IBM Rational Developer for System z
8.5

Common Access Repository Manager Developer's Guide

IBM

IBM Rational Developer for System z
8.5

*Common Access Repository Manager
Developer's Guide*

IBM

Contents

About this book

This book explains how to develop repository access managers (RAMs) and Common Access Repository Manager (CARMA) clients. It includes the following topics:

- How to develop a RAM capable of connecting to a software configuration manager (SCM)
- How to develop a CARMA client capable of accessing various SCMs through CARMA using RAMs

You can use this document as a guide to these tasks or as a programming reference.

Who should read this book

This book is intended for application programmers or anyone who wants to learn how RAMs and clients are developed.

To use this book as a guide for RAM development, you need to be familiar with the SCM for which you are developing a RAM. To use this book for CARMA client development, you need to understand generic SCM concepts.

Conventions used in this book

Throughout this book there are several references to data sets and members that have the high-level qualifier FEK. Depending on how your CARMA host has been configured, these data sets may actually have different file names. For example, the sample library referred to as FEK.SFEKSAMP in this book could actually be named MYCORP.TEST.SFEKSAMP on your host system. Thus, depending on the configuration of your host system, the FEK in the data set names referenced in this book may be replaced with some other string. Contact your system programmer to determine where these data sets are actually located on your host system.

Chapter 1. Introduction to CARMA

CARMA is a library that provides a generic interface to z/OS® software configuration managers (SCMs). Developers can build on top of CARMA by developing repository access managers (RAMs) that plug into the CARMA environment. RAMs define how CARMA should communicate with various SCMs. For example, a CARMA host (a z/OS host machine with CARMA on it) could be configured to use one RAM to communicate with IBM® Source Code Library Manager (SCLM) repositories and another RAM to communicate with your own custom SCM.

By using CARMA, developers of client software can avoid writing specialized code for accessing SCMs, and easily allow support for any SCM for which a RAM is available. CARMA is a DLL stored within an MVS™ PDS. Only z/OS clients can directly access CARMA. In order to access CARMA from a workstation, a software bridge between the workstation and host must be developed. This bridge software must act as a client to the CARMA host and as a server to workstations. IBM Rational Developer for System z ships with such a software bridge to allow the CARMA plug-in to access CARMA hosts.

Figure 1 illustrates an example CARMA environment.
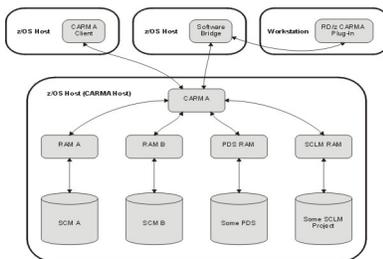
Figure 1. Example CARMA environment

CARMA currently ships with four sample RAMs:
- Sample TSO/ISPF PDS RAM - Provides access to the partition data sets (PDS) through the use of Library Management API of TSO.
- Sample SCLM RAM - Provides access to Software Configuration Library Manager (SCLM) projects.
- Sample COBOL RAM - Provides example COBOL code which demonstrates handling of ILC issues specific to COBOL-based RAM development.
- Skeleton RAM - Provides a starting point for RAM developers.

Note: The sample RAMs are provided for the purpose of testing the configuration of your CARMA environment and as examples for developing your own RAMs. **Do not use the provided sample RAMs in a production environment.**

To access your own SCMs using CARMA, you will need to obtain or develop additional RAMs. See Chapter 2, "General concepts," on page 5 and Chapter 3, "Developing a RAM," on page 11 for more information on developing a RAM to access your own SCM.

Supported operations

CARMA currently supports the following sets of generic actions:
- Browse an SCM
- Extract an SCM member
- Create an SCM member
- Update an SCM member
- Get SCM member metadata
- Update SCM member metadata
- Copy a member to a PDS or SDS
- Copy a member from a PDS or SDS
- Delete a member or container
- Lock, unlock, check in, and check out a member
- Browse an SCM member's history

Although CARMA supports all of these actions, it is quite possible that a given SCM may not support one or more of these actions due to its design. Developers of RAMs accessing such SCMs should follow the guidelines for handling unsupported operations in "Dealing with unsupported operations" on page 18.

CARMA also provides a framework called the Custom Action Framework (CAF) for customizing the actions a RAM can perform (see Chapter 4, "Customizing a RAM API using the CAF," on page 55 for more information).

Locating the sample files

Sample files have been included in the CARMA host installation packages. After your CARMA host has been successfully set up, you should be able to find these sample files as members within the sample library (FEK.SFEKSAMP). The following table summarizes these members:

Table 1. Sample CARMA development files

Member in FEK.SFEKSAMP	Description
CRA390H	Header needed for clients
CRA390SD	CARMA/390 DLL side deck
CRA#CCLT	JCL to compile a CARMA client to a PDS/E
CRA#PCLT	JCL to compile a CARMA client to a PDS
CRA#XCLT	JCL to run a host-based client
CRACLISA	Sample client source code
CRADSDEF	C header needed for clients and RAMs
CRAFCDEF	C header needed for RAMs
CRASUTIL	Source code for the RAM utility functions
CRAHUTIL	Header needed for RAM utility functions
CRA$VMSG	IDCAMS JCL to REPRO CRAMSG
CRAMSGH	Header file common to the sample PDS and SCLM RAMs
CRAMSGO	Object module common to the sample PDS and SCLM RAMs

Table 1. Sample CARMA development files (continued)

Member in `FEK.SFEKSAMP`	Description
CRA#CCOB	JCL to compile the COBOL Sample RAM to a PDS/E
CRA#PCOB	JCL to compile the COBOL Sample RAM to a PDS
CRA#CRAM	JCL to compile the Skeleton RAM to a PDS/E
CRA#PRAM	JCL to compile the Skeleton RAM to a PDS
CRA#CSLM	JCL to compile the sample SCLM RAM to a PDS/E
CRA#PSLM	JCL to compile the sample SCLM RAM to a PDS
CRA#CPDS	PDS RAM to a PDS/E
CRA#PPDS	PDS RAM to a PDS
CRARAMSA	Skeleton RAM source code
CRA$VDEF	JCL to REPRO CRADEF
CRA#VPDS	JCL to REPRO the sample PDS RAM's messages
CRA#VSLM	JCL to REPRO the sample SCLM RAM's messages
CRASPDS	Source code for the sample PDS RAM
CRA$VSTR	JCL to REPRO CRASTRS
CRASSCLM	Source code for the sample SCLM RAM

Note: The CRA$*members have been copied to FEK.#CUST.JCL for customization during the setup of Developer for System z®. Ask your system programmer for a copy of these customized JCLs to use as a starting point for your own

Chapter 2. General concepts

This section outlines several general concepts that are essential to understanding how CARMA works. For a more in-depth overview of these concepts, please read *Integrating Source Code Management Systems into WebSphere® Developer for zSeries® CARMA (SC23-5817-00)* located at the IBM Rational Developer for System z library (http://www.ibm.com/software/awdtools/devzseries/library/).

Browsing

CARMA views all entities within an SCM as **repository instances** (or RIs), members, and metadata. Repository instances are the entities at the highest level within an SCM. For example, the sample PDS RAM uses PDSs as RIs. RIs could be different libraries of code, different levels of code, or whatever the RAM developer thinks would make the most sense for CARMA clients. For most SCMs, a RI should represent a project or component in the SCM. Repository Instances are more generally referred to as instances during further discussion.

Members are entities contained within instances or other members. Members that contain other members are known as **containers**, while members that do not contain other members are known as simple members.

Figure 2 illustrates a simple hierarchy. "Build" and "Development" are repository instances, the components are containers, and the source files are simple members.
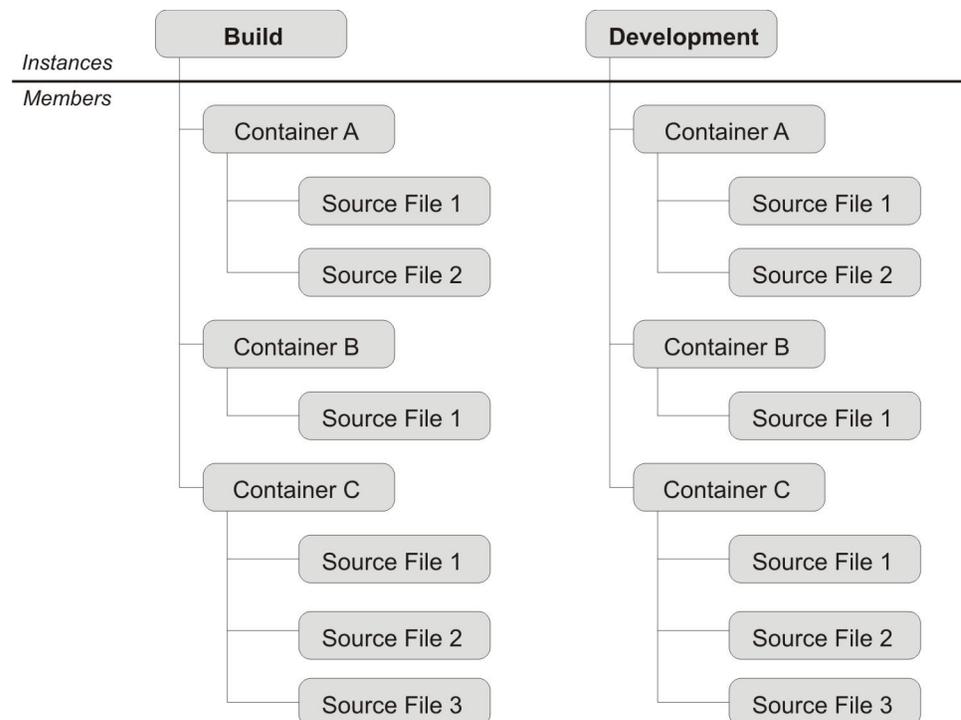
Figure 2. Example SCM hierarchy

Checking in and out

CARMA provides a generic interface across various SCMs, each of which may handle operations differently. Since it is not possible to predict whether the check in or check out operation for any given SCM will respectively expect or return a member's contents, CARMA has been designed such that the check in and check out actions are flag-setting operations. That is, no member contents are passed to or returned from the SCM as part of the check in and check out actions.

Certain SCMs might expect the contents of a member to be passed in during a check in operation for that member. A RAM for such an SCM should handle this case by storing the member contents in a temporary location before making the check in call to the SCM.

Similarly, certain SCMs might return the contents of a member during a check out operation for that member. A RAM for such an SCM should handle this case by storing the member contents in a temporary location until the client retrieves the contents.

Memory allocation

Many of the CARMA API functions require that either the RAM or the CARMA client allocate memory to store function results or parameters that are passed between the RAM and the CARMA client. For all functions other than `extractMember` and `putMember`, a one dimensional array will need to be allocated by the RAM and freed by the client to store sets of instance information, member information, and other information. The following diagram illustrates how the RAM should allocate this array:
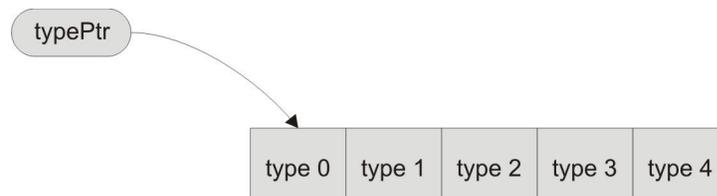
Figure 3. Simple one dimensional array as would be allocated by a RAM

Each element in the array depicted above is of data structure type `type`. `typePtr` is a `type` pointer (of type `type*`) that serves as a handle to the newly allocated memory. In C, this memory can be allocated with the following code:

```
typePtr = (type*) malloc(sizeof(type) * numElements);
```

where `numElements` is the number of array indices that need to be created. The memory `typePtr` points to must be freed by the client once it is no longer needed.

The `putMember` and `extractMember` functions use two-dimensional arrays to transfer member contents, with each array row containing one of the member's records. For `extractMember`, the RAM should allocate the array and the CARMA client should free the array. For `putMember`, the CARMA client should both allocate and free the array. In both cases, the array should be allocated as illustrated in the following diagram:

charPtrPtr

charPtr 0 | charPtr 1 | charPtr 2

char 0a | char 0b | char 1a | char 1b | char 2a | char 2b
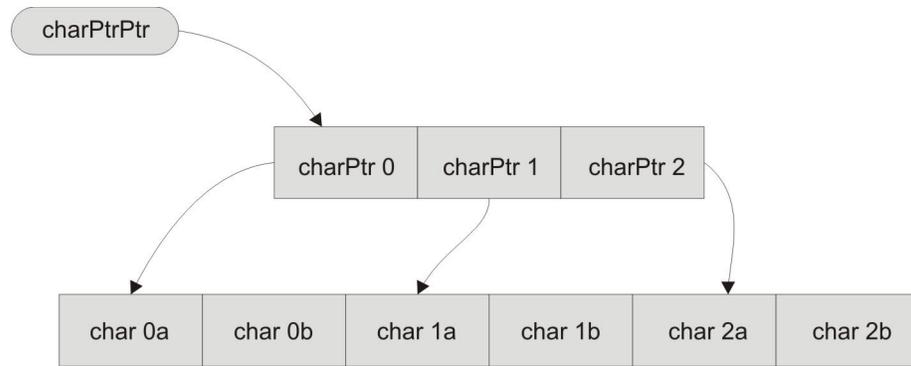
Figure 4. Two-dimensional character array as used in `extractMember` and `putMember`

`charPtrPtr` is a pointer to a `char` pointer (it is of type `char**`) that serves as a handle to an array of `char` pointers (elements of type `char*`). The data for the two-dimensional character array is actually stored in a one-dimensional character array; the idea of rows and columns is purely conceptual. The array of `char` pointers is used to provide handles to the first element in each row of the "two-dimensional" array. Thus, in the illustration, the first row of the two-dimensional array consists of elements 0a and 0b, with 0a being the first element of that row; the second row consists of elements 1a and 1b, with 1a being the first element of that row; and so on.

To allocate a two-dimensional array such as the ones required for the `extractMember` and `putMember` functions, the CARMA client must first create `charPtrPtr`. In C, use the following declaration:

```
char** charPtrPtr;
```

If the CARMA client is allocating the two-dimensional character array (as is the case for the `putMember` function) the array can now be allocated. In C, the CARMA client should use the following code:

```
charPtrPtr = (char**) malloc(sizeof(char*) * numRows);
*charPtrPtr = (char*) malloc(sizeof(char) * numColumns * numRows);
for(i = 0; i < numRows; i++)
   (charPtrPtr)[i] = ( (*charPtrPtr) + (i * numColumns) );
```

where `numRows` is the number of rows and `numColumns` is the number of columns in the two-dimensional array. The first line allocates the array of `char` pointers (one pointer for each row in the two-dimensional array), the second line allocates the array that holds the data for the two-dimensional array, and the `for` loop assigns each of the `char` pointers in the `char` pointer array to a row in the two-dimensional array.

If the RAM is allocating the two-dimensional character array (as is the case for the `extractMember` function) an extra step is required before the array can be allocated: `charPtrPtr` needs to be passed by reference to the RAM as `extractMember`'s `contents` parameter; that is, a pointer to `charPtrPtr` needs to be passed. This is necessary so that the client has a handle to the two-dimensional array after the RAM has allocated the array. Suppose that the RAM receives a parameter named `contents` of type `char***` in the RAM function that will allocate the two-dimensional array. The RAM should then allocate the two-dimensional array, using `contents` as a handle to the array. In C, the RAM should use the following code to allocate the two-dimensional array:

```
*contents = (char**) malloc(sizeof(char*) * numRows);
**contents = (char*) malloc(sizeof(char) * numColumns * numRows);
for(i = 0; i < numRows; i++)
   (*contents)[i] = ( (**contents) + (i * numColumns) );
```

where `numRows` is the number of rows and `numColumns` is the number of columns in the two-dimensional array. The first line allocates the array of `char` pointers (one pointer for each row in the two-dimensional array), the second line allocates the array that holds the data for the two-dimensional array, and the `for` loop assigns each of the `char` pointers in the `char` pointer array to a row in the two-dimensional array.

Regardless of who allocated the array, the CARMA client must free the two-dimensional character array in both the `extractMember` and `putMember` functions. In C, the CARMA client should use code similar to the following:

```
free(charPtrPtr[0]);
free(charPtrPtr);
```

This frees the data array before freeing the `char` pointer array, thus avoiding a memory leak.

Member contents

The contents of SCM members can be sent between the RAM, CARMA, and the client all at once or a piece at a time. It is recommended that the contents of large members be sent a piece at a time to avoid attempting to allocate a larger chunk of memory than is available.

The contents will be passed to and from the RAM as two-dimensional character arrays, each row in the array corresponding to a record in the member. As the RAM writes to or reads from a member, it should place the first member record it encounters at index 0 in the array, so that the indices of the array and member match.

CARMA also supports binary transfer of member contents. When binary transfers are performed, the contents are passed to and from the RAM as one-dimensional character arrays.

Character buffers

To match the convention for passing strings in MVS, the RAM should expect all character buffers passed to it to be padded with spaces instead of being null-terminated. The RAM should also set up any buffers being returned to the client in the same way. Assuming a buffer length of 30, the string "CARMA mechanic" would be passed in the format illustrated in Figure 5 instead of the format illustrated in Figure 6 on page 9 (where "?" represents an unknown character). Both RAM and client developers should initialize buffers that they have created to be filled with spaces.

C	A	R	M	A		m	e	c	h	a	n	i	c																

Figure 5. Example of correct RAM buffer usage

C	A	R	M	A		m	e	c	h	a	n	i	c	\0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

An improvement to CARMA in version 7.1 is the ability for the CARMA client to support null terminated character buffers (as shown in figure 6). All strings passed to the RAM will still be in the format shown in Figure 5, but the provided CARMA client will work with both space filled and null terminated character buffers. Before designing your RAM to provide null terminated character buffers, ensure it will only be used with version 7.1 or later of CARMA clients, or another appropriate client.

Figure 6. Example of incorrect *RAM buffer usage*

Return codes

All functions that run successfully should produce a return code of 0. If an error occurs, RAM developers may return a code based on Table 2 below

Table 2. Return code ranges

Error Type	Range
CARMA Errors	4 – 99
Generic RAM Errors	100 – 200
Software Bridge Errors	201 – 500
RAM Specific Errors	501 – 900
TSO Errors	901 – 999

If an error occurs, RAM developers may return a code between 100 and 200 or between 501 and 900. Codes ranging from 100 to 200 are reserved for generic errors that all RAMs may face. Codes ranging from 501 to 900 should be used for any errors that are specific to a certain RAM. Likewise, CARMA may return error codes between 4 and 99, a software bridge created between CARMA and a workstation client may return error codes between 201 and 500, and TSO errors may be flagged by returning error codes between 901 and 999. See Appendix A, "Return codes," on page 103 for a list of the predefined error codes. When an error results in a return code between 501 and 900, the RAM should fill the error buffer with the details of the error. When an error results in a return code between 100 and 200, CARMA will be able to recognize the error and will put the appropriate error message in the error buffer. If the RAM provides additional error information using its error buffer, CARMA will append this information to the error message it produces.

Logging

CARMA uses its own logging system. Trace levels can be used to filter log messages generated by CARMA and the RAM. The available trace levels are listed in the following table:

Table 3. Trace levels. Messages at the "None" trace level are not logged.

Enumeration	Trace Level
-1	None
0	Error
1	Warning
2	Information

Table 3. Trace levels. Messages at the "None" trace level are not logged. (continued)

Enumeration	Trace Level
3	Debug

All messages at or below the chosen level will be logged. For example, if the"Information" trace level is chosen, the following types of messages will be logged: information, warning, and error. Additional information on logging is discussed in "Logging" on page 18 (for RAM development) and "Logging" on page 76 (for CARMA client development).

Custom parameters and return values

Both custom parameters and return values are referenced by elements in `void` pointer arrays. Since parameters and return values can be of various data types, pointers to them are typecast to `void*` and then stored in a single array. Each such array holds either the custom parameter or the custom return values, but never both. The following diagram illustrates the structure of an example custom parameter array:
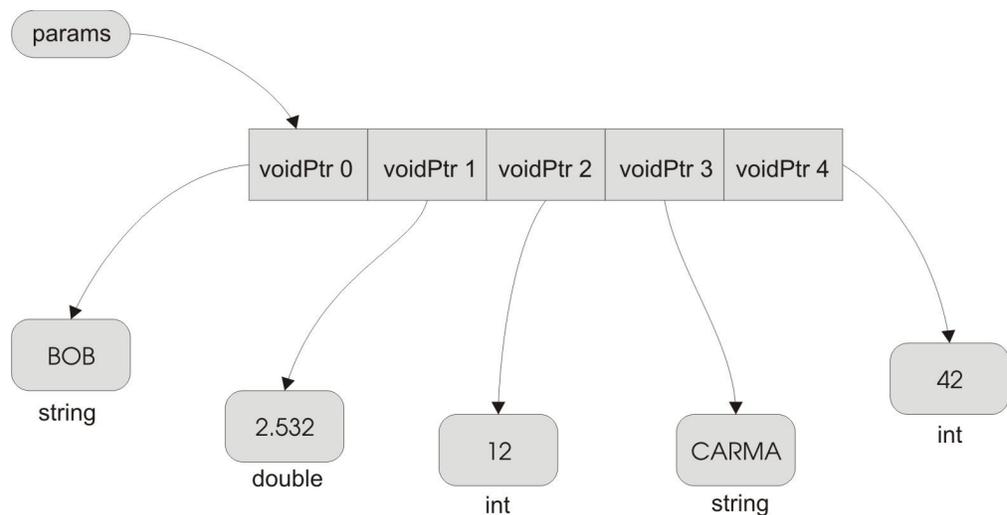
Figure 7. Custom parameter array example. Each element in the array is a pointer to a parameter. The value of each parameter is shown and labeled with its data type.

where `params` is a pointer to a `void` array and each `voidPtr` in the array is a `void` pointer that points to a parameter. Custom return value arrays should be similarly structured.

