Appendix B: Handler Interfaces

This appendix documents the handler mechanism, and the prototypes of a number of important handler operations.

The Bento sources come with a complete and well-documented set of example handlers that run in most environments, since they depend only on the ANSI C libraries. Many of the points in this appendix will be easier to understand if you read it in conjunction with the code.

Meta-Handler Interface

The handlers registered with Bento are actually metahandlers, because they are not called directly to carry out the operations. Instead, they are called to get procedure pointers to specific handlers that can carry out the desired operation. Typically, these procedure pointers will be cached and then used in the normal manner.

Each metahandler may provide handlers for any number of operations.

Each metahandler implements only one operation, with the following prototype:

```
CMProcPtr CMMetaHandler(CMType targetType,
                        const CMGlobalName operationType);
```

This is the required prototype of any metahandler registered by the application using `CMSetHandler`. When a specific operation is required, the `meta`handler is called, and it must return a `CMProcPtr` for the operation, or return `NULL` to indicate that the operation is not available. Once retrieved, the `CMProcPtr`s may be cached indefinitely.

`targetType` is the refnum of the type to which the operation will be applied, and `operationType` is the name of the desired operation. `targetType` is required because in some cases the operation may be applied to values whose type has no global unique name.

This approach provides more flexibility than simply passing a vector of procPtrs, and allows each operation to have its own prototype for static type checking, which would be impossible if operations were indicated by passing a selector.

Thre are three varieties of metahandlers: session, container, and value.[1] Each of these varieties is expected to provide certain operations, as documented below.

Session Handler Operations

There are three handlers that must be provided by the metahandler passed in to `CMStartSession()`. They should have the following prototypes

1. Note that this is not a fundamental distinction. In principle, a single metahandler could function in two or even all three of these roles. However, as a practical matter, this would not be a good way to structure things.


```
void error_Handler(CMErrorNbr errorNumber, ...)
```

The error reporting handler is a required special routine whose address is asked for by `CMStartSession()`. All errors are reported through here. Using the API routine `CMError()`, calls can be made outside the Container Manager as long as the session is defined (`CMStartSession()`). The intent is that handlers will call `CMError()` to report their errors just like the Container Manager does internally.

The API generally assumes the error handler will **never** return. It tries to protect itself in case you do, but don't count on it! Assume your container is screwed if this handler is called.

The Container Manager API makes available some of the same routines used internally. Specifically, the ability to take an string that can contain inserts and "edit in" those inserts (`CM[V]AddMsgInserts()`). See the utility routines documented at the end of this appendix.

```
void * Alloc_Handler(CMSize size);
```

The Container Manager API requires some form of memory management that can allocate memory and return pointers to it. By generalizing this as a handler you are free to choose a memory management mechanism appropriate to your environment.

If you are running in a standard C runtime environment, mapping this handler directly onto the C runtime `malloc()` may prove sufficient.

```
void Free_Handler(CMPtr ptr);
```

The Container Manager API calls this handler when it wants to free up memory it no longer needs. The memory attempting to be freed will, of course, be memory previously allocated by the memory allocator handler.

Container Handler Operations

Bento requires container metahandlers to provide certain operations. Bento implements all of the other operations in the API using these required operations.

Operations involving updating may be difficult to understand without reading the code.

Container handlers must provide the stream operations. These have interfaces and semantics derived from the standard C library stream operations:

```
VOID CMOpenStream(CMValue value,
                  CMStreamMode mode);

VOID CMCloseStream(CMValue value);

CMSize CMReadStream(CMValue value,
                  CMPtr buffer,
                  CMSize elementSize,
                  CMCount theCount);
```

```
CMSize CMWriteStream(CMValue value,
                     CMPtr buffer,
                     CMSize elementSize,
                     CMCount theCount);

CMSize CMGetPosStream(CMValue value);

CMSize CMSetPosStream(CMValue value,
                     CMSize posOff,
                     CMSeekMode mode);

VOID CMFlushStream(CMValue value);

CMStreamStatus CMEofStream(CMValue value);
```

