Chapter 4: API Definition

Changes Since Version 1.0a4

Aside from the introduction of the additional reference manipulation routines, the API changes are essentially refinements of the 1.0a4 spec.

Minor changes to the API are not listed here, but are marked with change bars in the body of the chapter. Purely expository changes are not marked at all.

Addition of Reference Routines

Four reference manipulation routines have been added to the API: `CMSetReference`, `CMDeleteReference`, `CMCountReferences`, and `CMGetNextReference`. In addition, for consistency, `CMGetReferenceData` has been renamed `CMNewReference`; its interface is unchanged.

Change to the Error Handler Prototype

The error handler prototype has been changed to take an error number and a variable number of strings rather than a single error string. This allows easier classification of errors reported to the handler.

Change Generation Number to 32 Bits

CMGeneration has been changed to 16 bits to 32 bits. A number of Bento reviewers had expressed a concern that a significant number of Bento containers might have lifetimes of more than 2^{16} generations. This change will generally impose no additional TOC overhead because of the new TOC format.

Container Encoding Control Eliminated

The encoding argument has been eliminated from CMOpenNewContainer. All container TOCs will have little-endian encoding.

AbortSession and AbortContainer Calls Added

Calls to abort a session or a container cleanly after unrecoverable errors were added.

Design Comments

Portability

The actual API headers are annotated with macros to support the various declarators needed for different systems and especially different C compilers. These annotations have been removed from the API as presented in this chapter of the specification.

We have carried out extensive testing to make sure that Bento will run in as many environments as possible.

Declaration Style

Declarations are deliberately made in a platform-independent manner. A mapping from the declarations as given to a specific platform will be required for each implementation.

Names specific to the API are prefixed with "CM" (Container Manager).

Error Reporting

The API calls an error handler provided in the initialization call to report errors. This handler could do a longjump, or it could use a more sophisticated error reporting scheme.

Error Codes

The error codes returned by the operations are defined in Appendix C. This list will continue to grow with extensions to the design. The list of errors possible from each operation is not yet included in the spec.

Types and Constants

Low level basic types

```
typedef char     CHAR;
typedef unsigned char     UCHAR;
```

Signed and unsigned1-byte values.

```
typedef short    SHORT;
typedef unsigned short    USHORT;
```

Signed and unsigned2-byte values.

```
typedef long    LONG;
typedef unsigned long    ULONG;
```

Signed and unsigned 4-byte values.

Types

All Container Manager types are defined here with the intent to aid in enforcing compiler type checking. Note that "refNums" for session data, container control blocks, values, and objects are uniquely typed to strictly enforce type checking of those entities. These types are defined as pointers derived from an "incomplete type", i.e. structs. The structs are **not** defined. In ANSI C, an "incomplete type" need not be defined.

```
typedef struct                          CMSession_ *CMSession;
```

Pointer to session (task) data.

```
typedef struct                          CMContainer_ *CMContainer;
```

"RefNum" for containers.

```
typedef struct                          CMObject_ *CMObject;
```

"RefNum" for objects.

```
typedef CMObject                        CMProperty;
```

"RefNum" for property description objects.

```
typedef CMObject                        CMType;
```

"RefNum" for type description objects.


```
typedef struct                          CMValue_ *CMValue;
```

“RefNum” for values.

```
typedef CHAR                            *CMOpenMode;
```

Handler open mode string pointers.

```
typedef CHAR                            *CMGlobalName;
```

Global unique name pointers.

```
typedef CHAR                            *CMErrorString;
```

Error message string pointers.

```
typedef CM_CHAR                          *CMMetaData;
```

Type metadata string pointers.

```
typedef void                            *CMRefCon;
```

Reference constants ("refCon"s).

```
typedef void                             *CMPtr;
```

Arbitrary data pointers.

```
typedef UCHAR                           *CMMagicBytes;
```

Magic byte pointers.

```
typedef CM_UCHAR                        *CMDataPacket;
```

"New value" handler data packets.

```
typedef CM_UCHAR                        *CMDataBuffer;
```

Ptr to data buffer for handlers.

```
typedef CM_UCHAR                        *CMPrivateData;
```

Ptr to private CM data for handlers.

```
typedef CM_UCHAR                        CMReference[4];
```

Referenced object data pointers.

```
typedef UCHAR                           CMSeekMode;
```

Container "fseek()" handler modes.

```
typedef UCHAR                           CMBoolean;
```

Boolean funct. results (0==>false).

```
typedef USHORT                          CMContainerUseMode;
```

Container open use mode flags.

```
typedef USHORT                          CMContainerFlags;
```

Container label flags.

```
typedef CM_USHORT                    CMContainerModeFlags;
```

Container open mode flags.

```
typedef USHORT                       CMEofStatus;
```

"feof()" handler result statusc

```
typedef CM_LONG                      CMErrorNbr;
```

Error handler error numbers.

```
typedef ULONG                        CMGeneration;
```

Container generation numbers.

```
typedef ULONG                        CMSize;
```

Sizes

```
typedef ULONG                        CMCount;
```

Amounts or counts.

```
typedef void                         *CMPtr;
```

Arbitrary data pointers.

```
typedef void                         ( *CMHandlerAddr)();
```

Handler address pointers.

```
typedef  CMHandlerAddr ( *CMMetaHandler)
                    (CMType,
                    const CMGlobalName);
```

Metahandler prototype.