The number of elements that should be in a custom parameter or return value array is dependent upon the CAF information in the CARMA VSAM clusters (see "Creating VSAM records from a RAM model" on page 64). Since it is the responsibility of the RAM developer to include information on the custom parameters and return values in the VSAM clusters, the RAM developer should already know how many elements to include in the custom parameter and return value arrays. CARMA client developers can use the `getCAFData` or `getCAFData2` CARMA function to retrieve information on the custom actions, parameters, and return values for a RAM (see "getCAFData" on page 100 for more information). Using this information, CARMA client developers can determine how many custom parameters and return values are required for each RAM action.

Chapter 3. Developing a RAM

Repository access managers (RAMs) provide CARMA with access to specific SCMs. A RAM is a dynamically linked library (DLL) that exports entry points for all API functions that it implements. An API function reference is included at the end of this chapter.

Most RAM functions have the following pattern:

1. Determine what instance and/or member the request applies to
2. Contact the SCM to carry out the requested operation
3. Allocate any memory necessary to return the result
4. Fill the allocated memory with the result
5. Return the result to CARMA

You can use the skeleton RAM source file, CRARAMSA (located in the sample library), as a starting point for your RAM if you are developing your RAM in C. Keep in mind that your RAM must follow the state, memory allocation, and API implementation guidelines given in this document; otherwise, serious problems could develop: CARMA might not communicate properly with the RAM; memory leaks could develop; or, in the worst case, CARMA or the RAM could abnormally end. Specifically, read the following sections carefully:

- "Memory allocation" on page 6
- "State functions" on page 21

RAM Construction

RAM construction is a process that deviates from the construction process of a normal load module or program object. Because of the requirements of DLL support, the process of creating the RAM for a PDS requires more effort than that for a PDS/E.

Construction for a PDS

The process for creating a RAM in a PDS requires the usage of the pre-linker. The steps outlined in creating a RAM for a PDS are as follows:

1. Compile
2. Pre-Link
3. Link

The compile step requires that each source be given the proper compile options for producing DLL object code. The pre-link step involves feeding the object code into the pre-linker. The output of the pre-linker is object code that is valid input for the linker. The pre-linker will create a side deck that may be required input for the linker for resolving external references. The final link step requires object code and side decks that are created by the previous steps as input.

To assist in performing compilations involving C, the JCL procedure CRACPL is provided in the CARMA sample library. Sample JCL for creating a RAM in a PDS is also provided in the members CRA#CRAM and CRA#PRAM. CRA#CRAM compiles to PDS/E, and CRA#PRAM can compile to either PDS or PDS/E. Only CRA#PRAM (or other compile to PDS JCL) requires CRACPL.

Construction of a PDS/E

The process for creating a RAM in a PDS/E involves two steps. The output of this process is a program object.

1. Compile
2. Bind

The first step involves using the compiler to generate the object code for the RAM. After the object code for all sources has been created, it may be fed to the binder as input for generating the RAM program object.

The process of creating a RAM in a PDS/E is simpler than that for a PDS. Example JCL is provided for creating the PDS, SCLM, and COBOL RAMs in a PDS/E. The sample JCL makes use of standard procedures for performing the processes of compiling and binding.

Notes:

1. RAMs written in C are only intended for use with the z/OS XL C compiler
2. RAMs written in COBOL are only intended for use with the Enterprise COBOL for z/OS compiler.

Using the RAM utilities module

The RAM utility functions are provided as sample source that may be compiled for usage by any RAM designed to work with CARMA. It provides access to methods that are frequently required by RAM designers and are often called several times within a single RAM. Using the RAM utilities module and its library of functions, developers will be able to save a great deal of time and simplify performing CARMA operations on PDS members.

The following methods are included in the RAM utilities module:

utilInitMemberList

This method initializes a list of PDS members for the specified PDS. It must be called before calls to `utilGetNextMember` are made. A call to `utilCloseMemberList` must also be made before the next call to `utilInitMemberList` if `utilInitMemberList` returns 0 for success.

`int utilInitMemberList(char pds[44], int* count, void** tempDataPtr)`

char pds[44]	Input	The specified PDS to list members for
int* count	Output	The number of members in the PDS
void** tempDataPtr	Output	State information stored for use by the module, created by this call

utilGetNextMember

This method places the next member in the PDS specified by `utilInitMemberList` into member. `utilGetNextMember` returns 0 for success, 1 for no members remaining and any other value on an error. `utilCloseMemberList` should be called when finished reading the member list to prevent memory leaks. If `utilGetNextMember` returns something other than 0 or 1, you do not have to call `utilCloseMemberList`.

`int utilGetNextMember(char member[8], void** tempDataPtr)`

char member[8]	Output	The next member in the PDS (space filled if no members exist)
void** tempDataPtr	Output	State information stored for use by the module, modified by this call

utilCloseMemberList

This method cleans up the PDS member list created by utilInitMemberList. It should be called before another utilInitMemberList is called.

void utilCloseMemberList(void** tempDataPtr)

void** tempDataPtr	Input	State information stored for use by the module, cleaned up by this call

utilGetAllMemberInfo

This method returns the following ISPF-maintained metadata available for the given PDS member. This metadata includes:

- Dataset
- Version
- Modification Level
- Creation Date
- Modification Data
- Modification Time
- Current Size
- Initial Size
- Number of Records Modified

int utilGetAllMemberInfo(char pds[44], char member[8], memberInfo* output)

char pds[44]	Input	The PDS which contains the member.
char member[8]	Input	The member name.
memberInfo* output	Output	Member information is placed in this structure.

utilGetMemberInfo

This method returns a piece of ISPF-maintained metadata available for the given PDS member, including the types listed in the "utilGetAllMemberInfo" method.

int utilGetMemberInfo(char pds[44], char member[8], char* info, int ukey)

char pds[44]	Input	The PDS which contains the member.
char member[8]	Input	The member name.

char* info	Output	A buffer large enough to contain the info. `U_ISPF_MI_SIZE[ukey]` will tell the size needed for a given key. It will not be NULL terminated, but the space should be filled to the size specified in `U_ISPF_MI_SIZE`.
int ukey	Input	Key for information wanted. See RAM Utilities Module header file for a complete list of keys.

utilSetMemberInfo

This method allows all ISPF-maintained metadata to be set. The metadata that may be set includes the types listed in the "utilGetAllMemberInfo" method.

`int utilSetMemberInfo(char pds[44], char member[8], char info[10], int ukey)`

char pds[44]	Input	The PDS which contains the member.
char member[8]	Input	The member name.
char info[10]	Input	The new information. `Ukey_ZLLIB` and `Ukey_ZLMSEC` are not supported.
int ukey	Input	Key for information wanted. See RAM Utilities Module header file for a complete list of keys..

utilGetAllPDSInfo

This method returns all ISPF metadata available for the given PDS.

`int utilGetAllPDSInfo(char pds[44], pdsInfo* output)`

char pds[44]	Input	The PDS to get all information about.
pdsInfo* output	Output	PDS information will be placed in this structure.

utilCopyPDStoPDS

`int utilCopyPDStoPDS(char fromInstanceID[44], char frommemberID[8],`
`            char toInstanceID[44], char tomemberID[8])`

char fromInstanceID[44]	Input	The PDS to copy from
char frommemberID[8],	Input	The PDS member to copy
char toInstanceID[44]	Input	The PDS to copy to
char tomemberID[8]	Input	The PDS member to replace (or create if it does not exist)

utilCopyPDStoSDS

```
int utilCopyPDStoSDS(char fromInstanceID[44], char frommemberID[8],
            char toInstanceID[44])
```

char fromInstanceID[44]	Input	The PDS to copy from
char frommemberID[8],	Input	The PDS member to copy
char toInstanceID[44]	Input	The SDS to copy to (This must exist.)

utilCopySDStoPDS

```
int utilCopySDStoPDS(char fromInstanceID[44],char toInstanceID[44],
      char tomemberID[8])
```

char fromInstanceID[44]	Input	The SDS to copy from
char toInstanceID[44]	Input	The PDS to copy to
char tomemberID[8]	Input	The PDS member to replace (or create if it does not exist)

utilCopySDStoSDS

```
int utilCopySDStoSDS(char fromInstanceID[44], char toInstanceID[44])
```

char fromInstanceID[44]	Input	The SDS to copy from
char toInstanceID[44]	Input	The SDS to copy to (This must exist.)

utilPutMemberInit

Will initiate a put to a PDS member. Call `utilPutMemberRecs` or `utilPutMemberRec` until all the required records are put.

```
int utilPutMemberInit(char pds[44], char member[8], int* lrecl)
```

char pds[44]	Input	The target PDS.
char member[8]	Input	The target PDS member.
int* lrecl	Output	The lrecl, or record size for the given PDS. (For VB, this will be the max record size.)

utilPutMemberRecs

Put multiple records of a fixed length.

```
int utilPutMemberRecs(char** contents, int numRecords)
```

char** contents	Input	2-D array of records (of size lrecl) to be put.
int numRecords	Input	The number of records in a members contents

utilPutMemberRec

Put a single record of variable length.

```
int utilPutMemberRec(char* contents, int length)
```

char* contents	Input	A single record to be put.
int length	Input	The length of the record to be put (maximum of lrecl).

utilPutMemberClose

Must be called for every utilPutMemberInit, except in the case of an error condition in utilPutMemberInit, utilPutMemberRec, or utilPutMemberRecs.

```
int utilPutMemberClose()
```

utilExtractMemberInit

Setup the PDS member to extract from.

```
int utilExtractMemberInit(char pds[44], char member[8], int* lrecl
                 int* recFM, int* numRecords)
```

char pds[44]	Input	The source PDS
char member[8]	Input	The source PDS member
int* lrecl	Output	The lrecl, or record size for the given PDS. (For VB, this will be the max record size.)
int* recFM	Output	A Flag representing record format. Choices are U_RECFM_VB, U_RECFM_FB, and U_RECFM_U.
int* numRecords	Output	The number of records in the PDS member. Because this uses ISPF statistics to determine the number of records, the maximum value is 65535 and this will only be accurate if the statistics are correct. utilExtractMemberRec returns 1 if out of records, and should be used to accurately determine when to stop extracting.

For a value of 65535, the PDS member could actually have more records.

utilExtractMemberRec

Extract is the next record.

Returns 0 for success, and 1 for no more records.

```
int utilExtractMemberRec(char* record, int* length)
```

char* record	Output	A char buffer of size lrecl, where the next record will be extracted to.

int* length	Output	The number of chars written to record.

A return value of 1 says that there are no more records, and no records were returned on this call.

utilExtractMemberClose

Must be called for every utilExtractMemberInit, except in the case of an error condition in utilExtractMemberInit or utilExtractMemberRec

```
int utilExtractMemberClose()
```

Additional information on the methods listed above can be found in the RAM utilities module header file.

Defining the RAM to CARMA

CARMA keeps its RAM information in several VSAM clusters, which must be populated with records for each of the RAMs in the environment. Refer to Chapter 4, "Customizing a RAM API using the CAF," on page 55 to learn how to insert the appropriate records for your RAM into these VSAM clusters. If you do not need to customize your RAM API, the only record you need to include in the VSAM cluster is the record for your RAM; you will not need to add parameter, return value, or action records.

Exporting functions

When CARMA attempts to load a RAM, it expects to be able to load the RAM API functions explicitly using the C dllqueryfn function. If using C, a #pragma export statement such as the one below is used to export each RAM function. The following example exports the initRAM function:

```
#pragma export(initRAM)
```

IDs vs. names

When a member, instance, or other type of data is being returned from the RAM to CARMA, both its ID and display name are typically returned. The ID should uniquely identify the entity to the RAM. It would be wise to return a member's absolute path (starting at the top-level container) in the ID field so that the member can easily be accessed by the RAM when future requests are made. The display name is simply the name that should be displayed on the client.

RAM predefined data structures

Most RAM functions use predefined structures to pass information back to CARMA.

The Descriptor structure consists of a 64-byte name character field and a 256-byte ID character field. It is used to describe instances, containers, and simple members. The KeyValPair structure consists of a 64-byte key field and a 256-byte value field. It is used for metadata key-value pairs. These structures are summarized in Table 4 on page 18 and Table 5 on page 18.

Table 4. `Descriptor` data structure

Field	Description
char id[256]	Unique ID to describe the entity
char name[64]	Display Name

Table 5. `KeyValPair` data structure

Field	Description
char key[64]	An index
char value[256]	The data

CRAFCDEF, a C header file in the sample library, must be included in the code for your RAM before you can use these data structures.

Logging

CARMA provides RAMs with a pointer to a logging function, a pointer to a log file, and a trace level (see Table 3 on page 9) at initialization. The trace level should be used to filter out some messages that may not interest users. The logging function takes a 16-byte sender character buffer, a 256-byte message character buffer, and the log file pointer that is passed in at initialization. An example call in C follows:

```
if(traceLevel > 1)
       (*writeToLog)("MyRAM", "Gathering instances", logPtr);
```

The job spool will indicate the name of the log created.

Dealing with unsupported operations

If you are developing a RAM that communicates with an SCM that does not support a CARMA operation, you should inform the client that it is disabled by appropriately modifying your RAM's CAF information (see Chapter 4, "Customizing a RAM API using the CAF," on page 55). You may assume that CARMA clients will not invoke actions marked as disabled. However, you should still account for the possibility of a client invoking a disabled action by taking one of the two following actions:

- Do not implement the function for the disabled action and do not include a `pragma export` statement for the function. This will cause CARMA to return a return code of 38 to any client that requests that operation from your RAM.

- Implement the function for the disabled action to simply return a return code of 107. Include the #pragma export statement for the function as you normally would.

Handling custom parameters and return values

Custom parameters are passed to the RAM using the `void**` params parameter. `params` is an array of `void` pointers that point to variables of several types. If these custom parameters have been defined as required parameters for a given function in the CARMA VSAM clusters (see Chapter 4, "Customizing a RAM API using the CAF," on page 55 for more information), it should be assumed that the client has set up the `params` properly. To retrieve the parameters, simply typecast the variables in `params` back to their proper types. Notice how `params` uses a char* for strings instead of a char**. Use the following C code as an example:

```
int param0;
char param1[30];
double param2;

param0= *( (int*) params[0]);
memcpy(param1, params[1], 30);
param2 = *( (double*) params[2]);
```

A pointer to an unallocated custom return values array is passed to the RAM as
void*** customReturn. If custom return values are defined in the CARMA VSAM
clusters, the RAM must allocate memory for customReturn and fill it appropriately.
Because the client must free the memory created in the RAM, it is important RAM
developers allocate memory for each return value seperately. The following C code
demonstrates returning an int, a string, and a double:

```
/* These are defined at the top */
int* return0;
char* return1;
double* return2;

/* Program body */

return0 = malloc(sizeof(int));
*return0 = 5;
return1 = malloc(sizeof(char) * 10);
memcpy(return1, "THE STRING", 10);
return2 = malloc(sizeof(double));
*return2 = 3.41;
/* Allocate and fill the return value structure */
*customReturn = malloc(sizeof(void*) * 3);
(*customReturn)[0] = (void*) return0;
(*customReturn)[1] = (void*) return1;
(*customReturn)[2] = (void*) return2;
```

If no custom return values are defined in the CARMA VSAM clusters,
customReturn should be set to NULL.

CARMA Defined Metadata

RAM specified file extension

The RAM provides the ability to suggest file extensions for CARMA resources in
CARMA clients that use the RAM. File extensions provide the client with insight
into the appropriate editor to use with a specific CARMA resource. Allowing the
RAM to specify the file extension eliminates the need for the user to specify
extensions on every resource.

File extensions can be acquired from three different sources:
- The RAM
- The client
- A parent container

The RAM can be configured to suggest file extensions to the client that can be used
in conjunction with CARMA resources. For example, assuming the RAM metadata
property "carma.file-extension" is set to "foo", and the client is set to look to the
RAM for an extension. The file name for the CARMA resource "Name" would be
displayed in the client as "Name.foo". This is because CARMA will look to the
RAM for a file extension if the client is configured to accept an extension from the

RAM. By default, the RAM does not suggest the file extension. However, it can be assumed that the client will provide an extension if one is not already provided by the RAM.

Table 6. RAM suggested file extension

Display Name	RAM Metadata Property (carma.file-extension)	Client Extension Property (set to accept the RAM's suggestion)	File Name in Client
Name	.foo	<unset>	Name.foo

Once the RAM has specified the file extension however, it is then up to the discretion of the client to either accept the suggested file extension or use one defined within the client. In the example provided in Table 6, the extension provided by the RAM was "foo", so the CARMA resource "Name" was displayed within the client as "Name.foo". Now assuming that the client has been set to not use the extension provided by the RAM and apply one of its own. The file "Name.foo" would be altered to display "Name.ext" where "ext" is the new extension specified within the client. In the event that the display name already has a file extension associated with it, the client can not remove the extension from the display name; it can only append a new extension to the existing file name.

Table 7. Client specified file extension. Client overrides the RAM suggested file extension and applies its own.

Display Name	RAM Metadata Property (carma.file-extension)	Client Extension Property (set to ignore the RAM's suggestion)	File Name in Client
Name	.foo	.ext	Name.ext
Name.foo	<unset>	.ext	Name.foo.ext

In the event that a file extension is not predefined within the RAM metadata property (carma.file-extension = <unset>), a CARMA resource will then direct itself to the client for an extension. If the client does not specify a file extension either, the CARMA resource will then inherit the default extension of its parent container.

Table 8. Inheritance of file extension. A file extension is not specified at any level, so the resource inherits an extension from its parent. An extension of "dft" represents the default extension of a parent as dictated by the CARMA client.

Display Name	RAM Metadata Property (carma.file-extension)	Client Extension Property	File Name in Client
Name	<unset>	<unset>	Name.dft

CARMA Version

The RAM provides the ability to track all available versions of CARMA members through the use of a specific metadata key: carma.version. By providing the carma.version key in the member info list, CARMA can provide specific functionality for versioned resources. For example, CARMA members that support version tracking may differ from members that do not support version tracking. The actions available on version enabled members depend on the SCM the

member originates from as well as the RAM used to connect to the SCM. It is up to the RAM developer to decide which actions to enable such as making versions editable, read-only, or providing access to past versions. When CARMA performs functions on members that have been version enabled, by default, the functions will always reference the most recent version of a member unless otherwise specified.

The use of the member info key does not uniquely identify the CARMA member. Each versioned CARMA member must have a unique member ID to indicate which version is being acted upon specifically. For example, a CARMA member with ID "member1" has 2 versions – version 1 and version 2. The member versions can be uniquely identified by appending a version number to the ID. See the example in Table 9. The RAM must be able to uniquely identify the member version based on the ID in order to provide functionality to support versioned members – such as checkin, extractMember, performAction, etc.

Table 9. CARMA member versioning example.

Member Version	Example Member ID
Version 1	member1_v1
Version 1.1	member1_v1.1
Version 2	member1_v2

For detailed information on calling version lists for CARMA members, refer to the section "getVersionList" on page 41.

carma.read-only

The RAM provides the ability to specify a CARMA member is read-only through the use of a specific metadata key: carma.read-only

If a RAM sets the value of carma.read-only on a member to "TRUE", "YES", or "Y" (case insensitive), then the client will mark that member as read-only and prevent calls to putMember.

Note: The metadata key carma.read-only is new in Developer for System z Version 8.5. Previous versions of the CARMA client do not support this metadata.

Note: The read-only functionality only applies to putMember. Copy to and from RAM remain unchanged.

State functions

The RAM has three state functions: `initRAM`, `terminateRAM`, and `reset`, as illustrated in Figure 8 on page 22. `initRAM` initializes the global variables of the RAM and establishes the connection to the repository. It cannot be called again within a session until the RAM has been terminated. `reset` restores the repository connection to its initial state. It can be called at any time except immediately after `terminateRAM`. `terminateRAM` can also be called at any time, but the only function that can be successfully called immediately after `terminateRAM` is `initRAM`.
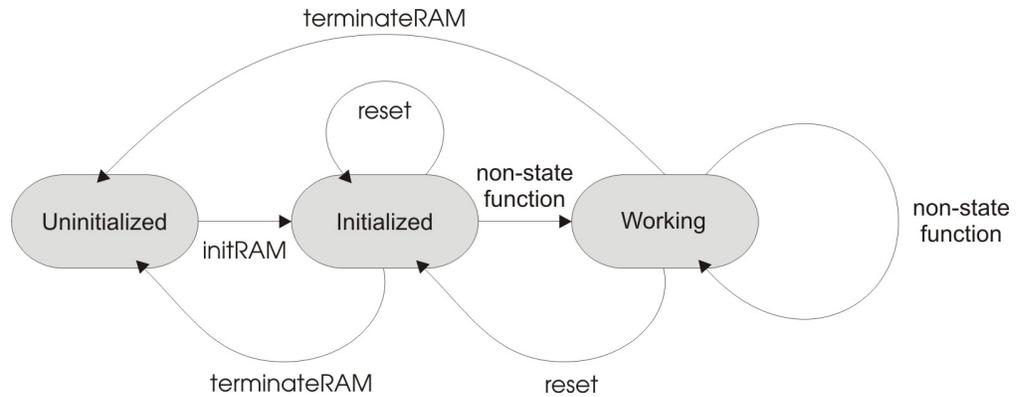
Figure 8. RAM state diagram

initRAM

```
int initRAM(Log_Func logFunc, FILE* log, int traceLev,
            char locale[8], char codepage[5], char error[256])
```

Log_Func logFunc	Input	A function pointer to the CARMA logging function. This should be stored for use in other RAM functions.
FILE* log	Input	A file pointer to the CARMA log. This should be stored for use along with the logging function.
int traceLev	Input	The logging trace level to be used throughout the session.
char locale[8]	Input	Tells CARMA the locale of the strings that will be returned to the client.
char codepage[5]	Input	Tells CARMA the code page of the strings that will be returned to the client.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

initRAM must be called before all other RAM operations occur. It should be used to initialize the SCM connection and to set up any global variables used within the program. Among these global variables should be ones used to store the three variables passed into this function.

terminateRAM

```
void terminateRAM(char error[256])
```

char error[256]	Output	If an error occurs, this should be filled with a description of the error.

terminateRAM should be used to close the SCM connection, and to free any
resources used by the RAM (such as memory and files).

reset

```
int reset(char buffer[256])
```

char error[256]	Output	If an error occurs, this should be filled with a description of the error.

reset is used to restore the SCM connection to its initial state.