When the Bento library calls the stream operations to perform I/O on a container, the value it passes as the first argument to the operations is the `referenceConstant` originally provided when the container was `Used`.

```
VOID CMReadLabel(void *attributes,
                     CMMagicBytes magicBytesSequence,
                     CMContainerFlags *containerFlags,
                     CMEncoding *encoding,
                     USHORT *majorVersion,
                     USHORT *minorVersion,
                     CMSize *tocOffset,
                     CMSize *tocSize);
```

This operation finds the label of the container specified by `attributes` and returns all of the information it contains. The location of the label is container type dependent. However, all files must have a label as specified in the format definition.

```
VOID CMWriteLabel(void *attributes,
                     CMValue value,
                     CMMagicBytes magicBytesSequence,
                     CMContainerFlags containerFlags,
                     CMEncoding encoding,
                     USHORT majorVersion,
                     USHORT minorVersion,
                     CMSize TOCOffset,
                     CMSize TOCSize);
```

This operation is called after all of the necessary information has been written to the container by Bento, just before it closes the container. It must write the label for the container (if any) and do any implementation dependent container writes. When this operation completes, the state of the container should be such that if the next call on the handler is a `CMCloseStream`, the container will be well formed and usable if it is opened again in the future.

```
CMValue CMReturnParentValue(CMRefCon refCon);
```

This handler routine is used **only** for embedded containers. It is called at open time by `CMOpen[New]Container()` so that the Container Manager may determine for itself that it is opening an embedded container for a value and what that value is. It is the parent `CMValue` for this handler that is retured by this function.

This is a **required** handler routine for embedded containers only. Therefore, container metahandlers will typically return `NULL` when asked for this handler.

```
static CM_UCHAR *returnContainerName_Handler(CMRefCon refCon)
```

When the Container Manager reports errors it passes appropriate string inserts that may be formatte into the error message. One of those inserts is usually an string that identifies for which container the error applies. The handler defined here is used by the Container Manager to get that identification.

For files, the most appropriate identification is typically the pathname for the container file.

This is an **optional** routine for reading and writing. If not provided, the type name passed to `CMOpen[New]Container()`, i.e., the metahandler type, is used for the identification.

```
CMType CMReturnTargetType(CMRefCon refCon,
                          CMContainer container);
```

If `CMOpenNewContainer()` is called with useFlags set to `kCMUpdateTarget`, then the intent is to open a new container which will record updating operations of updates applied to **another** distinct (target) container. The target container is accessed indirectly through a dynamic value whose type is gotten from a this handler. This handler must have be supplied and it must return a type which will spawn a dynamic value when `kCMUpdateTarget` is passed to `CMOpenNewContainer()`.

The process of creating a dynamic value (by the Container Manager using the returned type) will generate a "real" value in the parent container (the new container to record the updates). That value can be used by future `CMOpenContainer()`'s to "get at" the target again. To be able to find it, the created value becomes a special property of TOC #1. `CMOpenContainer()` will look for that property.

This handler is running in the context of the parent container, which is passed as a parameter along with the usual refCon. The handler registers the type in this container.

See the description of dynamic values and their base types for further details on how these types spawn dynamic values.

This is a **optional** routine for reading and writing. It is **required** for updating when `CMOpenNewContainer()` is called with useFlags set to `kCMUpdateTarget`.

```
void FormatData(CMRefCon refCon,
                    CMDataBuffer buffer,
                    CMSize size,
                    CMPrivateData data);
```

This handler is used to format "internal" Container Manager data to be written to the container (e.g., updating data). 1, 2, or 4 bytes (size 8-bit byte) "chunks" of data are expected to be copied from the data to a buffer. Pointers to the data and the buffer are passed in. The buffer can always be assumed large enough to hold the data. The pointer to the data can be assumed to point to a `CM_UCHAR` if size is 1, `CM_USHORT` if size is 2, and `CM_ULONG` is size if 4.