Constants

The following flags are passed to `CMOpen[New]Container()`. They modify the open in the indicated ways. Note that kCMReading, kCMWriting, kCMUpdating are also returned from `CMGetContainerInfo()` to indicate the mode of the container, i.e. it was opened for reading, writing, or updating.

```
const CMContainerUseMode             kCMReading = 0x0001
```

Container was opened for reading.

```
const  CMContainerUseMode            kCMWriting = 0x0002
```

Container was opened for writing.

```
const CMContainerUseMode             kCMReuseFreeSpace = 0x0004
```

Try to reuse freed space.

```
const CMContainerUseMode             kCMUpdateByAppend = 0x0008
```

Open container for update-by-append.

```
const CMContainerUseMode             kCMUpdateTarget = 0x0010
```


Open container for updating target.

```
const CMContainerUseMode            kCMConverting = 0x0020;
```

Open a container for "converting".

The following flags are options to the "seek" I/O handler.

```
const CMSeekMode                    kCMSeekSet = 0x00;
```

"fseek()" handler mode (pos).

```
const CMSeekMode                    kCMSeekCurrent= 0x01;
```

"fseek()" handler mode (curr+pos).

```
const CMSeekMode                    kCMSeekEnd = 0x02;
```

"fseek()" handler mode (end+pos).

Operation Definitions

Session Operations

The session is an explicit object created by initializing the library. It represents private Container Manager data that is global to all open containers. The intent is that this data is unique to the currently running session (or task). The caller may extend this data to include his or her own special per-session information.

```
CMSession  CMStartSession(CMMetaHandler metaHandler,
                          CMRefCon sessionRefCon)
```

This call is used for all global initialization of the Container Manager. It **must** be called before any other Container Manager routine and should only be called once. If not, every API routine will try to exit without doing anything.

An anonymous non-`NULL` pointer is returned if initialization is successful. `NULL` is returned for failure unless the error reporter (discussed below) aborts execution.

This routine takes as its main parameter the address of a metahandler. This metahandler must define operations for error handling, memory allocation, and memory deallocation. The interface to the metahandler and to the three specific handlers is documented in Appendix B.

In addition the caller can pass a “reference constant” (refCon) as the last parameter to this routine. It is saved in the session data. The refCon is not used by the API and can be anything, but usually it will be a pointer to the caller's own session data.

```
void  CMEndSession(CMSession sessionData,
                   CMBoolean closeOpenContainers)
```

This should be called as the **last** call to the Container Manager. It frees the space allocated for the session by `CMStartSession()` and optionally calls `CMCloseContainer()` on all remaining open containers.

The sessionData specifies the session data pointer returned from `CMStartSession()`. If `closeOpenContainers` is passed as 0 (i.e., "false"), then an error is reported for each container that has **not** been explicitly closed by `CMCloseContainer()`. If true (non-zero) is specified, then the Container Manager will call `CMCloseContainer()` for you for each remaining open container.

No further calls should be done once this routine is called. All memory occupied by the containers, as well as the session itself are freed.

```
void  CMAbortSession(CMSession sessionData);
```

This is basically a `CMAbortContainer()` for all currently open containers followed by a `CMEndSession()`. This routine **will** return to its caller. It is up to the user to actually abort execution if that is required. This call is intended to be used to abort the session from unrecoverable errors.

All containers are closed without further writing to those containers, i.e., as if all containers were opened for reading even when opened for writing. All memory allocated by all the container data structures are freed (if possible) and the container close handlers called to physically close the containers. All dynamic values currently in use are released in an attempt to allow them to properly clean up any files and memory allocated by their handlers. No further API calls should be done.

```
CMRefCon  CMGetSessionRefCon(CMContainer container)
```

This routine can be used to get at the user's session refCon saved as part of the session data created by `CMStartSession()`. The session data is "tied" to each container created by `CMOpen[New]Container()`. Thus typically the refCon will be accessed via a container refNum.

```
void  CMSetSessionRefCon(CMContainer container,
                         CMRefCon refCon)
```

This routine may be called to change the user's session refCon associated with the session data.

```
CMHandlerAddr  CMSetMetaHandler(const CMSession sessionData,
                         const CMGlobalName typeName,
                         CMMetaHandler metaHandler)
```

This routine records the association of Global Names with their metahandlers.