Browsing functions

getInstances

Retrieves the list of instances available in the SCM.

```
int getInstances(Descriptor** records, int* numRecords, void** params,
                 void*** customReturn, char filter[256],
                 char error[256])
```

Descriptor** records	Output	This is allocated and filled with the IDs and names of the available instances.
int* numRecords	Output	The number of records that have been allocated and returned.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char filter[256]	Input	This can be passed from the client to filter out sets of instances.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Operation:
1. Query the SCM for its list of instances, possibly applying a filter.
2. Allocate the records array. If developing a RAM in C, use the following code:

   ```
   *records = (Descriptor*) malloc(sizeof(Descriptor) * *numRecords);
   ```
3. Fill the records array with the IDs and names.

If it is not possible to query the SCM for instances, it may be useful to have the
client pass in a list of known instances using the filter buffer. The RAM should
then check the list and return the instances in the records array. The instances can
be hard-coded if they are constant for the SCM.

getInstancesWithInfo

Retrieves the list of instances available in the SCM and the metadata associated with those instances. This is an optional function. The RAM should implement this function if the SCM provides the metadata with the list of instances. The RAM is required to implement getInstances even if getInstancesWithInfo is implemented.

```
int getInstancesWithInfo(DescriptorWithInfo** records,
              int* numRecords, void** params, void*** customReturn,
              char filter[256], char error[256])
```

DescriptorWithInfo** records	Output	This should be allocated and filled with the IDs and names of the available instances. For each instance, the metadata information should be allocated and filled.
int* numRecords	Output	The number of records that have been allocated and returned.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char filter[256]	Input	This can be passed from the client to filter out sets of instances.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Operation:

1. Query the SCM for its list of instances and metadata, possibly applying a filter.
2. Allocate the records array. If developing a RAM in C, use the following code:

   ```
   *records = (DescriptorWithInfo*)
                   malloc(sizeof(DescriptorWithInfo) * *numRecords);
   ```
3. For each record in the array, do the following:
   a. Fill the id and name of the instance
   b. Set the `infoCount` to the number of metadata key-value pairs associated with that instance.
   c. Allocate the `info` array. If developing a RAM in C, use the following code, where i is the zero-based index of the instance you are working with:

      ```
      (*records)[i].info = (KeyValPair *)
              malloc(sizeof(KeyValPair) * (*records)[i].infocount);
      ```
   d. Fill in the info array with the metadata key-value pairs for the instance.

If it is not possible to query the SCM for instances, it may be useful to have the client pass in a list of known instances using the `filter` buffer. The RAM should

then check the list and return the instances in the records array. The instances can be hard-coded if they are constant for the SCM.

Note: If the getInstancesWithInfo function requires custom parameters or custom returns, set up the function to use the same custom parameters and custom returns as the getInstances function. This allows the CARMA client to call the appropriate function with no additional input from the user.

getMembers

Retrieves the list of members within an instance.

```
int getMembers(char instanceID[256], Descriptor** members,
               int* numRecords, void** params, void*** customReturn,
               char filter[256], char error[256]);
```

char instanceID[256]	Input	The instance for which the members should be returned.
Descriptor** members	Output	This should be allocated and filled with the IDs and names of the members within the instance.
int* numRecords	Output	The number of members for which the array has been allocated.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char filter[256]	Input	This can be passed from the client to filter out sets of members.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Operation:

1. Query the SCM for the given instance's members, possibly applying a filter.
2. Allocate the `members` array. If developing a RAM in C, use the following code:
   ```
   *members = malloc(sizeof(Descriptor) * *numRecords);
   ```
3. Fill the `members` array with the IDs and names of the members.

getMembersWithInfo

Retrieves the list of members available within the specified instance and the metadata associated with those members. This is an optional function. The client should call this function if it wants to retrieve a list of members and the metadata associated with all of those members. If the RAM does not support this function, the client should fall back to calling getMembers.

```
int getMembersWithInfo(char instanceID[256], MemberDescriptorWithInfo** members,
                       int* numRecords, void** params, void*** customReturn,
                       char filter[256], char error[256]);
```

char instanceID[256]	Input	The instance for which the members should be returned
MemberDescriptorWithInfo** members	Output	This should be allocated and filled with the IDs and names of the members within the instance. For each member, the metadata information should be allocated and filled. It should also be indicated if the member is a container.
int* numRecords	Output	The number of members for which the array has been allocated.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char filter[256]	Input	This can be passed from the client to filter out sets of members.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Operation:

1. Query the SCM for the given instance's members and associated metadata, possibly applying a filter.
2. Allocate the members array. If developing a RAM in C, use the following code:

   ```
   *members = (MemberDescriptorWithInfo*)
           malloc(sizeof(MemberDescriptorWithInfo) * *numRecords);
   ```

3. 3.For each member in the array, do the following

   a. Fill the ID and name of the member.

   b. Set if the member is a container. Set the isContainer field to 1 if it is a container, 0 if it is not.

   c. Set the infoCount field to the number of metadata key-value pairs associated with that member.

   d. Allocate the info array. If developing a RAM in C, use the following code, where i is the zero-based index of the member you are working with:

   ```
   *members)[i].info = (KeyValPair *)
           malloc(sizeof(KeyValPair) * (*members)[i].infocount);
   ```

   e. Fill in the info array with the metadata key-value pairs for the member.

Note: If the getMembersWithInfo function requires custom parameters or custom returns, set up the function to use the same custom parameters and custom returns as the getMembers function. This allows the CARMA client to call the appropriate function with no additional input from the user.

isMemberContainer

Sets isContainer to true if a member is a container; false if not

```
int isMemberContainer(char instanceID[256], char memberID[256],
                      int* isContainer, void** params,
                      void*** customReturn, char error[256])
```

char instanceID[256]	Input	The instance containing the member being checked.
char memberID[256]	Input	The member that is being checked.
int* isContainer	Output	Should be set to 1 if the member is a container; 0 if not.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Set *isContainer to 1 if the member is a container, or 0 if it is not a container.

getContainerContents

Retrieves the list of members available within a container

```
int getContainerContents(char instanceID[256], char memberID[256],
                         Descriptor** contents, int* numMembers,
                         void** params, void*** customReturn,
                         char filter[256], char error[256])
```

char instanceID[256]	Input	The instance containing the container.
char memberID[256]	Input	The container's ID.
Descriptor** contents	Output	Should be allocated and filled with the IDs and names of the members within the container.
int* numRecords	Output	The number of members for which the array has been allocated.

void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char filter[256]	Input	This can be passed from the client to filter out sets of members.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Operation:

1. Query the SCM for the given container's members, possibly applying a filter.
2. Allocate the contents array. If developing a RAM in C, use the following code:

   ```
   *contents = malloc(sizeof(Descriptor) * *numMembers);
   ```
3. Fill the contents array with the IDs and names of the members.

getContainerContentsWithInfo

Retrieves the list of members within a container and the metadata associated with those members. This is an optional function. The RAM should implement this function if the SCM provides the metadata with the list of members. The RAM is required to implement getContainerContents even if getContainerContentsWithInfo is implemented.

```
int getContainerContentsWithInfo(char instanceID[256],
              char memberID[256], MemberDescriptorWithInfo** members,
              int* numRecords, void** params, void*** customReturn,
              char filter[256], char error[256]);
```

char instanceID[256]	Input	The instance containing the container.
char memberID[256]	Input	The container's ID.
MemberDescriptorWithInfo** members	Output	This should be allocated and filled with the IDs and names of the members within the container. For each member, the metadata information should be allocated and filled. It should also be indicated if the member is a container.
int* numRecords	Output	The number of members for which the array has been allocated.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).

void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char filter[256]	Input	This can be passed from the client to filter out sets of members.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Operation:

1. Query the SCM for the given container's members and associated metadata, possibly applying a filter.
2. Allocate the members array. If developing a RAM in C, use the following code:

   ```
   *members = (MemberDescriptorWithInfo*)
           malloc(sizeof(MemberDescriptorWithInfo) * *numRecords);
   ```

3. For each member in the array, do the following

   a. Fill in the ID and name of the member

   b. Set if the member is a container. Set the `isContainer` field to 1 if it is a container, 0 if it not.

   c. Set the `infoCount` field to the number of metadata key-value pairs associated with that member.

   d. Allocate the info array. If developing a RAM in C, use the following code, where i is the zero-based index of the member you are working with:

   ```
   (*members)[i].info = (KeyValPair *)
           malloc(sizeof(KeyValPair) * (*members)[i].infocount);
   ```

   e. Fill in the `info` array with the metadata key-value pairs for the member.

Note: If the getContainerContentsWithInfo function requires custom parameters or custom returns, set up the function to use the same custom parameters and custom returns as the getContainerContents function. This allows the CARMA client to call the appropriate function with no additional input from the user.

Create/Delete

Create and delete provides functionality to create and delete both members and containers within a CARMA environment.

createMember

Creates a new member.

```
int createMember(char instanceID[256], char memberID[256], char name[64],
        char parentID[256], int* lrecl, char recFM[4], void** params,
        void*** customReturn, char error[256])
```

char instanceID[256]	Input	The instance containing the member being created.
char memberID[256]	Output	The ID of the member that is being created.
char name[64]	Input/Output	ID of the member being created.

char parentID[256]	Input	ID of parent container. (If no parent exists, space must be filled.)
int* lrecl	Output	The number of columns in the data set and array.
char recFM[4]	Output	Contains the data set's record format (FB, VB, etc.).
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

createContainer

Creates a new container.

```
int createContainer(char instanceID[256], char memberID[256], char name[64],
        char parentID[256], void** params, void*** customReturn, char error[256])
```

char instanceID[256]	Input	The instance containing the container being created.
char memberID[256]	Output	The ID of the container that is being created.
char name[64]	Input/Output	ID of the container being created.
char parentID[256]	Input	ID of parent container. (If no parent exists, space must be filled.)
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

delete

Deletes a member or container

```
int delete(char instanceID[256], char memberID[256], int force, void** params,
        void*** customReturn, char error[256])
```

char instanceID[256]	Input	The instance containing the member or container being deleted.
char memberID[256]	Input	The ID of the member that is being deleted.
int force	Input	Used to force a delete. A value of 1 will force a delete.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

The delete function may be used to delete both members and containers, however, it should not be used to delete a RAM Instance.

File transfer functions

extractMember

Retrieves a member's contents

```
int extractMember(char instanceID[256], char memberID[256],
                  char*** contents, int* lrecl, int* numRecords,
                  char recFM[4], int* moreData, int* nextRec,
                  void** params, void*** customReturn, char error[256])
```

char instanceID[256]	Input	The instance containing the member.
char memberID[256]	Input	The ID of the member being extracted.
char*** contents	Output	Will be allocated as a two-dimensional array to contain the member's contents.
int* lrecl	Output	The number of columns in the data set and array.
int* numRecords	Output	The number of records in the data set or the number of rows in the array.
char recFM[4]	Output	Will contain the data set's record format (FB, VB, etc.).

int* moreData	Output	Set the value of the variable to which this points as 1 if extract should be called again (because there is still more data to be extracted). Otherwise, assign the value to which it points as 0.
int* nextRec	Input/Output	**Input:** The member record where the RAM should begin extracting. **Output:** The first record in the data set that wasn't extracted if *moreData is set to 1; otherwise, undefined.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

extractMember returns the contents of the data set in a two-dimensional array. The function is designed to support sending the data in chunks, so that the array does not have to be allocated to the entire size of the file. The records in the data sets are considered to be indexed with the first record being record 0.

Operation:

1. Determine how many records are in the data set, what lrecl and the record formats are, and set *lrecl and recFM.

    a. If the *numRecords - nextRec is greater than RAM's data chunk size, set *numRecords to the data chunk's number of records, and set *moreData to 1; finally, allocate the array.

    b. Otherwise, set *numRecords to *numRecords - *nextRec and allocate the array. If developing a RAM in C, use the following code:

       ```
       *contents = (char**) malloc(sizeof(char*) * (*numRecords));
       **contents = (char*) malloc(sizeof(char) * (*lrecl) * (*numRecords));
       for(i = 0; i < *numRecords; i++)
           (*contents)[i] = ( (**contents) + (i * (*lrecl)) );
       ```

2. Fill the array with the expected set of records. Ensure that the records are not null-terminated. If there is more data to return, set *nextRec to the 0-based index of the next record.

Example

Setup: The member contains 26 records, each containing the next alphabetic character, starting with "A" in record 0. Its *lrecl value is 5, its recFM value is "FB", and the RAM's data chunk size is 10.

Figure 9 shows what `extractMember` should return for each call needed to extract all the contents.

First Call	Second Call	Third Call
A B C D E F G H I J	K L M N O P Q R S T	U V W X Y Z
*lrecl = 5 *numRecords = 10 *moreData = 1 *nextRec = 10	*lrecl = 5 *numRecords = 10 *moreData = 1 *nextRec = 20	*lrecl = 5 *numRecords = 6 *moreData = 0 *nextRec = X

*Figure 9. Example of return values for subsequent calls to `extractMember`. Notice that during the third call, *nextRec has a listed value of X. This means that the value of *nextRec is not significant and will not need to be altered.*

putMember

Updates a member's contents or creates a new member if the specified `memberID` does not exist within the instance.

```
int putMember(char instanceID[256],
              char memberID[256], char** contents, int lrecl,
              int* numRecords, char recFM[4], int moreData,
              int nextRec, int eof, void** params,
              void*** customReturn, char error[256])
```

char instanceID[256]	Input	The instance containing the member.
char memberID[256]	Input	The ID of the member being updated/created.
char** contents	Input	Contains the new member contents.
int lrecl	Input	The number of columns in the data set and array.
int* numRecords	Input/Output	The number of records in the data set or the number of rows in the array.
char recFM[4]	Input	Contains the data set's record format (FB, VB, etc.).
int moreData	Input	Will be 1 if the client has more chunks of data to send; 0 otherwise.

int nextRec	Input	The record in the data set to which the 0th record of the contents array maps.
int eof	Input	If 1, denotes that the last row of the array should mark the last row in the data set; 0 otherwise.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Like extractMember, putMember supports the data being sent in chunks. putMember should also support clients that wish to pass data chunks that are not in sequential order. For example, a client may send records 10 through 19, 20 through 29, and then 0 through 9. The RAM should handle such a situation and properly update the member, or return an error code and fill the error buffer with a string stating that it cannot handle such a situation.

numRecords describes how many records the client would like to update/write on input, and the RAM should set it to the number of records that were actually written for output. If there is a difference between the two, the client will attempt to put in the members that were not written. Therefore, after receiving a response from the RAM, the client will set nextRec to the new numRecords value plus nextRec on its next putMember call.

For putMember, nextRec tells the RAM where to begin writing the contents buffer that has been passed in. For example, if nextRec is 0, the RAM should start at the beginning of the member.

moreData signifies that the client will be calling putMember again with another chunk. It is up to the RAM developer to decide how to handle a situation where moreData is set and the next call to the RAM is not a call to the putMember function providing the next chunk of data. In such a case, the RAM might simply return an error. Alternatively, it could handle the problem and move on.

eof signifies that the current contents buffer contains the last records of a member. If a 40-record member needed to be shortened to 5 records, eof would be set to 1 when the 5th record were being passed in. This should never be set when moreData equals 1.

See the source for the Skeleton RAM and the sample PDS RAM for more help (see "Locating the sample files" on page 2 for information on how to find these source files).

Operation:

1. Ensure that the `lrecl`, `numRecords`, and `nextRec` values that were passed in are valid.
2. Open up the `dataset` and write from record `nextRec` to record `nextRec + numRecords`.
3. If `eof` is specified, ensure that all records starting with the record at index `nextRec + numRecords` are removed.
4. If `moreData` is equal to 0, close the data set. If `moreData` is equal to 1, either leave the data set open if its state cannot be maintained between calls, or close the data set and make sure that it can be reopened to the appropriate place with the values being passed in next time `putMember` is called.

Extract to External

CARMA provides RAM's with the ability to extract files from an SCM into a normal host environment of PDSs and Sequential files.

copyFromExternal

Copies a member from a PDS or an SDS.

```
int copyFromExternal(char instanceID[256], char memberID[256], char external[256],
            void** params, void*** customReturn, char error[256])
```

char instanceID[256]	Input	The instance containing the member being copied.
char memberID[256]	Input	The ID of the member being copied.
char external[256]	Input	The location to copy from. Either a PDS member or an SDS member. Examples: FEK.#CUST.EXT.STOR FEK.#CUST.EXT.PDS(MEMBER).
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

copyToExternal

Copies a member to a PDS or an SDS.

```
int copyToExternal(char instanceID[256], char memberID[256], char target[256],
            void** params, void*** customReturn, char error[256])
```

char instanceID[256]	Input	The instance containing the member being copied.
char memberID[256]	Input	The ID of the member being copied.
char target[256]	Input	The location to copy to. Either a PDS member or an SDS member. Examples: FEK.#CUST.EXT.STOR FEK.#CUST.EXT.PDS(MEMBER).
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Binary file transfer

To successfully transfer files containing binary data without incurring any corruption, the RAM uses a designated set of functions to extract and put binary members from an SCM. Once a binary member has been extracted from an SCM, the RAM then hands the member to CARMA390, which continues to pass it along until the member reaches the user's machine. At each stage of the transfer process, the member is recognized as containing binary data, and no changes are applied to the member because they would result in corruption of the data.

extractBinMember

Retrieves a binary member's contents.

```
int putBinMember(char instanceID [256], char memberID [256],
          char** contents, int* length, int* moreData, int start,
          void** params, void*** customReturn, char error [256])
```

char instanceID[256]	Input	The instance containing the member being extracted.
char memberID[256]	Input	The ID of the member that is being extracted.
char** contents	Output	Pointer to the member's contents.
int* length	Output	The length of the member's contents.
int* moreData	Output	If extract should be called again because there is more data, set the value of the variable to which this points to 1, otherwise assign the value to which it points to 0.
int start	Input	The byte location of the file to start extracting from.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

putBinMember

Updates a binary member's contents or creates a new member if the specified memberID does not exist within the instance.

```
int putBinMember(char instanceID [256], char memberID [256],
          char* contents, int length, int moreData, int start,
          void** params, void*** customReturn, char error [256])
```

char instanceID[256]	Input	The instance containing the member being updated/created.
char memberID[256]	Input	The ID of the member that is being updated/created.
char* contents	Input	Contains the new members contents.
int length	Input	Pointer to the length of data to be written.
int moreData	Input	Will be 1 if the client has more chunks of data to send; 0 otherwise.
int start	Input	The byte location of the file to start putting data.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Metadata functions

getAllMemberInfo

Retrieves all of a member or instance's metadata.

```
int getAllMemberInfo(char instanceID[256], char memberID[256],
                KeyValPair** metadata, int* num, void** params,
                void*** customReturn, char error[256])
```

char instanceID[256]	Input	The ID of the instance containing the member.
char memberID[256]	Input	The ID of the member for which metadata is being returned. The ID may be empty (set as all spaces) if member info is to be retrieved for the instance instead of a specific member.
KeyValPair** metadata	Output	This should be allocated and filled with all the metadata key-value pairs for the specified member.
int* num	Output	The number of key-value pairs for which the array has been allocated.

void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Operation:

1. Query the SCM for the given member's metadata.
2. Allocate the `contents` array. If developing a RAM in C, use the following code:

   ```
   *metadata = malloc(sizeof(KeyValPair) * *num);
   ```
3. Fill the `contents` array with the key-value pairs.

getMemberInfo

Retrieves a specific piece of a member or instance's metadata.

```
int getMemberInfo(char instanceID[256], char memberID[256],
                  char key[64], char value[256], void** params,
                  void*** customReturn, char error[256])
```

char instanceID[256]	Input	The ID of the instance containing the member.
char memberID[256]	Input	The ID of the member whose metadata is being retrieved. If set as all spaces, the metadata for the instance should be returned.
char key[64]	Input	The key for the value to be returned.
char value[256]	Output	The requested value.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

`getMemberInfo` returns the value of the specified key for the given member.

updateMemberInfo

Updates a specific piece of a member or instance's metadata.

```
int updateMemberInfo(char instanceID[256], char memberID[256],
                     char key[64], char value[256], void** params,
                     void*** customReturn, char error[256])
```

char instanceID[256]	Input	The ID of the instance containing the member.
char memberID[256]	Input	The ID of the member whose metadata is being set. If set as all spaces, the metadata for the instance should be set.
char key[64]	Input	The key for the value to be set.
char value[256]	Input	The value to set.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

updateMemberInfo attempts to update a member's metadata (specified by the given key) with the given value.