The 1, 2, or 4 bytes are, of course, stored in the `CM_UCHAR`, `CM_USHORT`, or `CM_ULONG` as a function of the architecture. These may be a different size than what is expected to be written to the container. Indeed, it is the potential difference between the architecture from the data layout in the container that this handler must be provided.

The information is stored in the container in a layout **private** to the Container Manager. For example, it is used to format the fields of the TOC. The library does repeated calls to this handler to format the information it needs into a buffer that is eventually written using the write handler.

In this example `CM_UCHAR`, `CM_USHORT` and `CM_ULONG` directly map into the container format 1, 2, and 4 byte entities. Hence the formatting is straight-forward.

This is an **optional** routine for reading and writing. It is **required** for writing or updating when `CMOpenNewContainer()` is called with useFlags set to `kCMUpdateTarget` or `kCMUpdateByAppend`.

```
void CMExtractData(CMRefCon refCon,
                    CMDataBuffer buffer,
                    CMSize size,
                    CMPrivateData data);
```

This handler is used to extract "internal" Container Manager data previously written to the container (e.g., updating data). 1, 2, or 4 bytes (size 8-bit byte) "chunks" of data are expected to be copied from a buffer to the data. Pointers to the data and the buffer are passed in. The buffer can always be assumed large enough to supply all the requested data. The pointer to the data can be assumed to point to a `CM_UCHAR` if size is 1, `CM_USHORT` if size is 2, and `CM_ULONG` is size if 4.

The 1, 2, or 4 bytes are, of course, formatted within the `CM_UCHAR`, `CM_USHORT`, or `CM_ULONG` as a function of the architecture. These may be a different size than what is expected to be written to the container. Indeed, it is the potential difference between the architecture from the data layout in the container that this handler must be provided.

This routine is used to store information in the container in a layout **private** to the Container Manager. For example, it is used to extract the fields of the TOC. The Container Manager does repeated calls to this handler to extract the information it needs from a buffer that it loads using the read handler.

This is a **optional** routine for writing. It is **required** for reading, or if an updating container is opened (i.e., a container used previously for updating). It is **required** for updating when `CMOpenNewContainer()` is called with useFlags set to `kCMUpdateTarget` or `kCMUpdateByAppend`.

Value Handlers

These are discussed in the chapter on Types and Dynamic Values.

Handler Support Routines

When writing handlers, certain facilities used by the API are needed to provide a consistent interface to the application.

Session Handler Pass-Throughs

The handlers need to use the session facilities for memory management and error reporting, since they are effectively running as part of the library. These routines provide access to those facilities.

```
void  CMMalloc(CMSize size, CMSession sessionData)
```

This routine provides a access path for the handler writer to use the same memory management allocator defined for the current Container Manager session. size bytes are allocated as defined by that handler.

The session memory allocator handler is defined by the metahandler passed to `CMStartSession()`. The `sessionData` is the current session refNum returned from `CMStartSession()`.

```
void  CMFree(CMPtr ptr,
                    CMSession sessionData)
```

This routine provides a access path for the handler writer to use the same memory management deallocator defined for the current Container Manager session. A pointer (ptr) which must have been allocated by `CMMalloc()` is passed to release the memory in the manner defined by the handler.

The session memory deallocator handler is defined by the metahandler passed to `CMStartSession()`. The `sessionData` is the current session refNum returned from `CMStartSession()`.

```
void CMError(CMSession sessionData, CMErrorString message, ...);
```

This routine provides an access path for a handler writer to use the same error reporter defined for the current Container Manager session. The session error reporting handler is defined by the metahandler passed to `CMStartSession()`. The `sessionData` is the current session refNum returned from `CMStartSession()`.

Note, as currently defined, the error reporting handler should not return to its caller. But if it does, `CMError()` will also.

