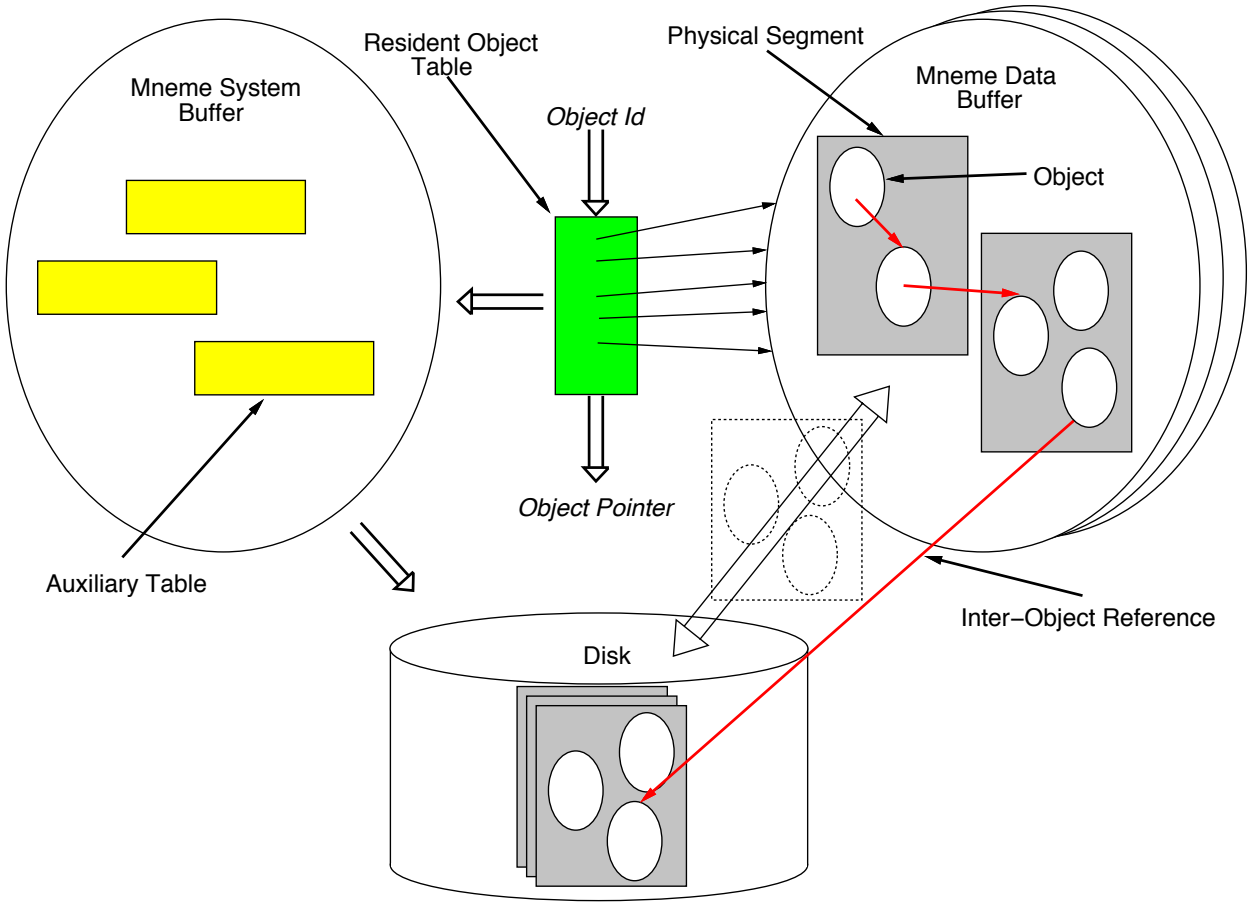

Figure 1: Mneme architecture overview. The data buffers contain resident objects clustered in physical segments. There may be multiple buffer pools, each with its own management policies. Pointers to resident objects are obtained by presenting object ids to the Resident Object Table. If an object is not resident, its pool is responsible for making it resident. The pool uses auxiliary tables kept in the system buffer to locate the object on disk and transfer its physical segment into the data buffer.

must also be able to locate for Mneme any identifiers stored in the objects. This would be necessary, for instance, during garbage collection of the persistent store.

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 4 bytes 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.

2.2 Pointers

Typically, an object is manipulated by obtaining a pointer to that object. This is done by presenting Mneme with the identifier of the desired object, which returns a pointer to the memory resident object. If the object was not previously memory resident, it is read into memory from the file that contains the object. Acquiring a pointer to an object guarantees access to it, until such time as a *release* operation is performed on the object. Clients must indicate if they have modified an object, any time prior to releasing it.