Other operations

lock

Locks the member.

```
int lock(char instanceID[256], char memberID[256], void** params,
         void*** customReturn, char error[256])
```

char instanceID[256]	Input	The instance containing the member.
char memberID[256]	Input	The ID of the member being locked.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

unlock

Unlocks the member

```
int unlock(char instanceID[256], char memberID[256], void** params,
           void*** customReturn, char error[256])
```

char instanceID[256]	Input	The instance containing the member.
char memberID[256]	Input	The ID of the member being unlocked.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

check_in

Checks in the member. This only consists of setting a flag to mark that it is checked in.

```
int check_in(char instanceID[256], char memberID[256], void** params,
             void*** customReturn, char error[256])
```

char instanceID[256]	Input	The instance containing the member.
char memberID[256]	Input	The ID of the member being checked in.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

check_out

Checks out the member. This only consists of setting a flag to mark that it is checked out.

```
int check_out(char instanceID[256], char memberID[256], void** params,
              void*** customReturn, char error[256])
```

char instanceID[256]	Input	The instance containing the member.
char memberID[256]	Input	The ID of the member being checked out.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

performAction

Performs the action identified in the actionID by using the parameters given and the return values in customReturn (when applicable).

```
int performAction(int actionID, char instanceID[256], char memberID[256],
            void** params, void*** customReturn, char error[256])
```

int actionID	Input	The custom action that is being requested, as defined in the CRADEF VSAM.
char instanceID[256]	Input	The instance the action is being performed on. If this and memberID are both set to all spaces, this indicates the action should be performed on the RAM.
char memberID[256]	Input	The member the action is being performaed on. If this is set to all spaces, this indicates the action should be performed on the instance.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

getVersionList

Provides a list of versions available for a given member.

```
int getVersionList(char instanceID[256], char memberID[256],
     VersionIdent** versions, int* num, void** params,
          void*** customReturn, char error[256])
```

char instanceID[256]	Input	The instance the member is within.
char memberID[256]	Input	The member to get a list of versions for.
int* num	Output	The number of versions.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

VersionIdent will be identified by the following struct:

```
typedef struct {
char memberID[256];  /*A versioned memberID, such as
     baseMemberID_VerNum*/
char versionKey[64]; /* A way to refer to the version, such as
                          "1.2.3"...should be the same as the value
                          for the carma.version metadata key*/
   char comments[256];  /* RAM supplied comments on the version,
                          could be timestamp, changes, etc..  */
} VersionIdent;
```

The version list should be a complete ordered version list, but the RAM Developer can chose to use a 'versioned' ID for the current version, or to use the unchanging ID. As an example, current version of a member might be accessible via "location(Member)" or "location(Member)_1.4" where the file is on version 1.4. The RAM developer could therefore choose to return either "location(Member)_1.4" or "location(Member)" as the newest version in the list.

When returning a list of members through browsing functions, such as getMembers, the memberIDs returned SHOULD NOT include the version. Changing the memberID for a member prevents a CARMA client from properly tracking that member.

In order to support versioning, RAM Developers should handle CARMA calls when presented with a 'versioned' ID for the memberID.

If a RAM developer wants to support versioning for some, but not all of the members, a return code of 130, which stands for "Member does not support versioning" can be used.

RAM development using COBOL

While the C programming language is a sufficient choice for the development of most RAMs, you may occasionally find it beneficial to develop a RAM in COBOL. Be warned that while there are certain advantages to using COBOL for RAM development, there are also certain disadvantages as well:

Advantages of RAM development in COBOL

- Code between functions is more clearly separated, enforcing stringent design and mandating a careful inventory of shared resources between RAM program functions.
- Since COBOL is heavily associated with the host, the facilities for COBOL development may be more readily available on your system.
- Since string manipulation in COBOL does not rely on NULL delimiters, protection exceptions are less likely than they would be during C development.
- RAMs that involve the incorporation of business logic implementation or heavy amounts of data shuffling are simpler to develop in COBOL.
- COBOL code has the property of being self-documenting.

Disadvantages of RAM development in COBOL

- Dynamic structures used by CARMA are cumbersome to deal with in COBOL.
- Usage of additional C-style facilities involves adding C code to COBOL-to-C source.
- Data typing available within C is not available in COBOL. You must exercise more care when dealing with pointers.

CARMA ships with a sample RAM developed in COBOL, appropriately called the sample COBOL RAM. This sample COBOL RAM requires the COBOL-to-C source in order to function properly. You may use this RAM as a starting point for your own RAM written in COBOL, but the provided sample COBOL RAM should not be used in a production environment.

Note: In order to use the sample COBOL RAM, you must update the Custom Action Framework (CAF) information in the VSAM clusters. Details on how to accomplish this can be found in the IBM Rational Developer for System z Host Configuration Guide (SC31-6930-02).

COBOL RAM program structure

Coding the program ID

RAMs developed in C implement CARMA RAM API functions, such as `initRAM` or `getMembers`. RAMs developed in COBOL implement each of these functions as individual COBOL programs (called *RAM function programs*). At compile time, the source code for each program is concatenated and compiled into a single DLL. Each program ID is exported to a definition side deck if the DLL is compiled to a PDS. The program ID of each RAM function program should match the name of the RAM function implemented by that program.

Note: This matching should be case-sensitive. For instance, the following code would define the program that implements the `getInstances` RAM function:

```
PROGRAM-ID.    'getInstances'.
```

The linkage section

Within a COBOL RAM function program, the linkage section is used for defining parameter values, establishing addressability to pointer values passed as parameters, and referencing the integer value returned by the RAM function.

Each parameter being passed to the RAM function should be defined as a 77-level item. Although these parameters cannot be grouped as 77-level items, it is recommended that they be defined adjacent to each other in the same sequence that they are passed to the program (for clarity, locality of reference, and readability).

Note: To help ease development, an example copy book with pre-defined parameters for use in a linkage section can be found in the sample library member CRACPY05.

For example, you could use the following code to define the parameters for the `getInstances` RAM function program:

```
77  ARG-RECORDS                POINTER.
77  ARG-NUMRECS                PIC S9(9) BINARY.
77  ARG-PARAMS                 POINTER.
77  ARG-RETURNS                POINTER.
77  ARG-FILTER                 PIC X(256).
77  ARG-ERROR                  PIC X(256).
77  INT-RVAL                   PIC S9(9) BINARY.
```

Note: The items used in the above procedure division are displayed as they are defined in the copy book.

77-level items should also be defined for areas referenced by pointers that are not dynamic in size. For instance, a definition should exist for referencing the 256-byte error buffer. Use the following definition for this error buffer:

```
77  ERROR-BUFFER            PIC X(256).
```

The linkage section should also contain a reference to the integer value being returned from the RAM function (the return code). Define this integer using the following code:

```
77  INT-RVAL               PIC S9(9) BINARY.
```

Addressability to the return code need not be established. It may simply be used as if it were defined within the working storage section.

Defining the procedure division

Parameters should be established with a `USING` phrase so that they can be made available to the COBOL program. Since parameters can be passed by reference or value, you should determine which method is most appropriate for your parameters depending upon the coding practices in use.

The following example procedure division for the `getInstances` declaration illustrates how you might designate parameters to be passed.

```
PROCEDURE DIVISION USING BY VALUE ARG-RECORDS
                         BY REFERENCE ARG-NUMRECS
                         BY VALUE ARG-PARAMS
                         BY VALUE ARG-RETURNs
                         BY REFERENCE ARG-FILTER
                         BY REFERENCE ARG-ERROR
                         RETURNING INT-RVAL.
```

Since each RAM function returns an integer value, the `RETURNING` phrase is used to specify that an integer value is being returned from the COBOL program.

Note: The order specified in the procedure division must also match the order defined in the API prototype.

Ending the program

Since each COBOL RAM function program serves the purpose of a C RAM function, each RAM function program should be terminated with an `END PROGRAM` directive. When compiling a COBOL RAM DLL, the COBOL source programs are provided to the COBOL compiler as a series of concatenated DD statement. Failing to provide `END PROGRAM` directives will cause programs to be treated as nested, which will yield compiler error messages.

Passing values from C to COBOL

Function arguments passed from a C program into a COBOL RAM function program must be handled in a manner that is appropriate to the method by which they are being passed. More information on this topic can be found in the guide: *Language Environment® Writing Interlanguage Communication Applications*. For specific information on passing values between languages, refer to Chapter 4, *Communicating between C and COBOL*. All examples within this section refer to behavior in which equivalent data types must be defined without the use of `#pragma` in the calling C program.

There are two ways of using parameters that have been passed from C. Parameters can be included with the `USING BY VALUE` phrase or the `USING BY REFERENCE` phrase of the procedure division header.

Receiving basic C data types passed by value

As a general rule, arguments of basic C data types such as int, double, float, or long that are passed into the C function by value should be received with the `BY VALUE` phrase in the COBOL program's procedure division. For information on each basic C data type passed BY VALUE and how it should be defined as a linkage section item, refer to *z/OS V1R8.0-V1R9.0 Language Environment Writing Interlanguage Communication Applications (SA22-7563-05)*, Chapter 4 "Communicating between C and COBOL", Table 11. "Supported Data Types Passed by Value (Direct) without #pragma".

Arguments that are passed from C using pointers (such as strings in the form of character arrays) received `BY VALUE` must be manually dereferenced using the `SET` operator. Alternatively, arguments that use pointers may also be received with the `BY REFERENCE` procedure division phrase in the receiving COBOL program, provided that there is no possibility for the passed pointer to have a NULL value. More information about this technique can be found in "Avoiding Dereferencing (Receiving C data types BY REFERENCE)" on page 46, located later in this section

Example: Receiving an integer `BY VALUE`.

In this example the COBOL program is receiving a parameter that is defined as type int in C.

First, a linkage section entry must be defined for the incoming integer value.
```
77 IN-INTEGER PIC S9(9) BINARY.
```

Then, we must add the correct information to the `PROCEDURE DIVISION` statement to make the incoming integer available to the program.

```
PROCEDURE DIVISION USING BY VALUE IN-INTEGER.
```

Within the COBOL program, `IN-INTEGER` may be used as if it were any other item in storage.

Example 2: Receiving Character Arrays `BY VALUE`.

Most of the C RAM API functions receive a space-padded C character array of 256 bytes called a memberID. In C, this array is passed by reference using a pointer.

When receiving a character array `BY VALUE`, the COBOL program receives a copy of the pointer that points to the storage location holding the characters. This pointer must be dereferenced manually before the string can be used within the COBOL program.

Define the item in the linkage section as a `POINTER`.

```
77 IN-MEMBERID   POINTER.
```

You must also define a second item in the linkage section for dereferencing the pointer.

```
77 DEREFERENCED-MEMBERID   PIC X(256)
```

Ensure that the `PROCEDURE DIVISION` receives the memberID properly.

```
PROCEDURE DIVISION USING BY VALUE IN-MEMBERID
```

Then, before working with the memberID, use the `SET` operator to dereference `IN-MEMBERID`.

```
SET ADDRESS OF DEREFERENCED-MEMBERID TO IN-MEMBERID.
```

Now `DEREFERENCED-MEMBERID` may be used as though it were defined in the working storage section:

```
MOVE 'MEMBER1' TO DEREFERENCED-MEMBERID.
```

Avoiding Dereferencing (Receiving C data types BY REFERENCE)

In receiving a parameter with the `BY REFERENCE` phrase, the COBOL program will take care of dereferencing operations provided that the item is defined properly in the linkage section. This is useful in avoiding dereferencing operations, but risky in cases where a NULL pointer may be passed into the receiving COBOL program.

Note: A COBOL program that receives a NULL pointer for an argument received `BY REFERENCE` will be likely to ABEND with a protection exception 0C4.

Example: Receiving character arrays `BY REFERENCE`.

In this example, memberID will be received `BY REFERENCE` from CARMA.

First, a linkage section entry must be defined to match the character array being passed.

```
77  IN-MEMBERID   PIC X(256).
```

The `PROCEDURE DIVISION` statement must reflect that this item is being received `BY REFERENCE`.

```
PROCEDURE DIVISION USING BY REFERENCE IN-MEMBERID.
```

IN-MEMBERID can now be used as if it were any other item defined in working storage.

```
MOVE 'MEMBER1' TO IN-MEMBERID.
```

Knowing when to receive BY REFERENCE

The following situations describe when it is appropriate for the COBOL program to receive parameters BY REFERENCE:

- The item being received is passed from C as a pointer to a simple data type that does not require multiple levels of dereferencing. (for example, int *, char *, and double *).
- The item being received is being passed into COBOL via a pointer and its value in the calling program is allowed to be changed by the called program.
- The item being received is a pointer that is guaranteed not to be NULL.

Knowing when to receive BY VALUE

The following situations describe when it is appropriate for the COBOL program to receive parameters BY VALUE:

- The item coming into COBOL is being passed by value from C. (The item is not being passed via a pointer and its value in the calling program should not be modified.)
- The item coming into COBOL is a type of pointer that will require multiple levels of dereferencing.
- The item coming into COBOL is a pointer that may potentially have a NULL value and must be validated before usage in order to prevent an exception (particularly 0C4).
- Any pointer coming into COBOL that requires manual dereferencing using the SET operator.
- Any pointers to C functions.
- Any incoming value that is a pointer and will need to have pointer arithmetic performed upon it.

Passing Data from COBOL to C

When calling C DLL functions from within COBOL, the method by which parameters are passed from the COBOL program must carefully match the data types of each of the parameters in the prototype for the receiving C function. This is necessary in order to avoid problems such as abnormal termination. The general rule is that if a C function receives an argument that is not a pointer, it should be passed from COBOL using the BY VALUE phrase. If the argument is a pointer, it should be passed using the BY REFERENCE phrase.

Passing COBOL items as basic C function arguments

Basic C data types found within function prototypes should be passed by value from the calling COBOL program. In the following example, a C function is invoked that accepts two arguments that are basic C data types from the calling COBOL program.

C function prototype:

```
int callme(int a, double b);
```

Working storage items as they should be defined in the calling COBOL program:

```
01  FUNC-ARG1    PIC S9(9).
01  FUNC-ARG2    COMP-2.
01  RETVAL       PIC S9(9) BINARY.
```

An example of the CALL statement in the COBOL program:

```
CALL "callme" USING BY VALUE FUNC-ARG1 FUNC-ARG2 RETURNING RETVAL.
```

Passing COBOL items into C functions by reference

C functions frequently receive arguments for reference modification. The most
prevalent example of this is a C-style string modification where a character array is
received via a copy of a pointer to the original string. Items may be passed from
COBOL to C for reference modification using the BY REFERENCE phrase inside the
CALL statement. The following example demonstrates such a situation.

Example:

C function prototype of receiving function:

```
int receiveString(char inString[256]);
```

Definitions for the working storage item being passed as an argument and the
return value:

```
01  THE-STRING  PIC X(256).
01  RETVAL      PIC S9(9) BINARY.
```

The CALL statement in the COBOL program:

```
CALL "receiveString" USING BY REFERENCE THE-STRING RETURNING RETVAL.
```

**Example 2: A C function that receives a pointer to an integer from the calling
COBOL program:**

C function prototype:

```
int changeInt(int * fromCOBOL);
```

Working storage entries in the calling COBOL:

```
01  THE-INT   PIC S9(9) BINARY.
01  RETVAL    PIC S9(9) BINARY.
```

The CALL statement in the COBOL program:

```
CALL "changeInt" USING BY REFERENCE THE-INT RETURNING RETVAL.
```

Dealing with pointer operations

Simple pointer operations

For most parameters passed to COBOL RAM function programs, a small amount
of pointer dereferencing code may need to be implemented using the SET operator.
For example, most programs will receive a pointer to a 256-byte buffer for a
detailed error message. Before you can fill this buffer though, it must be
dereferenced using the SET operator. For smaller items, dereferencing can be
avoided by USING BY REFERENCE.

As an example, the following code demonstrates how to establish addressability to
the error buffer. The pointer to the error buffer is passed by value to the procedure
division for getInstances and is defined in the linkage section as follows:

```
77  GIP-ERROR                    POINTER.
```

Later in the linkage section, a 77-level item is defined for dereferencing and performing operations on the error buffer:

```
77  ERROR-BUFFER                 PIC X(256).
```

Then, within the procedure division we establish addressability to the error buffer after verifying that `GIP-ERROR` is not NULL:

```
SET ADDRESS OF ERROR-BUFFER TO GIP-ERROR.
```

Now we can treat the error buffer as we would any normal 256-byte alphanumeric field. In this case, the error buffer is a 256-byte non-NULL-terminated string.

Complex pointer operations

For pointers with multiple levels of indirection, dereferencing operations can be complicated. The COBOL code to perform such dereferencing operations would require multiple 77-level items with a `SET` operation for each level of indirection. To complicate matters, dynamically allocated structures are difficult to access without knowing an absolute maximum size for the structure.

Instead of attempting complex pointer operations in COBOL, it is highly recommended that code of this nature be implemented in a modular fashion by using the COBOL-to-C source. Currently functions are implemented for memory allocation and contents buffer data insertion and retrieval. You may find it helpful to add to this code as necessary and use it for more complex operations.

Pointer Arithmetic

Alternatively, complex pointer operations can be performed within COBOL, but decrease code readability and maintainability. To deal with dynamic structures, pointer arithmetic is necessary. Pointer arithmetic is achieved through the use of redefines. To create a pointer that may be manipulated through pointer arithmetic, use code similar to the following within the working storage section:

```
01  SOME-POINTER                   POINTER.
01  SOME-POINTER-MANIP     REDEFINES SOME-POINTER.
    05   ADD-TO-ME                 PIC S9(9) BINARY.
```

After defining the pointer, you can manipulate it as necessary using the redefined version. The following code would change the pointer to point to the next structure in a contiguously allocated chunk of memory containing multiple structures.

```
ADD SIZE-OF-STRUCTURE TO ADD-TO-ME.
SET ADDRESS OF STRUCTURE TO SOME-POINTER.
```

Memory Allocation

Certain RAM functions, such as `extractMember` and `getAllMemberInfo`, require that the RAM allocate memory. This memory is later freed by CARMA, which uses C's `free` function to deallocate the memory. For this reason, a RAM implemented in COBOL must use C's `malloc` function or the Language Environment service `CEEGTST` to allocate memory. The COBOL-to-C source has a C function called `CMALLOC` to provide access to `malloc` from within COBOL code. The `CMALLOC` function accepts as an argument an integer representing the requested number of bytes and returns a pointer to the portion of memory that was allocated. It is the RAM developer's responsibility to ensure that the pointer is not NULL before attempting to use the allocated memory.

The following sample call to `CMALLOC` illustrates its use:

```
01  MALLOC-SIZE                 PIC S9(9) BINARY.
01  VOID-POINTER-RETURNED       POINTER.
MOVE 80 TO MALLOC-SIZE.
CALL "CMALLOC" USING BY VALUE MALLOC-SIZE
              RETURNING VOID-POINTER-RETURNED.
```

The Language Environment callable service CEEGTST is also available for
dynamically acquiring storage. For more information on this service and other
services provided by Language Environment refer to *z/OS V1R9.0 Language
Environment Programming Reference (SA22-7562-09).*

Variables shared between programs

Global variables that need to be shared between RAM function programs may be
declared as external. The following example illustrates how to declare variables
using the EXTERNAL keyword in the working storage entry of the initRAM function
program:

```
01  SHARED-VARIABLES EXTERNAL.
      05  LOG-FUNCTION-POINTER         FUNCTION-POINTER.
      05  TRACELEVEL                   PIC S9(9) BINARY.
      05  FILE-POINTER                 POINTER.
      05  LOCALE                       PIC X(8).
      05  CODEPAGE                     PIC X(5).
```

In the sample code above, the global variables have their values set within the call
to initRAM. Later, when terminateRAM is called, these values are displayed to show
that they are persistent and shared.

This type of definition should be used for values that need to be shared across
calls to different RAM function programs. If a working storage item will only be
accessed by one RAM function program, do not declare it as an external item.
Working storage items that do not need to be modified by other RAM function
programs should not be made external.

Handling Custom Action Framework data

The Custom Action Framework (CAF) allows you to expand upon the existing
function programs of your COBOL RAM by implementing new custom actions
that are designed to meet the needs of your CARMA client.