Error Reporting Support

```
char *CMAddMsgInserts(char *msgString,
                      CMSize maxLength,
                      ...);
```

This routine takes the (error) message string in `msgString` and replaces all substrings of the form "^n" with the inserts from the variable arg list of insert strings. The function returns the edited `msgString` as its result.

The string buffer must be passed in `msgString`. Its max length is also specified but this must be a value less than or equal 1024 (not including delimiting null). The message will be truncated to fit in the buffer. The string pointer is returned as the function result.

The substring "`^0`" is replaced with the first insert. "`^1`" is replaced with the second insert, and so on. It is assumed that there are enough insert strings to cover all the "`^n`"s called for (not unlike `printf()`).

Note, the "`^n`"s in the message string do not have to be in ascending order. Thus "`^1`" could occur before "`^0`" in the `msgString`.

```
char *CMVAddMsgInserts(char *msgString,
                      CMSize maxLength,
                      va_list inserts);
```

This routine is the same as `CMAddMsgInserts()` above, except that the extra (inserts) arguments are given as a variable argument list as defined by the "stdarg" facility.

```
CMErrorString CMGetErrorString(CMErrorString errorString,
                      CMSize maxLength,
                      CMErrorNbr errorNumber,
                      ...)
```

This routine takes a defined Container Manager (error) message number and its corresponding insert strings and returns a (english) string corresponding to the message number with the inserts filled into their proper positions. It is assumed the error number and inserts were the ones reported to the error handler.

The string buffer must be passed in errorString. Its max length is also specified but this must be a value less than or equal 1024 (not including delimiting null). The message will be truncated to fit in the buffer. The string pointer is returned as the function result.

This routine is provided as a convenience to easily convert the error codes and their corresponding inserts to a printable string.

```
CMErrorString CMVGetErrorString(CMErrorString errorString,
                      CMSize maxLength,
                      CMErrorNbr errorNumber,
                      va_list inserts)
```

This routine is the same as `CMGetErrorString`() above, except that the extra (inserts) arguments are given as a variable argument list as defined by the "stdarg" facility.


```
char *CMReturnContainerName(CMContainer container);
```

Generally the errors reported are provided with at least one insert that identifies which container we're talking about. The wording of the messages defined for the Container Manager assume this identification insert. The identification takes the form of the container "name" which is obtained from a handler routine provided for that purpose.

This routine is provided to test if the container metahandler provides a name handler, and call the handler if it exists. If it doesn't exist the container type name is returned.

Note, this routine is available to handler writers so that they can generalize their error reporting/message routines and word their messages with the container identification.

Value Handler Support

```
CMCount  CMScanDataPacket(CMType type,
                          CMMetaData metaData,
                          CMDataPacket dataPacket, ...)
```

This routine is used by a dynamic value's "new value" handler to extract the fields of a data packet passed to it by the Container Manager. The data packet represents all the `CMNewValue()` "..." parameters for the type also passed to the "new value hander".

Only that portion of the `CMNewValue()` "..." parameters associated with the type are passed in the data packet. The Container Manager determines the parameters by the placement of the type within its heirarchy (types may have base types) and the metadata.

The Container Manager accesses the metadata through a "metadata" handler for the type to build the data packet. `CMScanDataPacket()` inverts the the operation and allows the "new value" handler to extract the data back into distinct variables. The "new value" handler can use its own "metadata" handler to pass to the `CMScanDataPacket()` routine to extract the data. Each `CMScanDataPacket()` "..." parameter must be a pointer; extracted data read from the data packet are stored into the locations pointed to by the pointers.

The function returns the number of data items extracted and assigned to the parameters. This could be 0 if metadata is passed as `NULL`, or if an error is reported and the error reporter returns.

```
CMValue  CMGetBaseValue(CMValue value)
```

Returns the base value for a dynamic value and `NULL` if the value is not a dynamic value. It is expected that this routine will only be called from dynamic value handlers. Indeed, this is enforced!