2.3 Files

Mneme groups objects together into files supported by the operating system. 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 approximately 256 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. Therefore, 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).

Additionally, an object id is *unique* only within the object's file. Multiple files may be open simultaneously, however, so object ids in a file, called *persistent ids*, are mapped to globally unique identifiers, called *client ids*, when the objects are accessed. The number of objects that may be accessed simultaneously is bounded by the number of client ids, approximately 256 million.

Within a file, objects are organized into *physical segments* and *logical segments*. Physical segments are the unit of transfer between disk and main memory. Objects are physically grouped together in physical segments and the granularity of clustering within a file is the physical segment. Logical segments are logical groupings of objects and are used to identify and locate objects within physical segments. This allows an object's placement in a physical segment to be arbitrary and independent of the object identifier, and avoids restrictions on physical segment size.

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.

2.4 Object 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 few pool strategies that should be adequate for many 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.

Currently the mandatory routines that must be provided by a pool strategy include pool initialization, servicing a resident object table miss, closing all currently open (memory resident) physical segments, flushing all open and modified physical segments to the file, attaching to a buffer pool, and freeing a physical segment that has been chosen for replacement in the buffer pool. These routines are provided as "call-backs" to Mneme and represent most of the functionality that is customizable by the pool.

Pool strategies are globally unique and multiple pools from the same or different files may use the same strategy. A strategy is embodied by the routines that make up the strategy. As such, a strategy is part of the code of the application, not something persistent stored in a Mneme file. Care must be taken to ensure that a strategy is consistent across all applications that will access the same files (or more specifically, the same object pools).

In addition to the above strategy routines, an object pool must provide an object creation routine. Object creation involves choosing a physical and logical segment for the object, determining the object's persistent identifier, and initializing the object. Selection of the physical segment for a new object is based on the clustering scheme of the pool. If there is not enough room for the new object in the chosen physical segment, the pool must create a new physical segment. Creation of physical segments is supported by Mneme, which manages the lower level file operations of segment creation, storage, and retrieval.

Logical segments are groupings of objects in the persistent identifier space. Since the persistent identifier space is local to a file but shared by all pools in the file, a pool must request chunks of this identifier space from Mneme. Once a chunk of logical segment numbers has been allocated to a pool, the pool is free to assign the persistent identifiers in those logical segments to the objects created by that pool.

Typically, entire logical segments are placed in a single physical segment, simplifying location of objects in the file. Determining which physical segment contains a given object is then based on the object's logical segment number (part of the object's identifier), allowing for smaller auxiliary tables. Obviously, the pool must still be able to locate the object within the physical segment, but this may be done with implicit or explicit information in the physical segment.

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.

2.5 Buffer Pools

A buffer pool is used to cache Mneme physical segments in main memory. It consists of a chunk of main memory, or buffers, and routines to manipulate that memory. There are eight basic routines, including allocate, free, pin, unpin, reserve, sacrifice, modify reference count, and set modification flag. A buffer can be in one of three states: free, pinned, or unpinned. If a buffer is free, it may be allocated and used immediately. If a buffer is pinned, it is actively in use and may not be allocated. As long as the reference count for a buffer is positive, the buffer is considered pinned. If a buffer is unpinned, it is not actively in use and may be allocated after the owner has been requested to free the buffer. Unpinned buffers are placed on an unpinned list. The order of the buffers on this list dictates the replacement policy used in the buffer pool. Each of the buffer operations are described in more detail below.

Allocate is used to obtain a buffer from the buffer pool. Typically, allocation will be done from the free buffers in the pool. If there are no free buffers, unpinned buffers may be selected for replacement to create a free buffer and satisfy the current request. The allocated buffer will be marked as pinned and returned to the requesting object pool for arbitrary use. Note that initially the reference count is zero even though the buffer is pinned. If there are no unpinned buffers, a more aggressive strategy might attempt to free pinned buffers.

Pin is used to mark an allocated but unpinned buffer as pinned. Similarly, unpin is used mark an allocated but pinned buffer as unpinned. These routines are typically invoked by the modify reference count routine. This routine takes as an argument a (possibly negative) value by which to modify the reference count for a buffer. If the reference count becomes positive and the buffer is unpinned, pin is called. If the reference count goes to zero and the buffer is pinned, unpin is called. Free is used to return a buffer to the buffer pool and is usually called when a buffer has been selected for replacement and the owning pool must free the buffer. Free might also be called by an object pool that no

longer has a use for an allocated buffer. Note that if a buffer contains a modified physical segment, the object pool is responsible for saving the physical segment before freeing the buffer.