Handling Custom actions

Custom actions may be created by using the sample COBOL source file (CRACOB16,
located in the sample library) as an example for implementing the performAction
RAM function. Within the performAction RAM function program, use an EVALUATE
statement to selectively execute code based upon ARG-ACTIONID:

```
EVALUATE ARG-ACTIONID
         WHEN 119
             CALL 'ESREVER' USING BY VALUE ARG-PARAMS ARG-RETURNS
                 BY REFERENCE ARG-ERROR RETURNING RETCODE
             IF RETCODE NOT = 0
                 MOVE RETCODE TO INT-RVAL
                 EXIT PROGRAM
             END-IF
         WHEN OTHER
             MOVE RC-UNSUPPORTED TO INT-RVAL
             EXIT PROGRAM
      END-EVALUATE.
```

Handing Custom Parameters without using COBOL-to-C Utility Functions

Custom parameters can be retrieved through two dereferencing operations. After ensuring that the pointer passed to the RAM program is not NULL, establish addressability to the array of pointers. Then dereference each pointer to access each custom parameter that it refers to. The following excerpt from the linkage section for theperformAction RAM function program describes the fields as they are defined for dealing with two custom parameters:

```
77  PA-PARAMS                      POINTER.

01  PARAMS.
    05  PARAM1                      POINTER.
    05  PARAM2                      POINTER.

01  CUSTOM-PARAM1                   PIC S9(9) BINARY.
01  CUSTOM-PARAM2                   PIC X(8).
```

First establish addressability to the custom parameter pointer list using the following code:

```
SET ADDRESS OF PARAMS TO PA-PARAMS.
```

Then establish addressability to individual parameters:

```
SET ADDRESS OF CUSTOM-PARAM1 TO PARAM1.
SET ADDRESS OF CUSTOM-PARAM2 TO PARAM2.
```

The custom parameters can now be used as if they were normal fields in the working storage section. Also, it is assumed that the procedure division statement has specified that PA-PARMS is being used BY VALUE.

Note: The above example code does not include the checks for NULL pointers that you should include in your code.

Handling Custom Returns without using COBOL-to-C Utility Functions

Accessing custom return values within a COBOL RAM requires more caution than dealing with custom parameters. For custom returns to be established, a series of concise steps must be followed. The following code outlines linkage section items that are used to reference a list of two custom returns. It is assumed that the procedure division statement has specified that PA-PARMS is being used BY VALUE:

```
77  PA-RETURNS                     POINTER.
01  RETURNS-LV2                    POINTER.
01  RETURNS-LV3.
    05  RETURN1                     POINTER.
    05  RETURN2                     POINTER.

01  CUSTOM-RETURN1                  PIC X(8).
01  CUSTOM-RETURN2                  PIC S9(9) BINARY.
```

Begin by dereferencing the first level of indirection:

```
SET ADDRESS OF RETURNS-LV2 TO PA-RETURNS.
```

Then allocate the memory necessary for the array of pointers to the custom parameters:

```
COMPUTE MALLOC-SIZE =
    SIZE-OF-POINTER * NUM-CUSTOM-RETURNS
END-COMPUTE.
CALL "CMALLOC" USING BY VALUE MALLOC-SIZE
              RETURNING RETURN-POINTER.
```

Now set the second level pointer to point at that block of memory:

```
SET RETURNS-LV2 TO RETURN-POINTER.
```

Next, establish addressability to the list of pointers to return values that you have just allocated:

```
SET ADDRESS OF RETURNS-LV3 TO RETURNS-LV2.
```

Allocate the necessary memory for the custom parameters:

```
* Allocate space for 8 byte string
MOVE 8 TO MALLOC-SIZE.
CALL "CMALLOC" USING BY VALUE MALLOC-SIZE
                     RETURNING RETURN1.

*Allocate space for integer
MOVE 4 TO MALLOC-SIZE.
CALL "CMALLOC" USING BY VALUE MALLOC-SIZE
                     RETURNING RETURN2.
```

Note: This code automatically sets the list of pointers within a RETURNING phrase. As such, it is not necessary to set these pointers manually.

Finally, establish addressability to the return values and set them accordingly.

```
SET ADDRESS OF CUSTOM-RETURN1 TO RETURN1.
SET ADDRESS OF CUSTOM-RETURN2 TO RETURN2.
MOVE 'COBOLRAM' TO CUSTOM-RETURN1.
MOVE 42 TO CUSTOM-RETURN2.
```

Note: See the COBOL RAM sample source for documentation and examples on how to use the COBOL-to-C utility functions.

Differences between the utility DLL and the COBOL-to-C utility source

Within the CARMA documentation there are references to a *utility DLL* and *COBOL-to-C utility source*. There is potential for confusion between these two items.

The *utility DLL* name is a misnomer. The provided utility DLL is a set of C source code found within the member CRASUTIL within the CARMA SFEKSAMP library. No compiled DLL form of this source is provided. The source is intended to provide various utility functions in C that can be shared by various RAM implementations. These functions implement tasks that RAM developers may frequently need to perform within their code. This code is provided by the CARMA development team to ease the process of RAM development. You can compile the source as a DLL, or as object code and include it in the linking process for any RAM. In either case, the code is intended to be compiled with the compiler options for producing DLL code (for example: RENT,DLL). In creating the sample RAMs provided with CARMA, the utility object code was linked into the final module.

The other utility source provided, the COBOL-to-C utility source, is also C code found in the member CRACOBC1 of the SFEKSAMP library. This source is provided as a set of C functions accessible to COBOL RAM developers to simplify tasks that are cumbersome to implement in COBOL.The provided functions also make it possible to access the CARMA log, which is difficult from COBOL due to the CARMA RAM API specification for the format of the initRAM function.

Both the *utility DLL* and the "*COBOL-to-C utility*" are provided to developers as unsupported sample code with the intent of simplifying the task of RAM development. C developers will likely only consider using the *utility DLL* to develop a RAM and COBOL developers should consider utilizing both in order to simplify the development process.

Debugging and avoiding abnormal termination

There are multiple techniques and coding practices available to facilitate COBOL RAM development.

Displaying values to help debug your COBOL RAM

The DISPLAY verb can be used to inspect the values of program variables, parameters being passed, and buffers being filled. Moreover, DISPLAY statements can be most useful if they are inserted to trace the execution path. Most importantly, note that the displayed values for pointers are shown in decimal, not in hexadecimal. Output from the use of the DISPLAY verb will display in the CARMA job spool.

NULL pointers

Attempting to dereference a NULL pointer will almost certainly result in a protection exception. This effectively will result in not only the termination of the RAM, but also of CARMA. To avoid such an abnormal termination, all pointer values should be checked for NULL values. Further documentation is provided about pointers and checking for NULL values within *Enterprise COBOL for z/OS Language Reference*.

Properly exiting your RAM function programs

Conventionally STOP RUN is used to end the execution of a program written purely in COBOL. However, coding STOP RUN within a COBOL RAM will terminate both CARMA and the COBOL RAM. Avoiding STOP RUN statements is recommended unless circumstances require this sort of behavior. You should use EXIT PROGRAM instead of STOP RUN to leave execution of the COBOL RAM and return to CARMA processing.

Chapter 4. Customizing a RAM API using the CAF

The Custom Action Framework (CAF) is used by RAM developers to describe to CARMA clients how their RAM APIs differ from the standard RAM API. The CAF allows a RAM API to define the following differences between its API and the standard RAM API:

- Additional ("custom") actions
- Disabled standard actions
- Additional ("custom") parameters to standard actions
- Additional ("custom") return values to standard actions
- Fields describing metadata that should be displayed on the client

These differences are defined using CAF information. CAF information can be thought of as a contract between a RAM and the CARMA clients using that RAM; the RAM is guaranteed to run properly as long as CARMA clients follow the RAM's CAF information. Before attempting to define a RAM's CAF information, you may want to create a conceptual model of your RAM's CAF information. This will help you plan how you will define your RAMs CAF information in the CARMA VSAM clusters. This chapter provides a practical example of how to create such a model for a RAM and how to then define the CAF information for the RAM using that model.

Before you can follow the example, you should first understand the basic CAF object types. The example RAM model is designed using these objects.

CAF object types

There are five types of objects used in CAF information: RAMs, parameters, return values, actions, and fields.

RAM

RAMs provide CARMA with access to specific SCMs. CAF information for your RAM includes the following:

Name   The RAM's name

Description
A short description of the RAM

RAM ID
A numeric identifier for the RAM between 0 and 99

Programming Language
The programming language the RAM was written in (C, COBOL, or PL/I)

RAM DLL name
The name of the RAM DLL

Version
The version number of the RAM

Repository version
The repository version that the RAM was designed to work with

CARMA version
> The CARMA version the RAM was designed to work with

Parameter

Parameters are values passed to an action from the CARMA client. They are defined per-RAM; thus, once a parameter has been defined, its parameter ID can be used in the parameter list of any action defined for that RAM. This can be useful if many of the actions for a RAM require the same parameters.

CAF information for your RAM will include the following information about each parameter:

Name The parameter's name.

Description
> A short description of the parameter.

Parameter ID
> A numeric identifier for the parameter between 000 and 999 (3 bytes).

RAM ID
> The ID of the RAM the parameter belongs to.

Type The data type of the parameter. Choose from the following list of standard programming data types: int, long, double, and string.

Length
> A numeric value that is specified differently based on the parameter type:

Parameter Type	Specification Instructions
int	This value does not matter.
long	This value does not matter.
double	The precision of the parameter.
string	The field width of the parameter.

Constant
> Whether or not the parameter will always contain the same value.

Default value
> The parameter's default value. This is not optional information.

Prompt
> The prompt that should be displayed by CARMA clients when requesting a value for the parameter from users.

Return value

Return values are the result of an action called by CARMA. They are defined per-RAM; thus, once a return value has been defined, its return value ID can be used in the return value list of any action defined for that RAM. This can be useful if many of the actions for a RAM require the same return values.

CAF information for your RAM will include the following information about each return value:

Name The return value's name.

Description
> A short description of the return value.

Return value ID
A numeric identifier for the return value between 000 and 999 (3 bytes).

RAM ID
The ID of the RAM the return value belongs to (2 bytes).

Type The data type of the return value. Choose from the following list of standard programming data types: int, long, double, and string.

Length
A numeric value that is specified differently based on the return value type:

Parameter Type	Specification Instructions
int	This value does not matter.
long	This value does not matter.
double	The precision of the return value.
string	The field width of the return value.

Constant
Whether or not the parameter will always contain the same value.

Default value
The default value of the parameter.

Prompt
The prompt that should be displayed by CARMA clients when requesting from users a value for the parameter.

Action

All RAMs have a standard set of actions defined within the RAM API. You can use the CAF to modify these standard actions to use additional input parameters, to use additional return values, or to be hidden from CARMA (essentially disabling the actions).

Note: Although it is not possible to specify to the CAF that a default parameter in a standard action be removed, such a parameter can simply be ignored in the implementation of that action if passed to the action by a CARMA client.

You can also declare new ("custom") actions. Each declared custom action must have an assigned ID (called its action ID). When a CARMA client attempts to invoke a custom action in a RAM, CARMA will first call the RAM's `performAction` function, passing the action ID (provided by the CARMA client) of the custom action as a parameter. The `performAction` function should then attempt to call the function for the custom action with the specified action ID.

Note: It is the responsibility of the RAM developer to handle the case where an invalid action ID is provided to the RAM's `performAction` function. A reasonable way of handling this case would be to return an error to the client along with a detailed error message.

CAF information for your RAM will include the following information about each action. (For disabled actions, only the RAM and action IDs are required.)

Name The action's name.

Description
A short description of the action.

Action ID
A numeric identifier for the action between 0 and 999. Action IDs between 0 and 79 override standard actions (see Appendix B, "Action IDs," on page 105 for a full listing of the IDs for the standard actions). Action IDs between 80 and 99 are reserved for use by CARMA. Use an ID between 100 and 999 to define a custom action.

RAM ID
The ID of the RAM the action belongs to.

Parameter list
A list of the IDs for the parameters the action uses. If you are overriding a standard action, you only need a list of those parameters that are being added to the list of standard parameters. If you are defining a custom action, you must list the IDs of all the parameters required by the action except the instance and member IDs, which are passed by default to every custom action.

Return value list
A list of the IDs for the return values the action returns. If you are overriding a standard action, you only need a list of those return values that are being added to the list of standard return values. If you are defining a custom action, you must list the IDs of all the return values being returned by the action except for the action's return code, which must always be returned by every custom action.

Field

Fields describe metadata of particular interest to users. CAF information for fields includes the following:

Name The localized displayable name for the field.

Metadata Key
The metadata key to provide to the getMemberInfo function for the field to be displayed.

Default value
The localized displayable value for the field if no value is returned by a call to `getMemberInfo`.

Description
A localized displayable description of the metadata.

Developing the RAM model for a custom RAM

Suppose we want to create a RAM named SAMP RAM that is capable of accessing an SCM solution named Sample SCM. Assume that Sample SCM operates in a manner that would cause SAMP RAM to have the following differences from a standard CARMA RAM:

- It provides no support for checking out files.
- Its lock action returns the lock type in addition to the return values for the standard CARMA lock action.
- It has a "lock instance" action, which locks an instance within the SCM. This action requires the following parameters:
  - Instance ID

- Reason

and returns the following values:
- Lock type
- Return code

- Has a "disenflaguate" action, which removes a flag from a member within the SCM. This action requires the following parameters:
  - Instance ID
  - Member ID
  - Reason

and returns the following values:
- Return code

- Has a "concatenate" action, which concatenates the contents of two members within the SCM. This action requires the following parameters:
  - Target instance ID
  - Target member ID
  - Destination instance ID
  - Destination member ID

and returns the following values:
- New instance ID
- New member ID
- Return code

In order to fully support the functionality of Sample SCM, we will use the CAF to customize our RAM API. We would need to create three new custom actions (for the lock instance, disenflaguate, and concatenate operations) and override two of the standard actions (lock and check out).

Assume for this example that we are developing the first version of SAMP RAM (version 1.0), that it is being designed to access Sample SCM version 1.4 and work with CARMA version 2.5, and that it will be written in C and compiled into a DLL named SAMPRAM. For this example we will assign SAMP RAM a RAM ID of 1.

Note: We will assume that SAMPRAM, the RAM's DLL, is stored in the common PDS that contains all of the RAMs available on the CARMA host. See "RAM Construction" on page 11 to learn where a RAM's DLL should be stored.

We now have all the information about the RAM needed for the SAM RAM model (see "RAM" on page 55). The following table summarizes this information:

Table 10. Information about SAMP RAM

Name	SAMP RAM
Description	Provides CARMA access to instances of Sample SCM
RAM ID	1
Programming Language	C
RAM DLL Name	SAMPRAM
Version	1.0
Repository Version	1.4

Table 10. Information about SAMP RAM (continued)

CARMA Version	2.5

At this time, you may find it helpful to tabulate the information (as described in "Action" on page 57) for all of the actions that need to be created or overridden. The following tables summarize this information. Note that the action ID for the lock action matches the action ID of the standard lock action (see Appendix B, "Action IDs," on page 105) in order to ensure that the original lock action is overridden. The disabled check out action is similarly assigned an ID corresponding to the standard check out action.

Table 11. Information about SAMP RAM's lock instance action

Name	Lock instance
Description	Locks an instance within the SCM
Action ID	100
RAM ID	1
Parameter List	Instance ID Reason
Return Value List	Return code Lock type

Table 12. Information about SAMP RAM's disenflaguate action

Name	Disenflaguate
Description	Removes a flag from a member within the SCM
Action ID	101
RAM ID	1
Parameter List	Instance ID Member ID Reason
Return Value List	Return code

Table 13. Information about SAMP RAM's concatenate action

Name	Concatenate
Description	Concatenates the contents of two members within the SCM
Action ID	102
RAM ID	1
Parameter List	Destination instance ID Destination member ID Target instance ID Target member ID
Return Value List	Return code New instance ID New member ID

Table 14. Information about SAMP RAM's lock action. Note that we do not provide a description for this action, since the description from the standard action is already available to the client. You may override the existing description by specifying a new one in the VSAM clusters, but the client may or may not use the updated description.

Name	Lock
Description	
Action ID	10
RAM ID	1
Parameter List	Instance ID Member ID
Return Value List	Return code Lock type

Table 15. Information about SAMP RAM's check out action. Since this action is disabled, we do not need to include a description, parameter list, or return value list.

Name	Check out
Description	(Disabled)
Action ID	13
RAM ID	1
Parameter List	(Disabled)
Return Value List	

Since the instance and member IDs are passed by default to all actions (see the description of "Parameter list" in "Action" on page 57), only three additional parameters need to be defined for the custom actions (lock instance, disenflaguate, and concatenate) and the lock action: reason, target instance ID, and target member ID. For the concatenate action, we can map destination instance ID and destination member ID respectively to the default parameters instance ID and member ID.

We can now list all of the parameters needed for the SAMP RAM model. The following tables summarize this information. Note that the parameters are assigned parameter IDs sequentially, starting with 0 for the first parameter.

Name	Reason
Description	Reason why the action should be performed
Parameter ID	0
RAM ID	1
Type	String
Length	30
Constant	No
Default Value	None
Prompt	Why are you requesting that the action be performed?

Name	Target instance ID

Description	ID of the instance containing the member whose contents should be appended to the end of the given member.
Parameter ID	1
RAM ID	1
Type	String
Length	15
Constant	No
Default Value	None
Prompt	Which instance contains the member that you want to concatenate with the selected member?

Name	Target member ID
Description	ID of the member whose contents should be appended to the end of the given member
Parameter ID	2
RAM ID	1
Type	String
Length	30
Constant	No
Default Value	None
Prompt	Which member's contents do you want to append to the end of the selected member?

Only three additional return values need to be defined for SAMP RAM, since the return code is already returned by default (see the description of "Return value list" in "Action" on page 57). The following tables summarize the return value information needed for our SAM RAM model. Again, note that the return values are assigned return value IDs sequentially, starting with 0 for the first return value.

Name	Lock type
Description	The lock type being applied to the member
Return Value ID	0
RAM ID	1
Type	Int
Length	4

Name	New instance ID
Description	The instance in which the action's results have been placed
Return Value ID	1
RAM ID	1
Type	String
Length	30

Name	New member ID
Description	The member containing the results of the action
Return Value ID	2
RAM ID	1
Type	String
Length	30

With all of the information necessary to define SAMP RAM to the CAF neatly tabulated, we can represent the information visually. Figure 10 illustrates the relationship between the actions, parameters, and return values used in SAMP RAM. Before setting up the clusters for a RAM, you may find it helpful to develop a similar diagram.
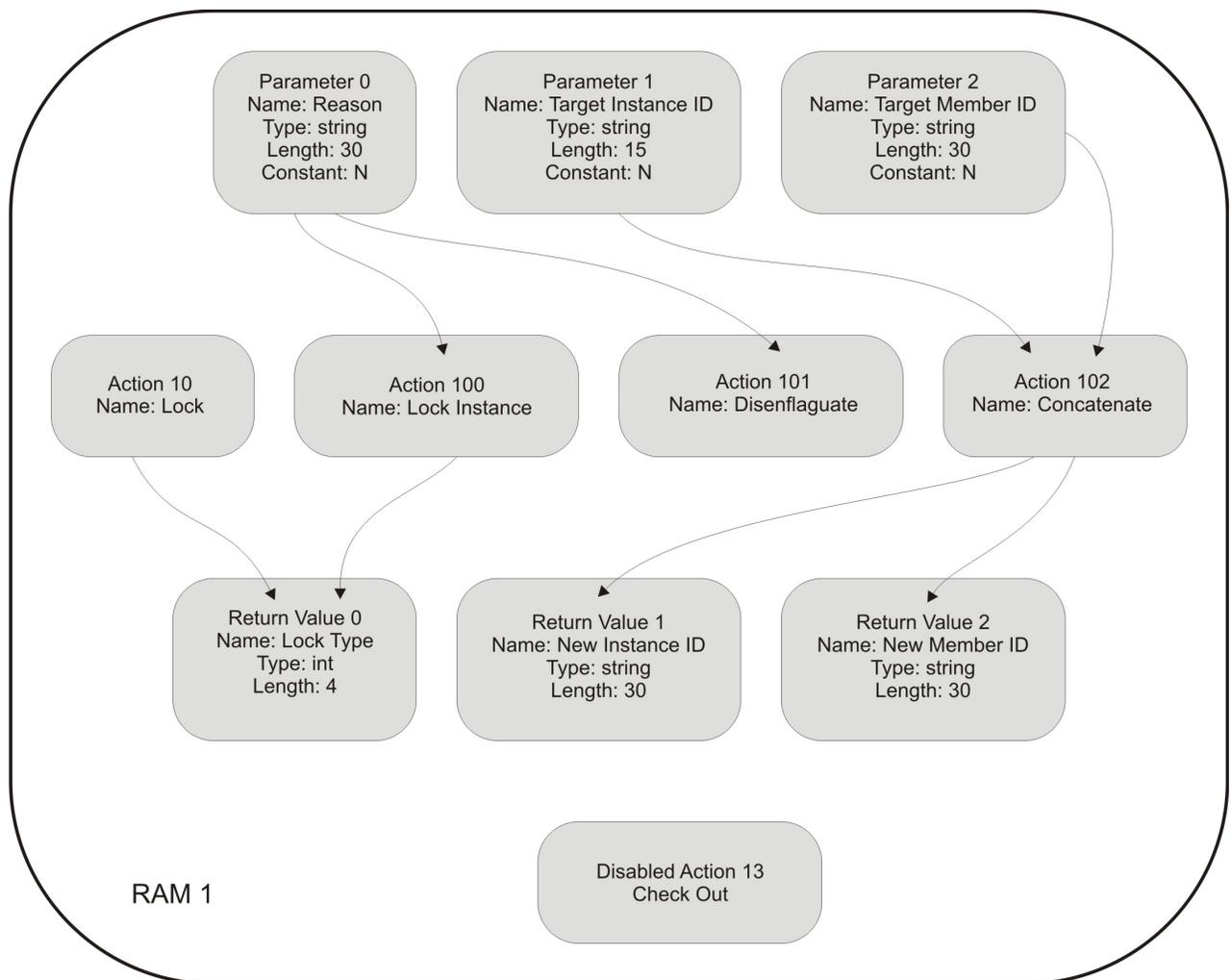

Figure 10. Visual representation of the SAMP RAM model. Only information relevant to the relationship between the objects is shown.