The designated metahandler will be associated with the `typeName`. The previous metahandler for this type name, if any, is returned. If there was no previous metahandler defined, `NULL` is returned. The association between handlers and type names is global within a session, rather than specific to a given container.

A metahandler will be called whenever Bento or the application needs to find out how to perform a given operation on a container or value of this type. The metahandler can define specific handlers for any number of different operations, potentially with completely different interfaces. The interface to the metahandler is documented in the section on handler interfaces.

This routine must be used to associate a type name with a metahandler before `CMOpen[New]Container()` is called, so that the Container Manager can find the appropriate metahandler for the container.

```
CMHandlerAddr  CMGetMetaHandler(const CMSession sessionData,
                    const CMGlobalName typeName)
```

This function searches the metaHandler symbol table for the specified `typeName` and returns the associated metahandler address. If no metahandler is associated with that type name, it returns `NULL`.

```
CMHandlerAddr CMGetOperation(CMType targetType,
                    const CMGlobalName operationType);
```

This routine takes a `targetType` which has a globally unique name and uses that name to find a metahandler. The metahandler, in turn, is called to get the handler routine address for the specified `operationType`. The function returns the resulting address.

Metahandler proc addresses are given to the Container Manager by calls to `CMSet-MetaHandler`. The global name for the input `targetType` is treated as the `typeName` to find the metahandler.

See Appendix B for more information on the hander mechanism.

Container Operations

Containers (files and blocks of memory) are always accessed through handlers, to provide platform independence and support nested containers. Handlers are responsible for creating a container if necessary, opening and closing it, managing stream I/O to it, and reading and writing the container label (which provides such information as the location of the Table of Contents). The interfaces to container handlers are documented in the Appendix B.

The types of storage that can be used as containers are limited only by the types of handlers available.

```
CMContainer  CMOpenContainer(CMSession sessionData,
                    CMRefCon attributes,
                    const CMGlobalName typeName,
                    CMContainerUseMode useFlags);
```

This operation opens an existing Bento container.

The `attributes` must designate management structures for the container storage. This `attributes` argument is not examined by Bento, but is simply passed to the appropriate handler interfaces. It is intended to provide the information necessary for the handlers to locate a specific container. Thus `attributes` serves as a communication channel between the application and the Open handler. In its simplest form for a container file it would be a pathname. For an embedded container, it would be the parent value (CMValue), corresponding to the embedded container.

The typeName is used to find a metahandler defined for that same typeName. The metahandler, in turn, defines the handlers for the container and thus knows how to get at the physical container. These handers must understand the attributes provided.

The useFlags must be 0 or kCMReuseFreeSpace. 0 implies that the container is to be open for reading only. No writes may be done. If kCMReuseFreeSpace is specified, than **both** reading and writing may be done to update the container. Free space from deleted data will be reused and overwrites of existing data may be done to change it (subject to the container label flags, see below).

A container refnum is returned.

Note that an individual value can be opened as an embedded container. Through the attributes, the value is passed to the handlers. This value must be typed as an embedded container value. Embedded containers can have embedded containers which can also be opened and read. The effect is that a tree of nested containers can be opened and read without restriction. However, when a CMCloseContainer() is done on a parent container, all of its descendents will also be closed.

```
CMContainer  CMOpenNewContainer(CMSession sessionData,
                    CMRefCon attributes,
                    const CMGlobalName typeName,
                    CMContainerUseMode useFlags,
                    CMGeneration generation,
                    CMContainerFlags containerFlags,
                    ...);
```

This operation opens a new Bento container for writing. This is similar to opening for reading (see documentation above) except that here a new and empty container is opened. A minimum TOC is created along with the special TOC object 1 with its seed and offset properties.

The resulting container can be updated.

In addition to kCMReuseFreeSpace, the useFlags may be 0, kCMConverting, kCMUpdateByAppend, or kCMUpdateTarget.

generation is the generation number of the container; it must be ≥ 1. If this container is a copy of a previous container, the generation number should be 1 greater than the generation number of the previous container.

containerFlags is the flag value that will be stored in the container label. No container flags are currently defined.

If the kCMConverting flag is set in CMContainerUseMode, the physical container is assumed to already contain a sequence of bytes that the caller wants to convert to container format. The application uses CMDefineValueData() to create values for objects in the bytes. All new stuff, including the TOC is written at the end of the existing stuff. Bento will not modify the existing data.

If the kCMUpdateByAppend or kCMUpdateTarget flags are set, all updates to a "target" container are recorded in the container being opened. Future opens of this container, with CMOpenContainer() will apply the updates to the target to bring it "up-to-date" while it is open.

If kCMUpdateByAppend is specified, then the container is opened for update-by-append. All updates are appended to the existing container and an additional TOC is layered on to the end of the container when closed. Each time the container is opened and then closed for update-by-append, the new updates and a new TOC are appended. Whenever such a container is opened (in any mode), all the updates are applied appropriately to the original container.