Reserve and sacrifice are used to change the priority of an unpinned buffer with respect to the replacement policy. Reserve would be called on a buffer which is going to be referenced in the near future and amounts to moving the buffer to the tail of the unpinned list (buffers are selected for replacement from the head of the list). Sacrifice would be called on a buffer which will not be referenced again and amounts to moving the buffer to the head of the unpinned list.

Set modification flag is used to set, clear, or obtain the status of the modification flag for a buffer. The buffer pool supplies the functionality to maintain this flag, but no semantics are associated with the flag as far as the buffer manager is concerned (i.e., the buffer manager does not act on the status of the modification flag—that is the object pool's responsibility).

A buffer pool is created independently of any object pool and may be shared by multiple object pools. In order to use a buffer pool, an object pool must attach to the buffer pool. Object pools manipulate two types of data in the file: the actual objects managed by the pool, and auxiliary information (e.g., tables) used by the pool to locate its objects in the file. The object pool may attach to different buffer pools for each type of data. When an object pool first attempts to allocate buffer space for either type of data, if it has not already attached to a buffer pool for that data, the attach to buffer pool strategy routine is called. This routine must either attach the object pool to an existing buffer pool or create a new buffer pool and attach to it. The object pool locates existing buffer pools by searching a system supported list of registered buffer pools (when a new buffer pool is created it must be registered with the system). Buffer pools can have attributes associated with them (currently an arbitrary string) that allow object pools to identify appropriate buffer pools.

2.6 Resident Object Table

The Resident Object Table (ROT) keeps track of which objects are resident in the Mneme buffers. The ROT is based on client identifiers, tracking the objects that have been mapped into client identifier space. A *probe* of the ROT will return a pointer to the object in the Mneme buffer if the object is resident, `NULL` if is not resident. A *lookup* of the ROT will always return a pointer to the object in the Mneme buffer. If the object is not resident, the ROT miss strategy routine of the object's pool is called to make the object resident.

The object pools are responsible for making entries in the ROT when objects are brought into main memory, and deleting entries in the ROT when objects are flushed from main memory. Entries are based on logical segments, using the assumption that a logical segment will be either entirely resident or entirely nonresident (since pools typically place entire logical segments in a single physical segment). When a logical segment is made resident, an Object Table Chunk (OTC) is created for the logical segment, which contains a pointer to each object in the logical segment. An ROT probe/lookup actually returns the address of the OTC for the logical segment containing the object. The pointer to the object is obtained by indexing the OTC with the object number. For pools that don't guarantee co-residency of all objects in a logical segment, the individual OTC entries must be verified to point to resident objects.

3 Getting Started

In this section we highlight the operations necessary to create and manipulate a Mneme persistent object store. This includes initialization, file creation, object pool creation, buffer pool creation, object creation, object storage, object retrieval, and termination. All routines will be referred to by name only and we assume a familiarity with the basic concepts from Section 2. See Section 4 for details regarding parameters and return types. A number of example programs that demonstrate the use of many of the routines are included with the distribution in the `mneme/examples` directory.

- **Initialization** Mneme is a library of routines that are linked into the application program. Before the routines can be called to do any work, some run-time initialization must take place. This is accomplished by calling `MnInit()`, which should be called only once each time the application is run. Other routines typically called at initialization time include `MnProfileReset()`, which resets the profiling statistics, and `MnHandlerPush()`, which pushes an exception handler onto the Mneme exception handler stack. Note that these last two functions may be called multiple times and at any time—even before `MnInit()`, but `MnInit()`

may be called only once and no other Mneme functions (besides the two just mentioned) may be called before `MnInit()`.

Next, the object pools that will be used must be initialized. This is accomplished by calling the appropriate strategy initialization routine provided by the object pool, e.g., `MnDpInitStrat()` for the supplied direct pointer pool. Again, this need be done only once for each pool at the beginning of the Mneme session to register the pool's strategy routines with the system. This is also the point at which any customization of a pool's strategy is carried out using `MnStrategySetRoutine()`.

- **File Creation** Before actually storing any objects, a Mneme file must be created to contain the objects. A file is created with `MnFileCreate()`. If the file already exists, it is opened with `MnFileOpen()`. Due to system data structures, an empty Mneme file will be approximately 16 Kbytes long. Primitive concurrency control at the granularity of a file can be achieved with `MnFileLock()`, which will attempt to place an advisory lock on the file.