Creating VSAM records from a RAM model

Now that we have a model for the SAMP RAM, we can easily define SAMP RAM's CAF information. To do this, it is first necessary to understand where and how the CAF information is stored. There are two CAF key-sequenced VSAM clusters that store all of the CAF information: `CRADEF` and `CRASTRS`. As CARMA is loaded, it discovers the RAMs available to it (as well as their corresponding actions, parameters, and return values) by reading `CRADEF`, which contains information about the capabilities of the RAMs available. As necessary, CARMA tries to determine if a user's preferred language is available for a given RAM by checking `CRASTRS`, which contains locale-specific information for the RAMs.

CRADEF

`CRADEF` stores all the language-independent CAF data (data that does not need to be translated from one locale to another), using English characters from code page 00037. It contains records for each of the CAF object types (RAMs, actions, parameters, and return values), using a record width of 1032 bytes. However, only action records may actually make use of all 1032 bytes; the other record types simply fill the unused bytes with spaces. `CRADEF` uses an 8-byte key and reserves the remaining 1024 bytes for data. Table A summarizes the composition of a generic record in `CRADEF`:

Table 16. `CRADEF` record format

1032-Byte Record	
(8 bytes) Key	(1024 bytes) Data

Record keys

`CRADEF` record keys are composed of the following fields:

- (1 byte) The type character ("A" for action, "D" for disabled action, "P" for parameter, "R" for RAM, "T" for return value, "I" for identifier, and "F" for field)
- (2 bytes) The two-digit RAM ID left-padded with 0s (a unique identification number between "00" and "99")
- (3 bytes) The three-digit secondary ID left-padded with 0s. For all RAMs, this should be ″000″. For standard actions you should use the predefined action ID, and for custom actions you should use a custom action ID greater than or equal to ″100″. For parameters, return values and fields, you should use sequential IDs starting at ″000″.
- (2 bytes) Unused (reserved for future use). Fill these bytes with spaces.

The following table summarizes the `CRADEF` key format.

Table 17. `CRADEF` key format. The number of bytes reserved for each field is specified in parentheses. Fields marked as "Unused" should be filled entirely with spaces.

8-Byte Key			
(1 byte) Type	(2 bytes) RAM ID	(3 bytes) Secondary ID	(2 bytes) Unused

Record data

The rest of the bytes in each record are used for the record data. These 1024 bytes contain different fields depending on the record type:

RAM

1. (8 bytes) The version number of the RAM. This value may be displayed to users by CARMA clients.
2. (8 bytes) The programming language the RAM is written in. Select from the following list of valid values: "C", "COBOL", "PLI". (Alternatively, "PL1" may be used.)
3. (8 bytes) The version number of the repository that the RAM is compatible with. This value may be displayed to users by CARMA clients.
4. (8 bytes) The version number of CARMA that the RAM is compatible with. This value may be displayed to users by CARMA clients.
5. (8 bytes) The name of the RAM DLL

Action

> **Note:** The combined width of fields (1) and (3) below should be less than or equal to 1023.

1. (0 to 1023 bytes) A list of the parameter IDs used by the action. The IDs listed should be separated by commas. Do not use a trailing comma at the end of the list.
2. (1 byte) The pipe character, "|". This symbol is used to denote the separation between the parameter ID list and the return value ID list.

> **Note:** This character must be included even if either the parameter ID list or the return value ID list is empty. However, it should *not* be included if *both* the parameter ID list and return value ID list are empty.

3. (0 to 1023 bytes) A list of the return value IDs used by the action. The IDs listed should be separated by commas. Do not use a trailing comma at the end of the list.

Disabled action
(1024 bytes) Empty spaces. No data is required for disabled actions.

Parameter

1. (16 bytes) The data type of the parameter. Choose from the following available values: "INT", "LONG", "DOUBLE", "STRING".
2. (16 bytes) The length of the parameter. This is either a precision (for parameters of type "DOUBLE") or field width (for parameters of type "STRING"). Specify this value numerically (for example, as "12" instead of "twelve"). Use an arbitrary value if the parameter type is neither "DOUBLE" nor "STRING".
3. (1 byte) A "Y" or "N" to indicate whether this parameter does or does not (respectively) have a constant value.

Return value

1. (16 bytes) The data type of the return value. Choose from the following available values: "INT", "LONG", "DOUBLE", "STRING".
2. (16 bytes) The length of the return value. This is either a precision (for return values of type "DOUBLE") or field width (for return values of type "STRING"). Specify this value numerically (for example, as "12" instead of "twelve"). Use an arbitrary value if the return value type is neither "DOUBLE" nor "STRING".

Field   (64 bytes) The Metadata Key used to identify the metadata to be displayed.

Optionally, this can be left all spaces, then the Name value (taken from the CRASTRS VSAM) will be used for the Metadata Key.

Identifier

(1024 bytes) A unique RAM identifier used to identify the RAM. If a unique ID is provided, the RAM will be selected by its unique ID, and the ramId field will be ignored.

The following table summarizes the `CRADEF` data formats for each of the CAF object types.

Table 18. `CRADEF` data formats for each CAF object type. (The "Type" column lists the abbreviated type characters instead of the full type names.) The number of bytes reserved for each field is specified in parentheses (a "*" indicates a variable-length field). Fields marked as "Unused" should be filled entirely with spaces.

Type	1024-Byte Data				
R	(8 bytes) RAM Version	(8 bytes) Programming Language	(8 bytes) Repository Version	(8 bytes) CARMA Version	(8 bytes) DLL Name
A	(* bytes) Parameter ID List		(1 byte) List Separator Pipe	(* bytes) Return Value ID List	
D	(1024 bytes) Unused				
P	(16 bytes) Type		(16 bytes) Length		(1 byte) Constant
T	(16 bytes) Type		(16 bytes) Length		
F	(64 bytes) Metadata Key				
I	(1024 bytes) Identifier				

CRASTRS

`CRASTRS` stores all the language-dependent CAF data (data that needs to be translated from one locale to another, such as descriptions and messages). The languages are indexed within the VSAM cluster based on an eight-character locale (for example, "EN_US " or "FR_FR ") and a five-character code page (for example, "00037"). As a CARMA client initializes CARMA, the client provides CARMA a locale and code page, which CARMA attempts to locate in `CRASTRS`. If the specified locale and code page combination is not available in the CARMA environment, CARMA will use the default locale ("EN_US") and code page ("00037") and return an error to the client.

When a client request the list of available RAMs, CARMA will reference `CRASTRS` to attempt to compose a list of the RAMs that are available in the client's requested locale and code page. By convention, if a RAM record is available in a given locale, it is expected for its actions, parameters, and return values to also be available in that same locale.

`CRASTRS` allows strings of a non-fixed length. (Previously, `CRASTRS` used a record width of 2101 bytes composed of a 21-byte key and 2080 bytes for data.) Strings are separated by a single non-space character used for delimiting strings throughout the file. You can set the delimiting character when you configure the

CRASTRS file. If you do not set the delimiting character, the default is the horizontal tab character (0x05 in EBCDIC)

If CRASTRS contains a record with the key "000000000000000000000" (twenty-one zeros), CARMA uses the non-fixed string format in the entire VSAM file. If CRASTRS does not contain this record key, CARMA uses the fixed-width string format for the entire VSAM file.

Note: Longer strings are limited to the previous length limits if older function calls are used.

Note: Disabled actions do not need records in CRASTRS since they have no string to be translated.

Record keys

CRASTRS record keys are composed of the following fields:
- (8 bytes) The locale of the record (for example, "EN_US ")
- (5 bytes) The code page of the record (for example, "00037")
- (8 bytes) The key to the CRADEF record to which this CRASTRS record corresponds

The following table summarizes the CRASTRS key format.

Table 19. CRASTRS key format. The number of bytes reserved for each field is specified in parentheses.

21-Byte Key		
(8 byte) Locale	(5 bytes) Code Page	(8 bytes) Record Key

Record data

The rest of the record is used for the record data. The record data contains different fields depending on the record type In previous versions, this record data was limited to 2080 bytes. When CRASTRS contains a record with the key "000000000000000000000" (twenty-one zeros). This limit no longer applies.

RAM, action, and return type

1. The name of the CAF object this record corresponds to. (Previously limited to 16 bytes.)
2. A description of the CAF object this record corresponds to. (Previously limited to 1024 bytes.)

Parameter

1. The name of the parameter this record corresponds to. (Previously limited to 16 bytes.)
2. The default value of the parameter this record corresponds to. (Previously limited to 16 bytes.)
3. The prompt the client should display when requesting a value for the parameter this record corresponds to. (Previously limited to 1024 bytes.)
4. A description of the parameter this record corresponds to. (Previously limited to 1024 bytes.)

Field

1. The name corresponding to this metadata. To use this localized name as the Metadata Key, leave the Metadata Key blank in CRADEF. Leaving

this blank will cause the name to be the Metadata Key defined in CRADEF. (Previously limited to 128 bytes.)

2. The default value to display if no value is provided by the RAM for a given member for the Metadata Key. (Previously limited to 256 bytes.)

3. A description of the metadata shown in this field. (Previously limited to 1024 bytes.)

All information in the data section should be in the locale and code page specified in the key. The following table summarizes the CRASTRS data formats for each of the CAF object types.

Table 20. CRASTRS data formats for each CAF object type. (The "Type" column lists the abbreviated type characters instead of the full type names.) Note that the disabled action type has not been included in this table because CRASTRS should not have any records for disabled actions.

Type	Data			
A R T	Name		Description	
P	Name	Default Value	Prompt	Description
F	Name		Default Value	Description

SAMP RAM VSAM records

Building on our earlier SAMP RAM example, we can define records for SAMP RAM in CRADEF as shown in the following table.

Table 21. SAMP RAM records (one per row) in CRADEF. Each cell represents a field. Refer to "CRADEF" on page 64 to determine the widths for these fields.

Key			Data				
A	01	010	000		|		
A	01	100	000		|	000	
A	01	101			|	000	
A	01	102	001,002		|	001,002	
D	01	013					
P	01	000	STRING		30		N
P	01	001	STRING		15		N
P	01	002	STRING		30		N
R	01	000	1.0	C	1.4	2.5	SAMPRAM
T	01	000	INT		4		
T	01	001	STRING		30		
T	01	002	STRING		30		

Please refer to FEK.SFEKVSM2(CRAINIT) for an example of the proper column format. This sequential data set is used to initialize CRADEF during CARMA installation. Initially, it contained records for the sample PDS RAM, the sample SCLM RAM, and Skeleton RAM. However, depending on the configuration of your host, FEK.SFEKVSM2(CRAINIT) may have been modified if RAMs have been added or removed from your CARMA environment.

To add a RAM to the `CRADEF` cluster, you should add its records to `FEK.SFEKVSM2(CRAINIT)`. Ensure that all record keys are in alpha-numeric order so that the data set can be successfully REPROed. You should use the JCL script located at `FEK.#CUST.JCL(CRA$VDEF)` to REPRO `FEK.SFEKVSM2(CRAINIT)`.

Now we need to define the locale-specific records in `CRASTRS`. Assume that SAMP RAM needs support for English and Brazilian Portuguese. We can define records for SAMP RAM in `CRASTRS` as shown in the following table.

Table 22. SAMP RAM records (one per row) in `CRASTRS`. Each cell represents a field. Refer to "`CRASTRS`" on page 66 to determine the widths for these fields. Note that the records that have a key ending with `A01010` have no data. The data for these records are optional, since these records correspond to standard actions that have been overridden. CARMA will provide the client with the default name and description for these overridden standard actions.

Key			Data			
EN_US	00037	A01010				
EN_US	00037	A01100	Lock Instance		Locks the instance	
EN_US	00037	A01101	Disenflaguate		Removes a flag	
EN_US	00037	A01102	Concatenate		Concatenates two data sets	
EN_US	00037	P01000	Reason	Why not?	Why do you want me to perform the action?	The reason for performing the action
EN_US	00037	P01001	Target Instance ID	MyInstance	In which instance is the member located?	The instance containing the member to be concatenated
EN_US	00037	P01002	Target Member ID	MyMember	Which member would you like to concatenate?	The member to be concatenated
EN_US	00037	R01000	Sample RAM		An example RAM	
EN_US	00037	T01000	Lock Type		The type of lock the SCM put on the member	
EN_US	00037	T01001	New Instance ID		The concatenation's instance ID	
EN_US	00037	T01002	New Member ID		The concatenation's member ID	
PT_BR	01047	A01010				
PT_BR	01047	A01100	Bloquear Instƒncia		Bloqueia a instƒncia	
PT_BR	01047	A01101	Tirar sinalizador		Remove um sinalizador	
PT_BR	01047	A01102	Concatenar		Concatena dois conjuntos de dados	
PT_BR	01047	P01000	Motivo	Por que nÆo?	Por que você deseja que eu execute a a‡Æo?	O motivo para executar a a‡Æo
PT_BR	01047	P01001	ID de Instƒncia de Destino	MyInstance	Em qual instƒncia o membro está localizado?	A instƒncia que cont,m o membro a ser concatenado

Table 22. SAMP RAM records (one per row) in CRASTRS. *Each cell represents a field. Refer to "*CRASTRS*" on page 66 to determine the widths for these fields. Note that the records that have a key ending with* A01010 *have no data. The data for these records are optional, since these records correspond to standard actions that have been overridden. CARMA will provide the client with the default name and description for these overridden standard actions. (continued)*

Key			Data			
PT_BR	01047	P01002	ID do Membro de Destino	MyMember	Qual membro você deseja concatenar?	O membro a ser concatenado
PT_BR	01047	R01000	RAM de Amostra		Um RAM de exemplo	
PT_BR	01047	T01000	Tipo de Bloqueio		O tipo de bloqueio que SCM coloca no membro	
PT_BR	01047	T01001	Novo ID de Inst∫ncia		O ID de inst∫ncia de concatena‡Æo	
PT_BR	01047	T01002	Novo ID do Membro		O ID do membro de concatena‡Æo	

Please refer to FEK.SFEKVSM2(CRASINIT) for an example of the proper column format. This sequential data set is used to initialize CRASTRS during CARMA installation. Like FEK.SFEKVSM2(CRAINIT), initially, it contained the strings for the sample PDS RAM, the sample SCLM RAM, and Skeleton RAM. Depending on the configuration of your host, FEK.SFEKVSM2(CRASINIT) may also have been modified if RAMs have been added or removed from your CARMA environment.

To add a RAM to the CRASTRS cluster, you should add its records to FEK.SFEKVSM2(CRASINIT). Ensure that all record keys are in alpha-numeric order so that the data set can be successfully REPROed. You should use the JCL script located at FEK.#CUST.JCL(CRA$VSTR) to REPRO FEK.SFEKVSM2(CRASINIT).

VSAM cluster access

When editing VSAM clusters, ensure that no clients are accessing CARMA. CARMA may exhibit abnormal behavior if the VSAM cluster changes while it is operating. It is recommended that only system administrators and RAM developers have write access to the VSAM clusters, but that all users have read access.

Chapter 5. Developing a CARMA client

CARMA clients can be designed to work specifically with a RAM, can provide a generic interface for any RAM to use, or can do a combination of the two. A good example of a generic client that can also be modified to work specifically with certain RAMs is IBM Rational Developer for System z. Rational Developer was designed to support the basic functions all RAMs have in common, so a RAM fitting perfectly into the CARMA RAM API specification would work with Rational Developer right out of the box. Rational Developer also provides extension points with which RAM developers can customize the client for their RAM(s). On the other end of the spectrum, a very specific, non-interactive client could be written to simply run maintenance operations through a RAM.

CARMA clients can make use of some or all of the basic CARMA API functions. The only functions that are required to be implemented are `initCarma`, `initRAM`, and `terminateCarma`. `terminateRAM` is not required because `terminateCarma` will take care of cleaning up the RAMs if it is called and CARMA still has RAMs loaded. However, special care should be taken with the memory that is passed to and from CARMA. Often, the RAM will allocate memory that the client is required to free. Please read through "Storing results for later use" on page 72 and "Memory allocation" on page 6 carefully, as memory leaks and abnormal program termination can easily result from not following the recommendations on handling memory for each function.

Compiling the CARMA client

CARMA clients can include the CARMA DLL's side deck during compilation (causing the CARMA DLL to be loaded implicitly) or can be compiled without the side deck (causing the CARMA DLL to be loaded explicitly). The example client (`CRACLISA` in the sample library) implicitly loads the CARMA DLL. The JCL code to compile a client that will implicitly load the CARMA DLL is in the sample file named `CRACLICM`.

Running the client

When running a CARMA client, you must ensure that CARMA and all its RAMs have the resources they require available to them. CARMA requires access to its message VSAM cluster (`CRAMSG`), the CAF VSAM clusters (`CRADEF` and `CRASTRS`), and the PDS containing the RAMs. Browse the JCL used to run clients (`CRACLIRN`, located in the sample library) to see the `DD` statements CARMA requires (`CRASTRS`, `CRAMSG`, and `CRADEF`) and how the CARMA DLL and the PDS containing all RAMs are added to the `STEPLIB DD` statement. RAMs should document any resources they require. For example, the sample PDS RAM and sample SCLM RAM each require a message cluster to be available, so the JCL used to run the client should be modified so that the RAM can access these resources. Failure to provide CARMA or the RAMs with access to their required resources may result in abnormal behavior.

When providing resources to RAMs, the TSO/ISPF message libraries should also be considered. RAMs may use the TSO/ISPF messages if errors occur. By default, the JCL used to run a client will provide the RAMs with the English (00037 code page) version of these messages. The JCL should be edited appropriately if the RAM should return TSO/ISPF messages to the client in a different language.

Storing results for later use

The client should store the results for most operations executed during a CARMA session, especially the results from browsing functions such as `getMembers` and `getInstances`. All instances, simple members, and containers have both an ID and a display name. The display name is what the client should display to the user. The display name for an entity should be given in the context of that entity's instance and, if applicable, all parent containers needed to reach that entity. The ID defines the entity to the RAM uniquely. For example, the entity's ID could simply contain its absolute path. Alternatively, the RAM could use a hashing function to obtain the entity's absolute path from the ID. The ID should be stored by the client so that it can be passed back to the RAM as needed. For example, a user might obtain a list of members within an instance and then check to see if one of those members is a container.

The other pieces of data that might need to be stored by the client (if they are not already known) are metadata keys, RAM CAF information, and names. The RAM CAF information is required by virtually every function that uses a RAM to carry out an operation.The CAF information that is required may be as simple as the ID of the RAM the action should be run by.

Client predefined data structures

Most RAM functions use predefined structures to pass information back to CARMA and then the RAM. Some structures pass information about a RAM, while others are used for communication to and from the RAM. It is the responsibility of the Client to free memory used by these structures, including any character arrays of undefined length. These arrays will be null-terminated in the normal C style.

When running an action against CARMA, the client should see if the action's respective `Action` structure exists for the RAM being worked with. If so, it should then use the `Action` structure and related `Parameter` structures to call the action. After the action is complete, the client should use the `returnValue` structures related to the action called to properly parse the action's response.

The applicable structures are summarized in the following tables. These structures are available in the `CRADSDEF` header file located in the sample library. They are almost always allocated by the RAM, so it is unlikely that the client will ever have to initialize any of their buffers. However, the client will have to free any memory that is allocated by the RAM.

The `RAMRecord` and `RAMRecord2` structures consist of an integer RAM ID, a 16–byte name character field, and several other character fields that describe the RAM.

Table 23. `RAMRecord` and `RAMRecord2` data structures

Field		Description
RAMRecord	**RAMRecord2**	
int id	int id	Unique ID to describe the RAM
char name[16]	char* name	Display name
char version[8]	char* version	RAM version
char reposLevel[8]	char* reposLevel	The level of the SCM the RAM accesses.

Table 23. `RAMRecord` and `RAMRecord2` data structures (continued)

Field		Description
RAMRecord	**RAMRecord2**	
`char language[8]`	`char* language`	Language in which the RAM is written
`char CRALevel[8]`	`char* CRALevel`	The level of CARMA for which the RAM was designed
`char moduleName[8]`	`char* moduleName`	Name of the RAM module to load
`char description[2048]`	`char* description`	Displayed as a RAM description by the client
not applicable	`char* uniqueId`	Specifies a unique RAM ID

The `Descriptor` structure consists of a 64–byte `name` character field and a 256–byte `ID` character field. It is used to describe instances, containers, and simple members.

Table 24. `Descriptor` data structure

Field	Description
`char id[256]`	Unique ID to describe the entity
`char name[64]`	Display name

The `DescriptorWithInfo` structure consists of a 64–byte `name` character field and a 256–byte `ID` character field, an integer storing the number of key-value pairs, and an array of key-value pairs containing metadata information.

Table 25. `DescriptorWithInfo` data structure

Field	Description
`char id[256]`	Unique ID to describe the entity
`char name[64]`	Display name
`int infoCount`	The number of key-value pairs in the info array
`KeyValPair* info`	An array containing metadata information in the form of key-value pairs

The `MemberDescriptorWithInfo` structure consists of a 64–byte `name` character field and a 256–byte `ID` character field, an integer storing the number of key-value pairs, and an array of key-value pairs containing metadata information, and an integer indicating if the member is a container.

Table 26. `MemberDescriptorWithInfo` data structure

Field	Description
`char id[256]`	Unique ID to describe the entity.
`char name[64]`	Display name.
`int infoCount`	The number of key-value pairs in the info array.
`KeyValPair* info`	An array containing metadata information in the form of key-value pairs.