Using kCMUpdateTarget is similar to kCMUpdateByAppend, but the updates are recorded in a new container.

In both cases the "target" container is specified in a type-dependent way, using the CMRefCon and the "..." parameters passed to CMOpenNewContainer(). These parameters are interpreted in exactly the same way as the corresponding parameters of CMNewValue(); see the documentation of CMNewValue() for further details on the "..." parameters.

A container refnum is returned.

Just as in reading, any number of embedded containers can be opened. Also embedded containers can be opened within embedded containers to any depth. The effect is that a tree of nested containers can be opened and written without restriction. However, when a CMCloseContainer is done on a parent container, all of its descendents will also be closed.

It is an error to call CMOpenNewContainer with a value that belongs to a container that is not updatable, since that call would create an embedded container open for writing.

```
void CMGetContainerInfo(const CMContainer container,
                        CMGeneration *generation,
                        CMContainerFlags *containerFlags,
                        CMGlobalName typeName);
```

The corresponding values for the designated container are returned. NULL may be passed for any reference; in that case the corresponding value is not returned.

```
CMSession  CMGetSession(CMContainer container)
```

The session global data pointer returned from CMStartSession() is passed to most of the handler routines defined in this file. This routine is provided to make it easier to retrieve the pointer as a function of the container refNum.

```
VOID CMCloseContainer(CMContainer container);
```

If the container was open for writing, all I/O to the designated container is completed, and the table of contents and label are built and written out.

This call closes the specified container and **all** its currently opened embedded containers (if any).

This call destroys the association between the container refnum and the designated container. On return the specifed container refNum and all the others corresponding to the embedded containers are invalid. All memory associated with a container's data structures is freed. After this call the container refNum may be returned by a subsequent `CMOpenContainer` call, designating another container.

```
VOID CMAbortContainer(CMconst_CMContainer container);
```

The container is closed without further writing to the container, i.e., as if it were opened for reading even when opened for writing. This is intended to be used to abort processing of the container from unrecoverable errors.

All memory allocated by the container data structures is freed (if possible) and the container close handler called to physically close the container. All dynamic values currently in use are released in an attempt to allow them to properly clean up any files and memory allocated by their handlers. No further API calls should be done on the container as it will be closed upon return.

Note, this routine will return to its caller. It is up to the user to actually abort execution if that is required.

Type and Property Operations

All types and properties must be registered before they can be used. The operations behave the same on standard types and properties as on normal types and properties. However, standard types and properties will not actually be given TOC entries for their descriptions just because they are registered. If additional, non-standard properties are added to the description of a standard type or property, they will be stored.

The refnum returned from registration can be used in exactly the same manner as an object refnum in the object and value operations.

Types and properties may be registered more than once; the refnum returned from all the different registrations of the same type is the same. Identity of types is defined by string equality of their names.

```
CMType CMRegisterType(CMContainer targetContainer,
                      const CMGlobalName name);
```

The designated type is registered in the designated container, and a refnum for it is returned. If a type with that name already exists, the refNum for it is returned.

Standard types may be registered, but this is not required.

```
CMProperty CMRegisterProperty(CMContainer targetContainer,
                      const CMGlobalName name);
```

The designated property is registered in the designated container, and a refnum for it is returned. If a property with that name already exists, the refNum for it is returned.

Standard properties may be registered, but this is not required.


```
CMBoolean CMIsType(CMObject theObject);

CMBoolean CMIsProperty(CMObject theObject);
```

These operations test the designated object and return non-zero if it is a type description or a property description, respectively, otherwise 0.

```
CMType CMGetNextType(CMContainer targetContainer,
                     CM CMType currType);
```

A refnum for the next type registered in the same container is returned. `currType` is generally a refNum previously returned from this routine. Successive calls to this routine will thus yield all the type descriptions in the container.

Types are returned in order of increasing ID. If there are no larger type IDs registered, `NULL` is returned. To begin the iteration, pass `NULL` as the type refnum.

```
CMProperty CMGetNextProperty(CMContainer targetContainer,
                     CMProperty currProperty);
```

A refun for the next property registered in the same container is returned. `currProperty` is generally a refNum previously returned from this routine. Successive calls to this routine will thus yield all the property descriptions in the container.

Properties are returned in order of increasing ID. If there are no larger property IDs registered, `NULL` is retured. To begin the iteration, pass `NULL` as the property refnum.

```
CMCount  CMAddBaseType(CMType type,
                     CMType baseType)
```

This routine defines base types for a given type so that layered dynamic values can be created. Base types essentially provide type inheritance. See the chapter on Types and Dynamic Values for a full description of how base types are used.

A base type is added to the specified type. For each call to `CMAddBaseType()` for the type a new base type is recorded. They are recorded in the order of the calls. The total number of base types recorded for the type is retuned. 0 is returned if there is an error and the error reporter returns.