- **Object Pools** All persistent data stored in a Mneme file is managed by some object pool[1]. When a file is created, Mneme automatically creates a *system* pool for the file to manage some administrative data stored in the file. In order to store any objects in the file, the user must create a *user* object pool to manage those objects. This is accomplished with `MnPoolCreate()`, which will create a pool that uses one of the strategies set up during initialization (see above).

- **Buffer Pools** A buffer pool is created by calling the create routine supplied with a particular buffer pool implementation. Currently, there are two implementations: the default buffer pool and the page buffer pool. The default buffer pool can handle arbitrary buffer sizes (segments) and and one is created with `MnDfltBuffPoolCreate()`. The page buffer pool manages fixed size buffers (pages) and one is created with `MnPageBuffPoolCreate()`. In order to use a buffer pool, an object pool must first attach to the buffer pool using `MnAttachBuffPool()`. This may be done explicitly, or may be handled by the object pool's `AttachBuffPool` strategy routine, which is called if an object pool requests a buffer but is not currently attached to a buffer pool.

- **Object Creation** Once an object pool has been created and its file is open, the object pool's object creation routine can be used to create objects. For example, the supplied direct pointer pool provides `MnDpObjectCreate()` to create new objects. The object creation routines typically return either the client or persistent identifier of the new object and a pointer to the new object in the Mneme buffer. The pointer may be used to manipulate the object and the persistent identifier may be stored in other objects from the same file to create inter-object references to the new object.

- **Object Storage** An object is active and guaranteed to be main memory resident as long as there are outstanding pointers to the object, obtained either at object creation or via one of the retrieval routines described below. To indicate that an object is no longer active and may be removed from main memory, either `MnPidBuffRelease()` or `MnCidBuffRelease()` is used.

- **Object Retrieval** A main memory pointer to an existing object in a file is obtained with `MnPidBuffPtr()` or `MnCidBuffPtr()`. If the object is already resident, a pointer to that object is returned. If the object is not resident, an "object fault" occurs, the object is made resident, and a pointer to the object is returned.

- **Termination** Before program termination, all open Mneme files must be closed with `MnFileClose()`. This will force all modified objects and system data to be flushed to disk. Before a new file is closed, however, the root object of the file should be identified with `MnFileSetRoot()`. This object then becomes the entry point into the file from which all other objects in the file may be reached. The root object in an open file can be obtained with `MnFileGetRoot()`.

4 User Interface

We now describe each of the functions in the Mneme user interface. Most functions return an `RCODE` to indicate the status of the function call. An `RCODE` less than 0 indicates an error of some sort. An `RCODE` of 0, or

[1] Actually, there is a file header and some other low level file management structures that are not managed by a pool

MN_SUCCESS, indicates complete success, and an RCODE greater than 0 indicates success and gives some additional status information. For those functions that return some other type, an error is indicated by an invalid return value (identified for those functions in their descriptions below). In those cases, the kind of error can be obtained from the global RCODE MnStatus. In the event of a generic system error where the error code is MNSYSERR, another global, int MnSysErrNo, will contain the *errno* set by the system.

Mneme supplies a simple exception handling mechanism for dealing with errors. A stack of exception handlers is maintained, and user defined handlers can be pushed onto or popped from the handler stack. When a Mneme function detects an error, an exception is raised with an appropriate error code. This amounts to calling the exception handler at the top of the stack, passing in the error code, and then returning from the function. The handler might print out an error message, attempt to recover from the error, halt execution, simply return the given error code, or defer to the next handler in the stack of exception handlers. The default exception handler simply returns the given error code. Currently, if a Mneme function detects an error raised by another Mneme function, the first function (the caller) raises another exception with the same error code returned by the second function (the callee). The implication of this is that, assuming the default handler is at the top of the handler stack, the error code representing the initial error will be propagated back up through the user interface function out to the user. As such, the user interface functions below could potentially return any of a large number of error codes.

Rather than document all of the error codes potentially returned by each function below, we classify errors into two categories: system errors and bugs. System errors are caused by external forces, such as resource failures (e.g., disk full, main memory exceeded, etc.). These errors are potentially recoverable, so after each function prototype below, we list the system errors that might be returned. Bugs include mistakes in the Mneme library as well as mistakes in the user code that calls the library routines. Rather than list all possible bug related error codes for each function, we list only error codes generated due to bad parameters passed into the interface functions. These appear in italics before any system error codes. For reference, we list all possible error codes in the appendix. We also suggest the use of the abort handler in combination with a debugger to track down bug related errors. The exception handler stack manipulation routines and provided handlers are described in Section 4.8.