Table 26. `MemberDescriptorWithInfo` data structure (continued)

Field	Description
int isContainer	A value of 0 indicates that the member is not a container. A value of 1 indicates that it is a container.

The `KeyValPair` structure consists of a 64–byte key field and a 256–byte value field. It is used for metadata key-value pairs.

Table 27. `KeyValPair` data structure

Field	Description
char key[64]	An index
char value[256]	The data

The `Action` and `Action2` structures consist of an integer ID, a 16-byte name, a pointer to an integer array to store the IDs of the parameters related to the action, an integer storing the number of parameters associated with the action, a pointer to an integer array to store the IDs of the return values related to the action, an integer storing the number of return values associated with the action, and a 1024-byte description.

Table 28. `Action` and `Action2` data structures

Field		Description
Action	**Action2**	
int id	int id	A numeric identifier for the action between 0 and 999. Action IDs between 0 and 79 override standard actions, while IDs between 100 and 999 to define custom actions. Action IDs between 80 and 99 are reserved for use by CARMA.
char name[16]	char* name	The action's name.
int* paramArr	int* paramArr	A list of the IDs for the parameters the action uses.
int numParams	int numParams	The number of elements in the paramArr array.
int* returnArr	int* returnArr	A list of the IDs for the return values the action returns.
int numReturn	int numReturn	The number of elements in the returnArr array.
char description[1024]	char* description	A short description of the action.

The `Parameter` and `Parameter2` structures consist of an integer ID, a 16-byte name, a 16-byte type, a 16-byte default value, an integer length, an integer specifying whether it is constant (a value of 1 indicates that it is), a 1024-byte prompt, and a 1024-byte description.

Table 29. `Parameter` *and* `Parameter2` *data structures*

Field		Description
Parameter	**Parameter2**	
`int id`	`int id`	A numeric identifier for the parameter between 0 and 999.
`char name[16]`	`char* name`	The parameter's name.
`char type[16]`	`char* type`	The data type of the parameter ("INT", "LONG", "DOUBLE", or "STRING").
`char defaultValue[16]`	`char* defaultValue`	The parameter's default value.
`int length`	`int length`	The precision of the parameter (if it is of the "DOUBLE" type) or the field width of the parameter (if it is of the "STRING" type). If the parameter is of some other type, then this value can be ignored.
`int isConstant`	`int isConstant`	Whether or not the parameter will always contain the same value.
`char prompt[1024]`	`char* prompt`	The prompt that the CARMA client should display when requesting a value for the parameter from users.
`char description[1024]`	`char* description`	A short description of the parameter.

The `returnValue` and `returnValue2` structures consist of an integer ID, a 16-byte name, a 16-byte type, an integer length, and a 1024-byte description.

Table 30. `returnValue` *and* `returnValue2` *data structures*

Field		Description
returnValue	**returnValue2**	
`int id`	`int id`	A numeric identifier for the return value between 0 and 999.
`char name[16]`	`char* name`	The return value's name.
`char type[16]`	`char* type`	The data type of the return value ("INT", "LONG", "DOUBLE", or "STRING").
`int length`	`int length`	The precision of the return value (if it is of the "DOUBLE" type) or the field width of the return value (if it is of the "STRING" type). If the return value is of some other type, then this value can be ignored.

Table 30. `returnValue` and `returnValue2` data structures (continued)

Field		Description
returnValue	**returnValue2**	
`char description[1024]`	`char* description`	A short description of the return value.

The `Field` and `Field2` structures consist of an integer ID, a 64-byte member key, a 128-byte name, a 256-byte default value, and a 1024-byte description.

Table 31. `Field` and `Field2` data structures

Field		Description
Field	**Field2**	
`int id`	`int id`	A numeric identifier for the field value between 0 and 999.
`char memberKey[64]`	`char* memberKey`	The metadata key to provide to the getMemberInfo function for the field to be displayed.
`char name[128]`	`char* name`	The localized displayable name for the field.
`char defaultValue[256]`	`char* defaultValue`	The localized displayable value for the field if no value is returned by a call to getMemberInfo.
`char description[1024]`	`char* description`	A localized displayable description of the metadata.

Logging

CARMA and RAMs will write messages to a log per CARMA session. When initializing CARMA, a trace level should be passed to it. The trace levels are shown in Table 3 on page 9. Logging can be disabled by sending CARMA a trace level of -1.

Handling custom parameters and return values

Custom parameters are passed to the RAM using the `void** params` parameter. `params` is an array of `void` pointers that point to variables of several types. The `getCAFData` or `getCAFData2` function will return the Custom Action Framework information for all RAM functions. Call this before running any other RAM functions to determine what custom parameters and return values the RAM functions use. Required custom parameters must be passed to the RAM using the `params` parameter. If there are no required custom parameters, set `params` to `NULL`. To fill `params`, simply assign the `void` pointers in the array to each custom parameter. Use the following C code as an example:

```
int param0 = 5;
char* param1 = "HELLO";
double param2 = 4.3234;
void** params = (void**) malloc(sizeof(void*) * 3);
params[0] = (void*) &param0;
params[1] = (void*) param1; /*the char pointer should not be dereferenced*/
params[2] = (void*) &param2;
```

```
/*  Function call goes here....*/

free(params);
```

CARMA clients must pass a void*** parameter into all RAM functions defined to
return custom return values. You may simply pass a pointer to a void** variable
that you define. Once the custom return values have been returned, they can be
unpacked as demonstrated in the following C code. It is the responsibility of the
client to free the custom returns:

```
/* Declared at top */
int return0;
char return1[15];
void ** returnVals = NULL;

/* Call the CARMA function with &returnVals for custom returns */

/* Unpack the void** (returnVals) */
return0 = *((int*) returnVals[0]);
memcpy(return1, (char*) returnVals[1], 15);

/*Free each return, and the array*/
free(returnVals[0]);
free(returnVals[1]);
free(returnVals);
```

CARMA Defined Metadata

RAM specified file extension

When using a CARMA client, CARMA resources can automatically obtain
suggested extensions from the metadata property which is specified by the RAM.
This is done to eliminate the need to have the user set the extension on every
CARMA resource. In some cases however, an extension may not be specified by
the RAM requiring the client to provide a default extension. In instances such as
this, the client should be configured to ignore the file extension provided by the
RAM and instead, utilize an extension specified from within the client. Examples
of how the client can override a RAM specified file extension can be found in
"RAM specified file extension" on page 19.

Extract to External

CARMA provides clients with the ability to extract files from an SCM into a
normal host environment of PDSs and Sequential files.

copyFromExternal

Copies a member from a PDS or an SDS.

```
int copyFromExternal(int ramID, char instanceID[256], char memberID[256],
     char external[256], void** params, void*** customReturn, char error[256])
```

int ramID	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance containing the member being copied.
char memberID[256]	Input	The ID of the member being copied.

char external[256]	Input	The location to copy from. Either a PDS member or an SDS member. Examples: FEK.#CUST.EXT.STOR FEK.#CUST.EXT.PDS(MEMBER).
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

copyToExternal

Copies a member to a PDS or an SDS.

```
int copyToExternal(int ramID, char instanceID[256], char memberID[256],
        char target[256], void** params, void*** customReturn, char error[256])
```

int ramID	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance containing the member being copied.
char memberID[256]	Input	The ID of the member being copied.
char target[256]	Input	The location to copy to. Either a PDS member or an SDS member. Examples: FEK.#CUST.EXT.STOR FEK.#CUST.EXT.PDS(MEMBER).
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

State functions

CARMA expects certain functions to be run in order. These state functions and their expected order are:

1. initCARMA — CARMA initializes several global variables; the session log, and the locale to be used for the session with this function. This function should not be called a second time unless a terminateCarma call is made first.

2. getRAMList or getRAMList2 — This should be called before loading any RAMs, but clients may cache the RAM list and ignore this function if desired. However, there is little performance benefit in doing this, because CARMA will run the function as it needs the list itself.

3. initRAM — This must be called for each RAM before attempting to run any of that RAM's functions. Once this is run, CARMA will keep a pointer to the RAM until termination. RAMs should not be re-initialized without first terminating them.

4. `reset` — This may be called if the user wants to reload the SCM environment because a change has occurred. It will tell the RAM to restore itself to its initial state.

5. `terminateRAM` — This function does not have to be called. Each loaded RAM's `terminateRAM` function will be called by `terminateCarma` if `terminateCarma` is called first. Once `terminateRAM` is called, each RAM must be re-initialized using the `initRAM` function before any other function can be called for that RAM.

6. `terminateCarma` — This should always be called when exiting the CARMA session. It will handle cleaning up all of the RAMs that are currently loaded. Once this is called, `initCarma` must be run again before attempting to call any other CARMA function.

initCarma

Will set up the CARMA environment, session log, and session locale

`int initCarma(int traceLev, char locale[5], char error[256])`

int traceLev	Input	The trace level for the current session. See "Logging" on page 76 for more information.
char locale[5]	Input	Five character, non-null terminated buffer containing the locale for which all displayable strings should be set.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

If this function is not called, a default locale of "EN_US" and a default trace level of 0 will be used.

getRAMList

Retrieves the list of available RAMs from CARMA.

`int getRAMList(RAMRecord** records, int *numRecords, char error[256])`

RAMRecord** records	Output	Will contain an array of RAMRecord data structures to be used for display information about the RAMs and accessing them with other functions.
int* numRecords	Output	The number of RAMRecord data structures contained in the records array.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

The list of RAMs that is returned is dependent on the locale that was passed into `initializeCarma`. All RAMs stored within the CARMA environment that have display strings for the specified client locale will be returned.

getRAMList2

Retrieves the list of available RAMs from CARMA

`int getRAMList2(RAMRecord2** records, int *numRecords, char error[256])`

RAMRecord2** records	Output	Will contain an array of `RAMRecord2` data structures to be used for display information about the RAMs and accessing them with other functions.
int* numRecords	Output	The number of `RAMRecord2` data structures contained in the `records` array.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

The list of RAMs that is returned is dependent on the locale that was passed into `initializeCarma`. All RAMs stored within the CARMA environment that have display strings for the specified client locale will be returned.

initRAM

Initializes a RAM. CARMA will store a pointer to the RAM for quick future access.

`int initRAM(int RAMid, char locale[8], char codepage[5], char error[256])`

int RAMid	Input	Tells CARMA which RAM should be initialized. This ID was obtained after running `getRAMList` or `getRAMList2`.
char locale[8]	Input	Tells CARMA the locale of the strings that should be returned to the client.
char codepage[5]	Input	Tells CARMA the code page of the strings that should be returned to the client.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

reset

Tells the RAM to reset itself to its initial state

`int reset(int RAMid, char error[256])`

int RAMid	Input	Tells CARMA which RAM should be reset. This ID was obtained after running `getRAMList` or `getRAMList2`.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

terminateRAM

Tells the RAM to clean up its environment. CARMA will release the RAM module.

`int terminateRAM(int RAMid, char error[256])`

int RAMid	Input	Tells CARMA which RAM should be terminated. This ID was obtained after running `getRAMList` or `getRAMList2`.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

terminateCarma

Will clean up the CARMA environment, including the environments of any loaded RAMs

`int terminateCarma(char error[256])`

char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Browsing functions

getInstances

Retrieves the list of instances available in the SCM

```
int getInstances(int RAMid, Descriptor** RIrecords,int* numRecords,
                 void** params, void*** customReturn, char filter[256],
                 char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList` or `getRAMList2`.
Descriptor** RIrecords	Output	This will be allocated and filled with the IDs and names of instances.
int* numRecords	Output	The number of records that have been allocated and returned
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).

char filter[256]	Input	This can be passed from the client to filter out sets of instances.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Note: Be sure to free the RIrecords array.

getInstancesWithInfo

Retrieves the list of instances available in the SCM and the metadata associated with those instances. This is an optional function. The client should call this function if it wants to retrieve a list of instances and the metadata associated with all of those instances. If the RAM does not support this function, the client should fall back to calling getInstances.

```
int getInstancesWithInfo(int RAMid, DescriptorWithInfo** records,
                int* numRecords, void** params, void*** customReturn,
                char filter[256], char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
DescriptorWithInfo** records	Output	This will be allocated and filled with the IDs and names of instances. It will also contain the metadata information for each instance.
int* numRecords	Output	The number of records that have been allocated and returned.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char filter[256]	Input	This can be passed from the client to filter out sets of instances.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Note: Be sure to free the metadata information array for each record returned. After freeing the individual metadata information, be sure to free the records array.

getMembers

Retrieves the list of members available within the specified instance.

```
int getMembers(int RAMid, char instanceID[256],
            Descriptor** memberArr, int* numRecords, void** params,
            void*** customReturn, char filter[256], char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance for which the members should be retrieved.
Descriptor** memberArr	Output	This will be allocated and filled with the IDs and names of instances.
int* numRecords	Output	The number of records that have been allocated and returned in the array.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char filter[256]	Input	This can be passed from the client to filter out sets of members.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Note: Be sure to free the memberArr array.

getMembersWithInfo

Retrieves the list of members available within the specified instance.

```
int getMembersWithInfo(int RAMid, char instanceID[256],
            MemberDescriptorWithInfo** members, int* numRecords,
            void** params, void*** customReturn, char filter[256],
            char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance for which the members should be retrieved.

MemberDescriptorWithInfo** members	Output	This will be allocated and filled with the IDs and names of members. It will also contain the metadata information for each member and if that member is a container.
int* numRecords	Output	The number of records that have been allocated and returned in the array.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char filter[256]	Input	This can be passed from the client to filter out sets of members.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Note: Be sure to free the metadata information array for each record returned. After freeing the individual metadata information, be sure to free the members array.

isMemberContainer

Sets isContainer to true if the member is a container; false if not.

```
int isMemberContainer(int RAMid, char instanceID[256],
                char memberID[256], int* isContainer,
                void** params, void*** customReturn,
                char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance the member is within.
char memberID[256]	Input	The member that may be a container.
int* isContainer	Output	Set this to 1 if the member is a container; 0 if not.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).

void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

getContainerContents

Retrieves the list of members within a container.

```
int getContainerContents(int RAMid, char instanceID[256],
                         char memberID[256], Descriptor** contents,
                         int* numMembers, void** params,
                         void*** customReturn, char filter[256],
                         char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance the member is within.
char memberID[256]	Input	The container for which the members are being retrieved.
Descriptor** contents	Output	This will be allocated and filled with the IDs and names of the members within the container.
int* numRecords	Output	The number of member records that have been allocated and returned in the array.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char filter[256]	Input	This can be passed from the client to filter out sets of members.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Note: Be sure to free the contents array.

getContainerContentsWithInfo

Retrieves the list of members available within the specified container and the metadata associated with those members. This is an optional function. The client should call this function if it wants to retrieve a list of members and the metadata associated with all of those members. If the RAM does not support this function, the client should fall back to calling getContainerContents.

```
int getContainerContentsWithInfo(int RAMid, char instanceID[256],
                 char memberID[256], MemberDescriptorWithInfo** members,
                 int* numRecords, void** params, void*** customReturn,
                 char filter[256], char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance the member is within.
char memberID[256]	Input	The container for which the members are being retrieved.
MemberDescriptorWithInfo** members	Output	This will be allocated and filled with the IDs and names of members. It will also contain the metadata information for each member and if that member is a container.
int* numRecords	Output	The number of member records that have been allocated and returned in the array.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char filter[256]	Input	This can be passed from the client to filter out sets of members.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Note: Be sure to free the metadata information array for each record returned. After freeing the individual metadata information, be sure to free the members array.

Create/delete

Create and delete provides functionality to create and delete both members and containers within a CARMA environment.

createMember

Creates a new member.

```
int createMember(int RAMid, char instanceID[256], char memberID[256], char name[64],
          char parentID[256], int* lrecl, char recFM[4], void** params,
          void*** customReturn, char error[256]);
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList` or `getRAMList2`.
char instanceID[256]	Input	The instance containing the member being created.
char memberID[256]	Output	The ID of the member that is being created.
char name[64]	Input/Output	ID of the member being created.
char parentID[256]	Input	ID of parent container. (If no parent exists, space must be filled.)
int* lrecl	Output	The number of columns in the data set and array.
char recFM[4]	Output	Contains the data set'record format (FB, VB, etc.).
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

To account for specific RAM naming conventions, a client calls create by requesting a certain name. The RAM can then provide a unique `memberID`, `lrecl`, `recFM` and an appropriate displayable name back to the client.

If the client requests the name "bob," for example, the RAM might return a `memberID` of "BOB" as well as a displayable name of "BOB". If the member "bob" already exists, it might return "BOB2", or instead return an error saying it can not create the requested member.

`parentID` is the `memberID` for the parent of the member being created. If the member being created does not have a parent (it is directly under the repository instance), `parentID` should be left blank (all spaces).

A RAM does not have to create a member when createMember is called, but can just provide the proper `memberID`, `lrecl`, `recFM`, and displayable name to the client. It is the client's responsibility to make a call to `putMember` with the new `memberID` in

order to create a concrete member. RAMs should support adding a member with
no records (even if they have to create a single blank record for the member).

createContainer

Creates a new container.

```
int createContainer(int RAMid, char instanceID[256], char memberID[256],
    char name[64], char parentID[256], void** params, void*** customReturn,
        char error[256]);
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance containing the container being created.
char memberID[256]	Output	The ID of the member that is being created.
char name[64]	Input/Output	ID of the container being created.
char parentID[256]	Input	ID of parent container. (If no parent exists, space must be filled.)
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

To account for specific RAM naming conventions, a client calls create by requesting
a certain name. The RAM can then provide a unique memberID, lrecl, recFM and an
appropriate displayable name back to the client.

If the client requests the name "bob," for example, the RAM might return a
memberID of "BOB" as well as a displayable name of "BOB". If the container "bob"
already exists, it might return "BOB2", or instead return an error saying it can not
create the requested container.

parentID is the memberID for the parent of the container being created. If the
container being created does not have a parent (it is directly under the repository
instance), parentID should be left blank (all spaces).

Unlike the createMember function, when createContainer is called, the container
should always be created immediately by the RAM, unless an error occurs.

delete

Deletes a member or container.

```
int delete(int RAMid, char instanceID[256], char memberID[256], int force,
   void** params, void*** customReturn, char error[256]);
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList` or `getRAMList2`.
char instanceID[256]	Input	The instance containing the member or container being deleted.
char memberID[256]	Input	The ID of the member that is being deleted.
int force	Input	Used to force a delete. A value of 1 will force a delete.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

The force parameter can be set to 1 to tell a RAM to delete a member it normally would not, such as a non-empty container. If a RAM can delete an item, but it requires a force parameter to do so, it can send a certain return code, along with an appropriate error, to inform the client. The client can then offer a user the option of deleting with the force parameter.

Optionally, the client could also allow the user to set the force parameter before calling delete.

The delete function may be used to delete both members and containers, however, it should not be used to delete a RAM instance.

File transfer functions

extractMember

Retrieves a member's contents.

```
int extractMember(int RAMid, char instanceID[256],
                  char memberID[256], char*** contents, int* lrecl,
                  int* numRecords, char recFM[4], int* moreData,
                  int* nextRec, void** params, void*** customReturn,
                  char error[256])
```

int RAMid	Input		Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input		The instance containing the member.
char memberID[256]	Input		The ID of the member being extracted.
char*** contents	Output		Will be allocated as two-dimensional array to contain the member's contents.
int* lrecl	Output		The number of columns in the data set and array.
int* numRecords	Output		The number of records in the data set or the number of rows in the array.
char recFM[4]	Output		Will contain the data set's record format (FB, VB, etc.).
int* moreData	Output		Set the value of the variable to which this points as 1 if extract should be called again (because there is still more data to be extracted). Otherwise, assign the value to which it points as 0.
int* nextRec	Input/Output		**Input:** The member record where the RAM should begin the extraction. **Output:** The first record in the data set that was not extracted if *moreData is set to 1; otherwise, undefined.
void** params	Input		Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void** customReturn	Output		Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output		If an error occurs, this should be filled with a description of the error.

The contents buffer is a two-dimensional character array that will be filled by the RAM and returned to the client. For the first extractMember call, nextRec must be 0. The RAM may choose to return the data in chunks of records. Extract should be called until moreData is 0. If moreData is 1, extractMember needs to be called again, and the extraction from the member will start with the record indexed by the value

of `nextRec` returned on the previous call. The RAM will need the client to pass that value of `nextRec` back in for the following call.

See Chapter 3, "Developing a RAM," on page 11 for an example of `extractMember`'s operation from the RAM's point of view.

Note: Be sure to free `contents` properly. It has been allocated as a large contiguous data chunk, so it should be freed in the following manner. (The example is in C.)

```
for(i = 0; i < numRecords; i++)
    free(contents[i]);
free(contents);
```

putMember

Updates a member's contents or creates a new member if the specified `memberID` does not exist within the instance.

```
int putMember(int RAMid, char instanceID[256],
              char memberID[256], char** contents, int lrecl,
              int* numRecords, char recFM[4], int moreData,
              int nextRec, int eof, void** params, void*** customReturn,
              char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList` or `getRAMList2`.
char instanceID[256]	Input	The instance containing the member.
char memberID[256]	Input	The ID of the member being updated/created.
char** contents	Input	Contains the new member contents.
int lrecl	Input	The number of columns in the data set and array.
int* numRecords	Input/Output	The number of records in the data set or the number of rows in the array.
char recFM[4]	Input	Contains the data set's record format (FB, VB, etc.).
int moreData	Input	Will be 1 if the client has more chunks of data to send; 0 otherwise.
int nextRec	Input	The record in the data set to which the 0th record of the contents array maps.
int eof	Input	If 1, denotes that the last row of the array should mark the last row in the data set; 0 otherwise.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).

void** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

The client may choose a chunk size for the function or attempt to pass the whole file's contents at once. The client may also choose to jump around within a file. For example, records 0 through 15 could be passed first, 40 through 50 next, and then 16 through 39. However, not all RAMs may handle non-sequential data chunks such as this properly.

If sending data in chunks, `moreData` should be 1 on every call until the final one, during which it should be 0. `nextRec` should always be set to the first record to be updated in the member. Remember that this uses a 0-based index. `eof` is used to specify that the member record at `nextRec + numRecords` should be the last one in the updated member. For example, if that sum is 15 and there are currently 30 records in the member, records 16 through 29 will be deleted by the RAM after it updates through record 15.

See the source for the sample client (`CRACLISA` in the sample library) for more help.

Note: The contents buffer should be allocated before the call in a manner similar to the following. (The example is in C.)