It is currently an error to attempt to add the **same** base type more than once to the type.

```
CMCount  CMRemoveBaseType(CMType type,
                     CMType baseType)
```

The specified base type previously added to the specifed type by `CMAddBaseType()` is removed. If `NULL` is specified as the baseType, **all** base types are removed. The number of base types remaining for the type is returned.

Note, no error is reported if the specified base type is not present or the type has no base types.

Object Operations

```
CMObject CMNewObject(CMContainer targetContainer);
```

A refnum to a new object in the designated container is returned. At this point the object has nothing but an identity.

```
CMObject CMGetNextObject(CMContainer targetContainer,
                    CMObject currObject);
```

A refum for the next object defined in the same container is returned. currObject is generally a refNum previously returned from this routine. Successive calls to this routine will thus yield all the objects in the container.

Objects are returned in order of increasing ID. If there are no larger object IDs defined, `NULL` is returned. To begin the iteration, pass `NULL` as the object refnum.

Since type and property descriptions are objects, they will be returned in sequence as they are encountered. Only objects in the current container will be returned, not objects in any base containers.

```
CMProperty CMGetNextObjectProperty(CMObject theObject,
                    CMProperty currProperty);
```

A refnum for the next property defined for this object is returned. `currProperty` is generally a refNum previously returned from this routine. Successive calls to this routine will thus yield all the properties for the given object.

This routine returns the refNum for the next property defined for the given object. If there are no more properties defined for this object, `NULL` is returned. If `currProperty` is `NULL`, the refNum for the first property for the object is returned.

```
CMObject CMGetNextObjectWithProperty(CMContainer targetContainer,
                    CMObject currObject,
                    CMProperty property)
```

This routine returns the refNum for the next object in the container that has the given property. `currObject` is generally a refNum previously returned from this routine. Successive calls to this routine will thus yield all the objects with the given property.

If `currObject` is `NULL`, the search starts with the first object in the container. If there is no next object with the given property, `NULL` is returned.

```
CMContainer CMGetObjectContainer(CMObject theObject);
```

The container of the designated object is returned.

```
CMGlobalName CMGetGlobalName(CMObject theObject);
```

The name of the designated object is returned. This operation is typically used on types and properties, but it can be applied to any object with a Globally Unique Name property. `NULL` is returned if the object does not have a Globally Unique Name.


```
CMRefCon  CMGetObjectRefCon(CMObject theObject)
```

This routine returns the user's "refCon" (reference constant) that s/he may associate with any object refNum (i.e., a `CMObject`). The refCon is a `CM_ULONG` that the user may use in any way. It is not touched by the API except to initialize it to 0 when the object is read into memory.

Note that the refCon is **not** preserved across closed containers, i.e., it is not saved in the TOC.

```
void  CMSetObjectRefCon(CMObject theObject,
                        CMRefCon refCon)
```

This routine is used to set the user's "refCon" (reference constant) to be assoicated with an object. The refCon is a `CM_ULONG` that the user may use in any way. It is not touched by the API.

Note that the refCon is **not** preserved across closed containers, i.e., it is not saved in the TOC.

```
VOID CMDeleteObject(CMObject theObject);
```

The specified object and all its properties and values are deleted. It is an error to use the object refnum after this call has been made.

A deleted object will be treated by all Bento operations as though it does not exist. For example, it will not be found by `CMGetNextObject`, etc.

Objects containing values that are currently open embedded containers cannot be deleted. Also, some objects created and used in the management of the TOC itself cannot be deleted.

```
VOID CMDeleteObjectProperty(CMObject theObject,
                            CMProperty theProperty);
```

The designated object property is deleted along with all of its values.. It is an error to use the refnum of any value of this property of this object after this call has been made.

A deleted object property will be treated by all Bento operations as though it does not exist. For example, it will not be found by `CMGetNextObjectProperty`, etc.

Object properties containing values that are currently open embedded containers cannot be deleted. Also, some object properties created and used in the management of the TOC itself cannot be deleted.

```
VOID CMReleaseObject(CMObject theObject);
```

The association between the object refnum and the designated object is destroyed. After this call the refnum is invalid and may be returned from one of the object calls to designate another object.

This call is also used to destroy the association between properties and types and their associated refnums.

Value Operations

All the I/O calls in the current API do I/O to or from a buffer provided by the application.

The generation number can be set, but it is not passed in the `NewValue` call. NewValue defaults the generation number to the current generation value for the container. Normally this is what is wanted.

```
CMCount CMCountValues(CMObject object,
                      CMProperty property,
                      CMType type);
```

A property for an object can be defined to have more than one value. This routine returns the number of values for the specified property belonging to the specified object.