4.1 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.

- RCODE MnInit (void)
  Errors: MNMEMREQFAIL

  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 FNAMEs 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.

- RCODE MnFileExists (IN FNAME *fname)
  Errors: *MNNULLFNAME*, MNFILENACCESS, MNSYSERR

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

- `RCODE MnFileCreate (IN FNAME *fname, OUT FILEID *fileId)`
  Errors: *`MNNULLFNAME, MNPTRISNULL`*, `MNMEMREQFAIL, MNFILENACCESS, MNDQUOT, MNFILEEXISTS, MNFNOSPC, MNTOOMANYFILES, MNFTOOBIG, MNFILENEXIST, MNSYSERR`

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

- `RCODE MnFileDestroy (IN FNAME *fname)`
  Errors:

  Destroy the named file.

- `RCODE MnFileRename (IN FNAME *oldName, FNAME *newName)`
  Errors:

  Rename the file.

- `RCODE MnFileOpen (IN FNAME *fname, OUT FILEID *fileId)`
  Errors:

  Open the named file, assigning it a file id.

- `RCODE MnFileId (IN FNAME *fname, OUT FILEID *fileId)`
  Errors:

  Determine the file id of the named file. Operation fails if the file does not exist or it is not open.

- `RCODE MnFileIsOpen (IN fname)`
  Errors:

  Returns `MN_TRUE` if the named file is open, `MN_FALSE` otherwise. Note that no check is made to verify that the file exists.

The remaining file operations are oriented towards file ids.

- `RCODE MnFileLock (IN FILEID fileId, FLOCK_OPER oper, BOOL block)`
  Errors:

  Perform the lock operation specified by `oper` on the file identified by `fileId`. The possible `oper` values are:

`MN_FILE_RDLCK`	obtain a read lock on the file (shared use)
`MN_FILE_WRLCK`	obtain a write lock on the file (exclusive use)
`MN_FILE_UNLCK`	unlock the file

  `block` is a boolean flag indicating whether or not to block in the event that the desired lock cannot be obtained. This function is based on the `fcntl` system call and will work with files shared via NFS.

- `RCODE MnFileClose (INOUT FILEID *fileId)`
  Errors:

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

- `RCODE MnFileCloseAll (void)`
  Errors:

  Close all currently open files.

- `RCODE MnFileForce (IN FILEID fileId, BOOL sync, BOOL release)`
  Errors:

  Force any pending writes to take place for the indicated file. The file remains open. If `sync` is true, we guarantee that the disk has been updated (i.e., the file system buffers have been forced). If `release` is true, the buffers occupied by the file are freed.

- `RCODE MnFileName (IN FILEID fileId, OUT FNAME *fname)`
  Errors:

  Determine the name of the identified file. Operation fails if the given file identifier is not valid.

- `RCODE MnFileGetPools (IN FILEID fileId, OUT POOLID *(poolIds[]), int *count)`
  Errors:

  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.

- `RCODE MnFileSetRoot (IN FILEID fileId, CID cid)`
  Errors:

  Set the root object of the file from the client identifier.

- `RCODE MnFileGetRoot (IN FILEID fileId, OUT CID *cid)`
  Errors:

  Return the client identifier of the root object for the given file.

4.2 Identifier Operations

The following operations convert between persistent and client object identifiers and map object identifiers to pool and file identifiers.

- `POOLID MnPidToPoolId (IN PID pid, FILEID fileId)`
  Errors:

  Return the pool id of the pool that owns the object identified by the given persistent identifier and file id.

- `POOLID MnCidToPoolId (IN CID cid)`
  Errors:

  Return the pool id of the pool that owns the object identified by the given client identifier.

- `CID MnPidToCid (PID pid, FILEID fileId)`
  Errors:

  Return the client identifier for the object identified by the given persistent identifier and file id.

- `PID MnCidToPid (IN CID cid)`
  Errors:

  Return the persistent identifier for the object identified by the given client identifier.

- `FILEID MnCidToFileId (IN CID cid)`
  Errors:

  Return the file id for the object identified by the given client identifier.

4.3 Object Pointer Operations

An object is manipulated by obtaining a pointer to the object. This is achieved by calling `MnCidBuffPtr` or `MnPidBuffPtr`. If the object is already resident in a Mneme data buffer, these routines return the main memory address of the object and increment the reference count associated with the buffer. If the object is not resident, the object's pool must first make the object resident. Since the objects are manipulated directly by pointers, the client must inform Mneme if an object has been modified. This can be done when a pointer to the object is first obtained, when the object is released, or any time in between. An object is guaranteed to be resident between the time when a pointer to the object is first obtained and the time when the object is released. Each pointer acquisition must have a corresponding release that decrements the buffer's reference count, such that pointer acquisition and object release are paired operations. However, the `cnt` parameter to the release calls may be used to pair a single release call with multiple pointer acquisitions. Each operation has two versions: one that accepts a client identifier and one that accepts a persistent identifier and file id.

- `PTR MnCidBuffPtr (IN CID cid, int mod)`
  Errors:

  `PTR MnPidBuffPtr (IN PID pid, FILEID fileId, int mod)`
  Errors:

  Return a pointer to the identified object. If it is known in advance that the object will be modified, the `mod` flag should be true. The reference count associated with the buffer that contains the object is incremented by 1.

- `RCODE MnCidBuffMod (IN CID cid)`
  Errors:

- `RCODE MnPidBuffMod (IN PID pid, FILEID fileId)`
  Errors:

  Mark the identified object as modified.

- `RCODE MnCidBuffRelease (IN CID cid, int mod, int cnt)`
  Errors:

  `RCODE MnPidBuffRelease (IN PID pid, FILEID fileId, int mod, int cnt)`
  Errors:

  Release the identified object. `mod` indicates if the object has been modified and `cnt` indicates how many pointer acquisitions for the given object the release should be paired with. Note that if the object was previously flagged as modified, setting `mod` to false here does *not* clear that flag. The reference count associated with the buffer that contains the object is decremented by `cnt`.

4.4 Pool and Strategy Operations

Recall that an object's pool determines its management policy. The following routines permit the creation of pools and tailoring of the strategy routines.

- `POOLID MnPoolCreate (IN FILEID fileId, STRATID strategyId)`
  Errors:

  Create a new pool in the indicated file, having the given strategy.

- `RCODE MnStrategySetRoutine (IN STRATID stratId, STRAT_EVENT stratEvent, RCODE (*routinePtr)())`
  Errors:

  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.

- `POOLID MnPoolNumId (IN FILEID fileId, POOLNUM poolNum)`
  Errors:

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

- `POOLNUM MnPoolIdNum (IN POOLID poolId)`
  Errors:

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

4.5 The Supplied Pools

A number of pool implementations are currently supplied with Mneme. A pool is defined by its strategy and a strategy is embodied by the routines that make up that strategy. Therefor, building a new pool is really a matter of assembling a number of routines and defining a new strategy. From the user's perspective, the only visible pool routines are (usually) the strategy initialization routine and the object creation routine. The rest of the routines that make up the strategy are invoked by Mneme as a result of the user calling one of the Mneme interface routines. For the pool builder (often referred to as the "sophisticated user" in Mneme terminology), a number of routines are provided in the "standard" pool that can be used as the foundation for a new pool. In fact, any existing pool strategy can be used to construct new pools. Here we simply describe the user interface to the supplied pools. More detailed discussions of pool strategy construction will appear elsewhere in future revisions of this document.

4.5.1 Direct Pointer Pool

The direct pointer pool defines an object format and supports variable size objects (which may result in variable size physical segments) manipulated by direct pointers. The object format divides the contents of an object into *slots* and *bytes*, where slots contain the persistent identifiers of other objects and bytes contain arbitrary data. An object consists of a header followed by the slots (if any) and then the bytes (if any). The header specifies the number of slots and bytes and supports a one byte attribute that the user may set and examine arbitrarily. The details of the object format may be found in the file `pool_dirptr.h`. A pointer to an object will point to the first slot in the object or, if there are no slots, the first byte. The strategy id of the pool is `DIRPTR_STRAT`. The user routines associated with the pool are:

- `RCODE MnDpInitStrat (void)`
  Errors:

  Initialize the direct pointer pool strategy, `DIRPTR_STRAT`.

- `PTR MnDpObjectCreate (IN POOLID poolId, LONG numSlots, LONG numBytes, int attr, CID nearCid, OUT CID *cid)`
  Errors:

  Create a new object in the direct pointer pool identified by `poolId`. `numSlots` and `numBytes` specify the number of slots and bytes, respectively, that the object should contain. `attr` is the initial value for the one byte attribute in the object. `nearCid` may specify the client identifier of another object in the same pool near which the new object should be clustered. Use `EMPTY_ID` if no near object is to be specified. `cid` is the address of a `CID` into which the client identifier of the new object will be stored. The routine returns a pointer to the new object.