If the type is specified as `NULL`, the total number of values for the object's property is returned. If the type is not `NULL`, 1 is returned if a value of that type is present (because there can be a maximum of one value of that type), and 0 otherwise. If the property is not defined for the object, 0 is always returned.

```
CMValue CMUseValue(CMObject object,
                      CMProperty property,
                      CMType type);
```

This routine is used to get the refNum for the value of an object's property of the given type. `NULL` is returned if the value does not exist, or if or the object does not contain the property. If the type of the value corresponds to a global type name that has an associated "use value" handler, or if its base types (if any) have associated "use value" handlers, a the refnum returned will refer to a dynamic value rather than the base value. (Normally, an application will never be aware of this difference.)

If the value is typed as an embedded container, then this refnum can be passed to `CMOpenContainer` as the attributes argument.

```
CMValue  CMGetNextValue(CMObject object,
                      CMProperty property,
                      CMValue currValue)
```

This routine returns the refNum for the next value (according to the current value order) in the objects property following `currValue`. If currValue is `NULL`, the refNum for the first value for that object's property is returned. If `currValue` is not `NULL`, the next value for that object's property is returned. `NULL` is returned if there are no more type values following `currValue` or the object does not contain the property.

`currValue` is generally a refNum previously returned from this routine. Successive calls to this routine will thus yield all the values for the specified property of the specified object as long as no other operations change the value order.


```
CMValue CMNewValue(CMObject object,
                   CMProperty property,
                   CMType type,
                   ...);
```

A new entry is created for the designated object, with the designated property and type and a refnum to the entry is returned. The generation number of the value defaults to the generation number of the container, but it may be set with `CMSetValueGeneration`.

An object's properties can have more than one value. However, the all the types for the values belonging to a given object property must be **unique**. It is an error to attempt to create a value for a property when there is already a value of the same type for that property.

If the specified type corresponds to a global type name that has an associated "use value" handler, or if its base types (if any) have associated "use value" handlers, a dynamic value will be created and returned. The value will be initialized using the `dataInitParams` arguments, which must correspond to the initialization arguments for a value of that type. See Chapter 5: Types and Dynamic Values for details.

Note that the value refnum at this point has no associated data. The value data is set with `CMWriteValueData` or `CMOpenNewContainer` (to write an embedded container). If the value will be used as an embedded container it must have the embedded container type. Using `CMWriteValueData` on a value of this type is an error.

The value is created at an unspecified location in the sequence of values for the specified property. Creating a new value may cause the order of the values for that property to change.

```
CMValue  CMVNewValue(CMObject object,
                     CMProperty property,
                     CMType type,
                     va_list dataInitParams)
```

This routine is the same as `CMNewValue()` above, except that the dynamic value data initialization (i.e., "...") parameters are given as a variable argument list as defined by the "stdarg" facility.

This routine assumes the caller sets up and terminates the variable arg list using the "stdarg.h" calls as follows:

```
#include <stdarg.h>

callersRoutine(args, ...)

{

va_list dataInitParams;

- - -

va_start(dataInitParams, args);

value = CMVNewValue(object, property, type, dataInitParams);
```



```
va_end(dataInitParams);
- - -
}
CMSize CMGetValueSize(CMValue value);
```

The size of the designated value is returned.

If the storage size of the value is different from its size as seen by the application (for example, if it is compressed) the value handlers are responsible for keeping track of the application visible size and responding correctly. See Chapter 5: Types and Dynamic Values for details.

```
CMSize CMReadValueData(CMValue value,
                       CMPtr buffer,
                       CMCount offset,
                       CMSize maxSize)
```

The data, starting at the offset, for the value is read into the buffer. The size of the data read is returned. Up to `maxSize` characters will be read (can be 0).

The data is read starting at the offset, up to the end of the data, or maxSize characters, whichever comes first. Offsets are relative to 0. If the starting offset is greater than or equal to the current data size, no data is read and 0 returned.

It is an error to attempt to read a value which has no data, i.e., a value where only a `CMNewValue` has been done.

```
void CMWriteValueData(CMValue value,
                      CMPtr buffer,
                      CMCount offset,
                      CMSize size)
```

The buffer is written to the container and defined as the data for the value. If the value already has data associated with it, the buffer overwrites the "old" data starting at the offset character position. `size` bytes are written.

If the current size of the value data is T (it will be 0 for a new value created by `CMNewValue`), then the offset may be any value from 0 to T+1. That is, existing data may be overwritten or the value extended with new data. The value of T can be gotten using `CMGetValueSize`. Note that no "holes" can be created. It is an error to use an offset greater than T+1.

Once data has been written to the container, it may be read using `CMReadValueData`. `CMWriteValueData` may only be used on an updateable container.