- `RCODE MnDpPtrInfo (IN PTR ptr, OUT LONG *numSlots, LONG *numBytes)`
  Errors:

  Set `numSlots` to the number of slots and `numBytes` to the number of bytes in the object pointed to by `ptr`.

- `RCODE MnDpPtrAttrs (IN PTR ptr, BYTE andMask, BYTE xOrMask, OUT BYTE *result)`
  Errors:

  Modify the byte attribute for the object pointed to by `ptr`. The existing attribute is 'and'ed with `andMask`, 'xor'ed with `xOrMask`, and the resulting attribute is stored in the object and returned in `result`.

4.5.2 Cross File Reference Pool

The cross file reference pool supports inter object references bewteen objects in different files. Since only persistent identifiers may be stored in objects, and persistent identifiers are unique only within a file, we need a special mechanism to store a reference to an object from another file. This is done by creating a "surrogate" object in the source file that represents the referent object in the other file. The surrogate object is created in a cross file reference pool of the source file. When a surrogate object fault occurs, the cross file reference pool opens the file containing the referent object (if necessary) and causes the referent object to be faulted in. Note that the referent object is in fact faulted in by the pool that manages that object in the other file. The result of the object fault returned by the referent object's pool (most

likely a pointer to the object) is then returned by the cross file reference pool as the result of the surrogate object fault. Since the cross file reference pool can open other files in the process of servicing object faults, `MnFileCloseAll()` should be used at the end of a Mneme session to ensure that all files are closed. The routines are:

- `RCODE CfrInitStrat (void)`
  Errors:

  Initialize the cross file reference pool strategy.

- `RCODE CfrObjectCreate (IN POOLID poolId, PID cfrPid, FILEID cfrFileId, OUT PID *pid)`
  Errors:

  Create a cross file reference. `poolId` is the id of a cross file reference pool in the file containing the source object. `cfrPid` is the persistent identifier of the referenct object in the other file. `cfrFileId` is the file id of the file containing the referent object. Note that the file must be open. `pid` returns the persistent identifier of the surrogate object in the source file.

4.6 Buffer Pools

Buffer pools are created using the buffer pool create routine supplied by the buffer pool implementation. Buffer pools are identified with a `BPID`. Each buffer pool is registered with the system and a global list of buffer pools is maintained that may be searched by an object pool in order to locate suitable buffer pools. Buffer pools have attribute strings associated with them for identification purposes. The generic buffer pool operations are:

- `RCODE MnAttachBuffPool (IN BPID bpid, POOLID poolId, BP_SELECTOR bp_type)`
  Errors:

  Attach the object pool identified by `poolId` to the buffer pool identified by `bpid`. `bp_type` specifies which type of persistent information the object pool will use the buffer pool for, i.e., system (`SYS_BP`) or data (`DATA_BP`).

- `BPID MnDetachBuffPool (IN POOLID poolId, BP_SELECTOR bp_type)`
  Errors:

  Detach the object pool identified by `poolId` from its `bp_type` buffer pool. The identifier of the buffer pool from which the object pool was detached is returned.

4.6.1 Default Buffer Pool

The default buffer pool provides arbitrarily sized buffers. Free space is managed with a splay tree resulting in a best fit policy for allocation from the free list. If the internal fragmentation of an allocated buffer exceeds some threshold, the unused portion is split off and returned to the free list. Neighbor coalescing is done when buffers are returned to the buffer pool. The replacement policy for unpinned but allocated buffers is least-recently-used (LRU). The associated routines are:

- `BPID MnDfltBuffPoolCreate (IN int size, char *attr_str)`
  Errors:

  Create a default buffer pool of the given size with the given attribute string. The identifier of the new buffer pool is returned.

- `RCODE MnDfltBuffPoolDestroy (IN BPID *bpid)`
  Errors:

  Destroy the default buffer pool identified by `bpid`. All main memory allocated to the buffer pool and associated data structures is freed.

4.6.2 Page Buffer Pool

The page buffer pool manages a buffer of fixed size pages. The page size is specified at buffer pool creation time. The replacement policy for unpinned but allocated buffers is least-recently-used (LRU). The associated routines are:

- `BPID MnPageBuffPoolCreate (IN int page_size, int num_pages, char *attr_str)`
  Errors:

  Create a page buffer pool with the given number of pages, page size, and attribute string.