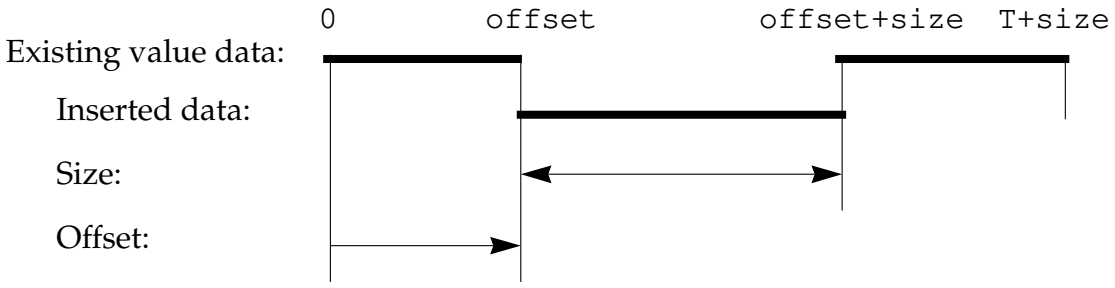
`CMWriteValueData` calls for a particular value do not have to be contiguous. Writes for other values can be done in between writes to a given value. The library takes care of generating separate value segments. The data is physically not contiguous in the container in this case. `CMWriteValueData` and `CMReadValueData` hide this by allowing the user to view the data as contiguous. The input offset is mapped to the proper starting segment and to the offset within that segment.

`CMWriteValueData()` may only be used for a container opend for writing (or converting) using `CMOpenNewContainer()`. It is an error to write to protected values, which are created implicitly by the API. This includes the predefined TOC objects (seed and offset values) and objects representing currently opened embedded containers.

For creating embedded containers, `CMOPenNewContainer` is used instead of `CMWriteValueData`. See `CMNewValue` and `CMOpenContainer` for further details.

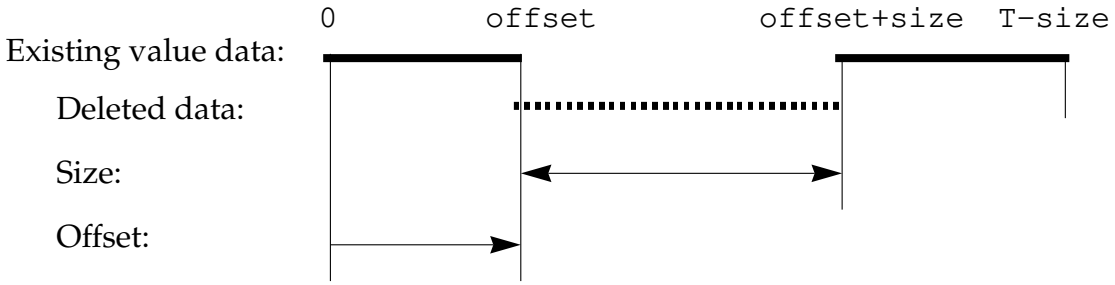
```
VOID CMInsertValueData(CMValue value,
                       CMPtr buffer,
                       CMCount offset,
                       CMSize size)
```

If the current size of the value data is T, `offset` must be ≤ T+1. The existing data in the value is "pushed aside" and the buffer is written in the space created.




```
VOID CMDeleteValueData(CMValue value,
                       CMCount offset,
                       CMSize size)
```

Let T be the length of the value data. The bytes from `offset` to `offset + size` are deleted from the value, and the value is "closed up". After this operation, the size of the value data is T-`size` (assuming `offset+size` is ≤ T). If `offset` is greater than T, no data is deleted. If `offset+size` is greater than T, all the data from `offset` to T is deleted. Neither case produces an error.




```
VOID CMDefineValueData(CMValue value,
                       CMSize offset,
                       CMSize size);
```

Existing data in the container, which must have been in the container when it was opened by Bento, is defined as the data for the value. No data is written to the container. The container must have been opened using `CMOpenNewContainer`, with the flag `kCMConverting` set in the `useMode`.

The designated value is set to reference the indicated data. The offset given is the offset from the beginning of the container. It is an error to give an offset or a size that would result in the value containing bytes outside of the data that was in the container when it was opened. The offset therefore, must be in the range of 0 to N-1, where N is the size of preexisting data at the time the container was opened.

Additional calls to CMDefineValueData() for the **same** value will define additional. i.e., continued, segments when the offset produces noncontiguous data definition. If the size of the last (most recent) value segment is T, and the offset for that segment is such that offset+T equals the offset for the additional segment, then the last segment is simply extended. This follows the same rules as `CMWriteValueData()`.

```
void  CMMoveValue(CMValue value,
                  CMObject object,
                  CMProperty property)
```

Moves the specified value from its original object property to the specified object property. The value is physically deleted from its original object/property as if a `CMDeleteValue()` were done on it. If the value deleted is the only one for the property, the property itself is deleted as in `CMDeleteObjectProperty()`.

The value is added to the "to"s object propery in the same manner as a `CMNewValue()`. The order of the values for both the value's original object property and for the value's new object property may be changed.

Note, that although the effect of a move is logically a combination `CMDeleteValue()` and `CMNewValue()`, the refnum of the value remains valid. Its association is now with the new object property.

This operation may be done at any time. No data need be assoicated with the value at the time of the move. Only moves **within the same container** are allowed.

```
VOID CMGetValueInfo(CMValue value,
                    CMContainer *container,
                    CMObject *object,
                    CMProperty *property,
                    CMGeneration *generation);
```

The container, object, property, and generation of the designated entry are returned. `NULL` may be passed for any argument except the first.


```
void  CMSetValueType(CMValue value,
                     CMType type)
```

The type of the value is set as specified.

```
void  CMSetValueGeneration(CMValue value,
                     CMGeneration generation)
```

The generation for the specified value is set. The generation number must be greater than or equal to 1. Normally this routine doesn't need to be used since the value inherits its generation from its container.

```
void CMDeleteValue(CMValue value);
```

The designated value is deleted from its object property. A deleted value will be treated by all Bento operations as though it does not exist. For example, it will not be found by `CMUseValue`, counted by `CMCountValues`, etc. .

If the value deleted is the only one for the property, the property itself is deleted as in `CMDeleteObjectProperty`. If that property is the only one for the object, the object is also deleted as in `CMDeleteObject`. Some values are protected from deletion. Protected values include the predefined TOC object values (seed and offset) and any currently open embedded container values.

```
void CMReleaseValue(CMValue value);
```

The association between the `Value` refnum and the entry is destroyed. After this call the refnum is invalid, and may be returned from a subsequent `CMUseValue` or `CMNewValue` call to designate another value.

Reference Operations

```
CMReference   *CMNewReference(CMValue value,
                     CMObject referencedObject,
                     CMReference theReferenceData)
```

This is the only way to get a persistent reference to an object that can be saved in a value, and then read from the value and used to refer to that object when the container is opened in another environment. `CMNewReference` does some bookkeeping behind the scenes and returns a token (`theReferenceData`) that will refer to `referencedObject`, but this reference will only be valid in the context of `value`!

The caller should write the `theReferenceData` to the value. It can be embedded in any way in the value: encrypted, compressed, whatever you want. The size of `theReferenceData` is determined by the size of the `CMReference` type.

```
CMObject  CMGetReferencedObject(CMValue value,
                     CMReference theReferenceData)
```

Provides the object refNum corresponding to `theReferenceData`.

value must be the value that contained theReferenceData. Values from many containers may be present at the same time, and the caller may not be aware of what container a given reference is from, especially in the presence of I/O redirection. Furthermore, the reference may have been "fixed up" using the other references routines below. Such fixups only apply to a hidden reference table associated with the value, so the value must be used as the context for converting the persistent reference.

```
CMReference  *CMSetReference(CMValue value,
                    CMObject referencedObject,
                    CMReference theReferenceData);
```

This call is similar to CMNewReference() except that here the caller defines the CMReference key to associate with an object. The specified key must be a nonzero value. The (input) pointer to theReferenceData key is returned.

In all cases the specified CMReference key is associated with the specified referencedObject. These associations are maintained in a persistent table attached to the value. If theReferenceData key is new, a new reference is recorded. If theReferenceData key matches one of the previously recorded keys in the table the reference associated with that key is **changed** to associate it with the (new) referencedObject.

This call can be used to "fix up" existing references if a value is copied as part of a structure or moved to a new environment. It can also be used to associate object references with pre-existing keys in the value data.

The only difference between CMNewReference() and CMSetReference() is that with CMNewReference(), the Container Manager defines the CMReference key, while with CMSetReference() the caller can define the key. The net result is the same; the keys are recorded in a persistent form attached to the value.

Note that multiple references to the same object can be recorded by passing different keys as theReferenceData.

Once these associations are recorded, they may be counted, deleted, and accessed using CMCountReferences(), CMDeleteReference(), and CMGetNextReference() respectively.

```
void CMDeleteReference(CMValue value,
                    CMReference theReferenceData);
```

This call deletes a single object reference created by CMNewReference() or CMSetReference() associated with the theReferenceData key in the reference table attached to the value.

The value's reference table is searched for the specified theReferenceData key. If it is found, the association is removed. If it is not found this routine does nothing. It is not an error if the theReferenceData key is not found.

```
CMCount CMCountReferences(CMValue value);
```

Returns the total number of references in the reference table attached to the value. These references were recorded by CMNewReference() or CMSetReference().


```
CMReference *CMGetNextReference(CMValue value,
                    CMReference currReferenceData);
```

This routine returns the next reference key following the `currReferenceData` key in the reference table for the specified value.

If `currReferenceData` is 0, then the first object reference key is returned. If `currReferenceData` is not 0, the next reference key after `currReferenceData` is returned. The next reference key is stored into `currReferenceData` and the pointer to `currReferenceData` is returned as the function result. `NULL` is returned and `currReferenceData` is undefined if there are no references following the key passed in as `currReferenceData`.