- `RCODE MnPageBuffPoolDestroy (IN BPID *bpid)`
  Errors:

  Destroy the page buffer pool identified by `bpid`. All main memory allocated to the buffer pool and associated data structures is freed.

4.7 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.

- `RCODE MnFileInfoPrint (IN FILEID fileId)`
  Errors:

  Print allocation information about the file.

- `RCODE MnFileInfoCopy (IN FILEID fileId, OUT FILEDATA *fdata)`
  Errors:

  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.

- `RCODE MnProfileReset ()`
  Errors:

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

- `RCODE MnProfilePrint ()`
  Errors:

  Print the current profiling counts.

- `RCODE MnProfileCopy (PROFDATA *pdata)`
  Errors:

  Sample the current profiling counts.

4.8 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 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.

- `RCODE MnHandlerPush (HANDLER *handler)`
  Errors:

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

- `RCODE MnHandlerPop ()`
  Errors:

  Pop a handler from the stack.

Besides the default handler, two other handlers are supplied:

- `MnReturnHandler` – prints a simple error message and returns the error code.
- `MnAbortHandler` – prints a simple error message and calls `abort` so that execution is *not* allowed to continue. This handler is most useful in conjunction with a debugger, where execution will stop at the point where the error is detected and the state of the program can be examined.

A. Error Codes

Error Code	Description
MN_SUCCESS	Operation completed successfully
MN_FAILURE	Operation failed
MN_TRUE	Boolean true
MN_FALSE	Boolean false
MNUNIMPL	Operation not yet implemented
MNPTRISNULL	Pointer argument to function is null
MNSEGWRITE	Unable to write physical segment
MNSEGREAD	Unable to read physical segment
MNBADFILEID	File identifier is invalid
MNEMPTYSTRAT	Pool strategy is empty
MNPOOLCONFLICT	Pool conflict between new and near object
MNBADOBJECT	Incorrectly specified object (id is invalid)
MNBADSTRATID	Strategy identifier is invalid
MNTOOMANYPOOLS	Exceeded pool limit for file
MNBADCID	Invalid client object identifier
MNBADSLOTNUM	Slot number is out of range
MNBADBYTENUM	Byte number is out of range
MNBADPOOLID	Pool identifier is invalid
MNEMPTYOPT	Option is empty
MNBADEVENT	Bad strategy event
MNNOPROFILE	Profiling is disabled
MNPOPHANDLER	Tried to pop default handler
MNBADPID	Invalid persistent id
MNBADPTR	Bad direct object pointer
MNRELEASE	Release of non-resident object
MNCIDFULL	CID space is full
MNBUFFAIL	Buffer request failed
MNSEGEXIST	segment does not exist
MNBADLSEG	lseg not found in pool
MNMEMREQFAIL	memory request failed
MNLSGRP	no more lsid groups free in file
MNLSEGNOTMAP	lseg not mapped into cid space
MNEMPTYEVENT	empty event in pool strategy

Error Code (cont)	Description
MNBUFSIZE	bad size in mn_alloc request
MNLOOKUPOTC	lseg not found after object fault
MNUPDATEOTE	ote to update not found
MNNOBP	pool not attached to buffer pool
MNPOOLFULL	no more room for objects in pool
MNBADBPID	bad buffer pool id
MNMARKMOD	attempt to mark non-resident object modified
MNFILEEXISTS	file already exists
MNFILENEXIST	file does not exist
MNFILENOPEN	file not open
MNFILENCLOSED	file not closed
MNTOOMANYFILES	too many files
MNFILENACCESS	file access denied
MNSEGSIZE	segment size mismatch
MNNEGSEGLEN	negative segment length
MNNULLFNAME	null filename
MNSEGGROW	segment grow failed
MNGETHDR	get file header failed
MNPUTHDR	put file header failed
MNGETBITMAP	get file bitmap failed
MNPUTBITMAP	put file bitmap failed
MNADDBITMAP	add file bitmap failed
MNBADGRAN	bad granularity on file create
MNBADHDRSZ	req hdr size != stored hdr size
MNSYSERR	system error
MNBADFD	bad file descriptor in fid
MNDQUOT	disk quota exceeded
MNFNOSPC	no space for new file
MNFTOOBIG	write caused file to exceed size limit
MNROTENFOUND	rot entry not found
MNTABLEFULL	a Mneme table is full
MNPBSIZE	bad page size for page buffer pool
MNALIGN	given size not aligned
MNBADFLO	bad file lock operation
MNSEGFETCH	Segment fault occurred
MNCFRFAULT	cross file reference fault
MNWOULDBLCK	can't get desired file lock