```
contents = (char**) malloc(sizeof(char*) * (numRecords));
*contents = (char*) malloc(sizeof(char) * (lrecl) * (numRecords));
for(i = 0; i < numRecords; i++)
    (contents)[i] = ((*contents) + (i * (lrecl)));
```

It should also be freed after the call in a manner similar to the following. (The example is in C.)

```
free(contents[0])
free(contents);
```

Binary file transfer

extractBinMember
Retrieves a member's contents.

```
int putBinMember(int RAMid, char instanceID [256], char memberID [256],
            char** contents, int* length, int* moreData, int start,
            void** params, void*** customReturn, char error [256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList` or `getRAMList2`.
char instanceID[256]	Input	The instance containing the member being extracted.
char memberID[256]	Input	The ID of the member that is being extracted.

char** contents	Output	Pointer to the member's contents.
int* length	Output	The length of the member's contents.
int* moreData	Output	If extract should be called again because there is more data, set the value of the variable to which this points to 1; otherwise assign the value to which it points to 0.
int start	Input	The byte location of the file to start extracting from.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

putBinMember

Updates a member's contents or creates a new member if the specified memberID does not exist within the instance.

```
int putBinMember(int RAMid, char instanceID [256], char memberID [256],
         char* contents, int length, int moreData, int start,
         void** params, void*** customReturn, char error [256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList` or `getRAMList2`.
char instanceID[256]	Input	The instance containing the member being updated/created.
char memberID[256]	Input	The ID of the member that is being updated/created.
char* contents	Input	Contains the new members contents.
int length	Input	Pointer to the length of data to be written.
int moreData	Input	Will be 1 if the client has more chunks of data to send; 0 otherwise.
int start	Input	The byte location of the file to start putting data.

void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Metadata functions

getAllMemberInfo

Retrieves all metadata for the given member.

```
int getAllMemberInfo(int RAMid, char instanceID[256],
                     char memberID[256], KeyValPair** metadata,
                     int* num, void** params, void*** customReturn,
                     char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance the member is within.
char memberID[256]	Input	The ID of the member for which metadata is being returned. The ID may be empty if member info is to be retrieved for the instance instead of a specific member.
KeyValPair** metadata	Output	This will be allocated and filled with the keys and values of the metadata.
int* num	Output	The number of metadata KeyValPair structs allocated and returned in the array.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Note: Be sure to free the metadata array.

getFieldsData

Retrieves the fields data for the given RAM. The fields provide suggestions for metadata that should be displayed to the user.

```
int getFieldsData(int RAMid, Field** fields, int * numFields, char error[256])
```

int RAMid	Input	Tells CARMA which RAM to gather fields for. This ID was obtained after running `getRAMList`.
Field** fields	Output	This will be allocated and filled with the id, metadata key, name, default value, and description for each field.
int * numFields	Output	The number of Field structs allocated and filled in the array.
char error[256]	Output	In an error occurs this should be filled with a description of the error.

Note: Be sure to free the fields array.

getFieldsData2

Retrieves the fields data for the given RAM. The fields provide suggestions for metadata that should be displayed to the user.

```
int getFieldsData2(int RAMid, Field2** fields, int * numFields, char error[256])
```

int RAMid	Input	Tells CARMA which RAM to gather fields for. This ID was obtained after running `getRAMList2`.
Field2** fields	Output	This will be allocated and filled with the id, metadata key, name, default value, and description for each field.
int * numFields	Output	The number of Field structs allocated and filled in the array.
char error[256]	Output	In an error occurs this should be filled with a description of the error.

Note: Be sure to free the fields array.

getMemberInfo

Retrieves a specific piece of metadata for the given member.

```
int getMemberInfo(int RAMid, char instanceID[256],
                  char memberID[256], char key[64], char value[256],
                  void** params, void*** customReturn, char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance the member is within.
char memberID[256]	Input	The member for which metadata is being retrieved.
char key[64]	Input	The key of the metadata value to be retrieved.
char value[256]	Output	The value being retrieved.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

updateMemberInfo

Updates a specific piece of metadata for the given member.

```
int updateMemberInfo(int RAMid, char instanceID[256],
                     char memberID[256], char key[64], char value[256],
                     void** params, void*** customReturn,
                     char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance the member is within.
char memberID[256]	Input	The member for which metadata is being set.
char key[64]	Input	The key of the metadata value to be set.
char value[256]	Input	The value being set.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).

void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Other operations

lock

Locks the member.

```
int lock(int RAMid, char instanceID[256], char memberID[256],
         void** params, void*** customReturn, char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance the member is within.
char memberID[256]	Input	The member to be locked.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

unlock

Unlocks the member

```
int unlock(int RAMid, char instanceID[256], char memberID[256],
         void** params, void*** customReturn, char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance the member is within.
char memberID[256]	Input	The member to be unlocked.

void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

checkin

Check in the member. This only sets a flag. A `putMember` call is expected immediately after this call.

```
int checkin(int RAMid, char instanceID[256], char memberID[256],
            void** params, void*** customReturn, char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
char instanceID[256]	Input	The instance the member is within.
char memberID[256]	Input	The member to be checked in.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

checkout

Check out the member. This only sets a flag. A `extractMember` call is expected immediately after this call.

```
int checkout(int RAMid, char instanceID[256], char memberID[256],
             void** params, void*** customReturn, char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.

char instanceID[256]	Input	The instance the member is within.
char memberID[256]	Input	The member to be checked out.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

performAction

Instructs the specified RAM to perform the action identified in the actionID by using the parameters given and the return values in customReturn (when applicable).

```
int performAction(int RAMid, int actionID, char instanceID[256], char memberID[256],
            void** params, void*** customReturn, char error[256])
```

int RAMid	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running getRAMList or getRAMList2.
int actionID	Input	The custom action that is being requested, as defined in the CRADEF VSAM.
char instanceID[256]	Input	The instance the action is being performed on. The ID may be blank if the action is expected to be performed on the RAM itself instead of a specific instance or member.
char memberID[256]	Input	The member the action is being performed on. The ID may be blank if the action is expected to be performed on an instance or on the RAM itself instead of a specific member.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 76).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 76).

char error[256]	Output	If an error occurs, this should be filled with a description of the error.

getCAFData

Retrieves the CAF data for the requested RAM.

```
int getCAFData(int RAMid, Action** actions, int* numActions,
               int** disabledActions, int* numDisabled,
               Parameter** params, int* numParams,
               returnValue** returnVals, int* numReturn,
               char error[256])
```

Table 32. getCAFData parameters

int RAMid	Input	Tells CARMA for which RAM the CAF data should be pulled. This ID was obtained after running `getRAMList`.
Action** actions	Output	This will be allocated and filled with the custom actions for the given RAM.
int* numActions	Output	The number of actions being returned.
int** disabledActions	Output	This will be allocated and filled with the disabled actions for the given RAM.
int* numDisabled	Output	The number of disabled actions being returned.
Parameter** params	Output	This will be allocated and filled with the custom parameters for the given RAM.
int* numParams	Output	The number of parameters being returned.
returnValue** returnVals	Output	This will be allocated and filled with the custom return values for the given RAM.
int* numReturn	Output	The number of return values being returned.
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

See Chapter 4, "Customizing a RAM API using the CAF," on page 55 for more information on the types of data that may be returned. The data that is returned should be stored for the remainder of the session so that it can be checked before any function call for the respective RAM.

getCAFData2

Retrieves the CAF data for the requested RAM

```
int getCAFData2(int RAMid, Action2** actions, int* numActions,
                int** disabledActions, int* numDisabled,
```

```
                Parameter2** params, int* numParams,
                returnValue2** returnVals, int* numReturn,
                char error[256])
```

Table 33. getCAFData2 parameters

int RAMid	Input	Tells CARMA for which RAM the CAF data should be pulled. This ID was obtained after running `getRAMList2`.
`Action2** actions`	Output	This will be allocated and filled with the custom actions for the given RAM.
`int* numActions`	Output	The number of actions being returned
`int** disabledActions`	Output	This will be allocated and filled with the disabled actions for the given RAM.
`int* numDisabled`	Output	The number of disabled actions being returned.
`Parameter2** params`	Output	This will be allocated and filled with the custom parameters for the given RAM.
`int* numParams`	Output	The number of parameters being returned.
`returnValue2** returnVals`	Output	This will be allocated and filled with the custom return values for the given RAM.
`int* numReturn`	Output	The number of return values being returned.
`char error[256]`	Output	If an error occurs, this should be filled with a description of the error.

See Chapter 4, "Customizing a RAM API using the CAF," on page 55 for more information on the types of data that may be returned. The data that is returned should be stored for the remainder of the session so that it can be checked before any function call for the respective RAM.

getVersionList

Provides a list of versions available for a given member.

```
int getVersionList(char instanceID[256], char memberID[256],
        VersionIdent** versions, int* num, void** params,
            void*** customReturn, char error[256])
```

Table 34. getVersionList parameters

int RAMid	Input	Tells CARMA for which RAM the CAF data should be pulled. This ID was obtained after running `getRAMList` or `getRAMList2`.

char instanceID[256]	Input	The instance the member is within.
char memberID[256]	Input	The member to get a list of versions for.
VersionIdent** versions	Output	A list of all versions of the member available. This should be an ordered list, with the 'newest' version first, and the oldest version last.
int* num	Output	The number of versions.
void** params	Input	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 18).
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 18).
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

`VersionIdent` will be identified by the following struct:

```
typedef struct {
char memberID[256];  /*A versioned memberID, such as
     baseMemberID_VerNum*/
char versionKey[64]; /* A way to refer to the version, such as
                            "1.2.3"...should be the same as the value
                             for the carma.version metadata key*/
   char comments[256];  /* RAM supplied comments on the version,
                             could be timestamp, changes, etc..  */
} VersionIdent;
```

The version list will be a complete ordered version list, but the RAM Developer can chose to use a 'versioned' ID for the current version, or to use the unchanging ID. As an example, current version of a member might be accessible via "location(Member)" or "location(Member)_1.4" where the file is on version 1.4. The RAM developer could therefore choose to return either "location(Member)_1.4" or "location(Member)" as the newest version in the list.

When returning a list of members through browsing functions, such as `getMembers`, RAMs *should not* include the version in the `memberID`. Changing the `memberID` for a member prevents CARMA clients from properly tracking that member.

In order to support versioning, RAM Developers should handle CARMA calls when presented with a 'versioned' ID for the `memberID`.

Clients should support the return code of 130, which stands for "Member does not support versioning".

Clients can support a variety of calls against versioned members, such as the file transfer functions and the metadata functions.

Appendix A. Return codes

Return Code	Description
20	Internal error.
22	No RAMs defined for this locale.
24	CRADEF could not be opened for reading.
25	CRASTRS could not be opened for reading .
26	No records found in CRADEF.
28	CRADEF read error.
30	(placeholder)
32	Invalid CRADEF record found.
34	Requested RAM not found.
36	Could not load RAM module.
38	Could not load pointer to RAM function.
40	Requested RAM *RAM name* has not been loaded.
42	Invalid CRASTRS record found.
44	CARMA has not been initialized.
46	Failed attempting to load the RAM list.
48	Out of memory.
50	Record in CRADEF does not have equivalent in CRASTRS for this locale.
52	Action references unknown parameter.
54	Action references unknown return type.
56	CRASTRS read error.
58	Neither the specified locale or the default locale (EN_US, codepage 00037) could be found in CRASTRS.
60	CRAMSG not found.
62	CRAMSG read error.
64	CRADEF error: Action 16 can not have custom parameters and/or returns.
66	Invalid type specified in VSAM record.
68	Invalid default value in VSAM record.
101	Could not allocate memory.
102	TSO/ISPF Library functions not available.
103	Invalid member identifier.
104	Cannot allocate (out of space).
105	Member not found.
106	Instance not found.
107	Function not supported.
108	Member is not a container.

Return Code	Description
109	Invalid parameter value.
110	Member cannot be updated.
111	Member cannot be created.
112	Not authorized.
113	Could not initialize.
114	Could not terminate.
115	Resource out of sync.
116	File locked.
117	Specified next record out of range.
118	Unsupported record format.
119	Invalid LRECL.
120	Invalid metadata key.
121	Cannot update property value.
122	Invalid metadata value.
123	Property value is read-only.
124	Requested member is empty.
125	Empty instance.
126	No members found.
127	Reset error.
128	Delete error.
129	Member/Version is readOnly.
130	Member does not support versioning.
197	(Encapsulated ISPF/LMF error message.)
198	Unable to access log file.
199	Unknown RAM error.
222	Error retrieving Custom Action Framework parameter list.
223	Missing an expected Custom Action Framework parameter.
224	Unknown data type specified for Custom Action Framework parameter.
225	Error retrieving Custom Action Framework return values.

Appendix B. Action IDs

Action ID	Action Name
0	initRam
1	terminateRam
2	getMembers
3	extractMember
4	putMember
5	getAllMemberInfo
6	getMemberInfo
7	updateMemberInfo
8	isMemberContainer
9	getContainerContents
10	lock
11	unlock
12	checkIn
13	checkOut
14	getInstances
15	reset
16	performAction
17	createMember
18	createContainer
19	delete
20	copyToExternal
21	copyFromExternal
22	putBinMember
23	extractBinMember
24	getVersionList
25	getMembersWithInfo *
26	getContainerContentsWithInfo *
27	getInstancesWithInfo *
80	initCarma
81	terminateCarma
82	getRAMList
83	getCAFData

Note: * If the "WithInfo" functions require custom parameters or custom returns, set up the function to use the same custom parameters and custom returns as the base functions. This requirement allows the CARMA client to call the appropriate function with no additional input from the user.

Appendix C. Sample RAMs

This appendix functions as a resource for the sample RAMs that are shipped with the Common Access Repository Manager (CARMA). The sample RAMs are provided for the purpose of testing the configuration of your CARMA environment and as examples for developing your own RAMs.

Note: Do *not* use the provided sample RAMs in a production environment.

PDS RAM

RAM Description

The Partitioned Data Set (PDS) RAM allows you to access a PDS associated with TSO users by leveraging ISPF services. In this case, the TSO/ISPF services are the SCM and the repository is the user's PDS.

Navigation Structure

Within the PDS RAM, CARMA displays a list of all the PDS data sets that are available to you on a particular connection. Each PDS can then be expanded to display a collection of Sequential Data Sets (SDS), also called members which make up each PDS.

Supported actions

The following actions are currently only available for files with a record format of "Fixed Block".

- Extract
- Upload Local File
- Replace Local File

Unsupported actions

The following CARMA actions are unsupported by the PDS RAM, since it does not have version control capabilities:

- Lock
- Unlock
- Check Out
- Check In

SCLM RAM

RAM Description

The Software Configuration Library Manager (SCLM) sample RAM is another demonstration of CARMA's ability to interface with Source Code Managers (SCMs). The purpose of this appendix is to give the RAM developer an understanding of CARMA's SCLM RAM implementation. The SCLM RAM interfaces with an IBM Software Configuration and Library Manager (SCLM).

IBM SCLM provides an API like ISPF that provides for the dialog manager. In addition, SCLM interfaces with ISPF library management services for most of its functions. ISPF/SCLM creates and accesses variables, lists, and reports as a result of the API calls that it makes. For a full description of all the ISPF/SCLM programming services, refer to the *Software Configuration Library Manager Reference, z/OS Version 1 Release 7.0*, and also the ISPF services manual for detailed information on ISPF library management services. CARMA leverages both the ISPF library management services and SCLM services in the SCLM sample RAM.

Navigation Structure

Within the SCLM RAM, CARMA displays a selected SCLM project that is available to you on a particular connection. Each SCLM project can then be expanded to display the groups and types associated with the project.

Supported actions

The SCLM sample RAM employs the core functions of ISPF and SCLM in CARMA's User Interface. This functionality enables users to send requests to the SCLM RAM on the z/OS host and then display the results at their workstation. The following is a list of SCLM functions that can be invoked from a member's selection in the CARMA UI.

Table 35. Basic Functions

Function Name	Description
LOCK	This is a stand-alone function that enables a user to lock a member or add an access key to limit or restrict access to it by other users. This function can be enabled by right-clicking an SCLM member in the CARMA UI and selecting **Lock** from the context menu.
UNLOCK	This function will unlock a member that has been locked by removing the access key. It can be accessed by right-clicking the locked member in the CARMA UI and selecting **Unlock** from the context menu.
DELETE	This function will delete all traces of an SCLM member, including all text and any metadata, from an SCLM project. This function can be accessed by right-clicking an SCLM member in the CARMA UI and selecting **Delete** from the context menu.

The following functions are custom actions that are specific to SCLM RAM content. They can be accessed by right-clicking an SCLM member from the CARMA UI and selecting **Custom** from the context menu.

Note: The custom commands below will prompt users for additional parameters.

Table 36. Custom Actions

Service Name	Description
MIGRATE	The MIGRATE service creates or updates the SCLM accounting information for members in a development library. Pattern matching is not provided at this time.
BUILD	The BUILD service compiles, links, and integrates software components according to the project's architecture definition. Before a member is built however, the member's dependency information must exist in the project database. For this reason, either the STORE or SAVE services for a member must be completed successfully before the BUILD service can be preformed.

Table 36. Custom Actions (continued)

Service Name	Description
PROMOTE	The PROMOTE service moves data, or promotes data through the project database according to the project's architecture definition and project definition. Before SCLM can promote a member, it must have a blank access key in addition to having successfully completed the BUILD service. If a member has an access key, you must call the UNLOCK service to reset the access key before you can promote the member.
DELETE	The DELETE service deletes database components. You can delete an entire member, its associated accounting records, and build map, just the accounting records and the build map, or simply the member's build map.
EDIT	The EDIT service is not like SCLM's edit. This service is instead used to announce the intent of an edit. The user will be prompted for the development group to move the member to. A refresh is required so the user can select the member at development level by double-clicking on member. At is point; the source code will appear in the UI's edit panel.

The following are services of the SCLM API that the SCLM RAM uses to provide functionality.

Table 37. Integrated SCLM Services

Service Name	Description
SCLMINFO	The SCLMINFO service is used by CARMA's getmembers function to retrieve all SCLM groups and types stored in ISPF variables for later retrieval from getContainer.
SAVE	The SAVE service locks and parses a member then proceeds to store that member's statistical, dependency, and historical information all in one service call. The SAVE service called within CARMA's putMember function calls the LOCK, PARSE, and STORE SCLM services.

The following SCLM services maintain session integrity within existing CARMA functions.

Table 38. SCLM services

Service Name	Description
INIT	The INIT service initializes an SCLM ID. During this process, it also initializes the specified project definition.
START	The START service initializes an SCLM services session. It generates an application ID that identifies the services session.
END	The END service stops an SCLM service session and frees an application ID generated by the START service. Each START service invocation needs a matching END service invocation. This service also calls the FREE service to free any SCLM IDs associated with the given application ID that have not been explicitly freed.
FREE	The FREE service frees an SCLM ID generated by the INIT service. Each INIT service invocation needs a matching FREE service invocation. After freeing the SCLM ID, SCLM closes all project data sets and frees the project definition specified on the INIT service.

Unsupported actions

The SCLM RAM does not support the following actions. Attempting to execute any of these actions will result in an error dialog.

- Check Out

  To gain exclusive access to a source file for editing, use the Lock action. Other users will still be able to access the source file by extracting it from the repository, but they will not be permitted to check in their updates to this file until you have unlocked the file.

- Check In

  To allow another user to edit a source file, use the Unlock action.

COBOL RAM

RAM description

The COBOL RAM is an implementation of the PDS RAM written in COBOL. It is comprised of a DLL resulting from linking compiled COBOL and C source. The COBOL RAM provides functionality for browsing PDS assets in the same manner as the PDS sample RAM. Some of the functionality present in the PDS RAM is not implemented, but skeleton programs are provided for implementing additional functionality.

Navigation structure

Within the COBOL RAM, CARMA display a list of all the PDS instances available under the user's high-level qualifier on the CARMA host. Each instance can then be expanded to display the list of members that make up each instance.

Supported capabilities

The following functions are already configured on the sample COBOL RAM. Depending on what your COBOL RAM will be supporting, additional functions may need to be implemented:

- `extractMember`

  Extracts a member in the same manner as the PDS RAM. The function is coded such that extracting from any member associated with an instance of the COBOL RAM will return the records from a dataset referenced by the DD CBLIN. This DD must be added to the CARMA start-up CLIST for `extractMember` to work.

- `putMember`

  Stores a PDS member to the PDS instance specified.

- `getInstances`

  Provides a list of PDS instances.

- `getMembers`

  Returns the list of members associated with a PDS instance.

- `initRAM`

  Sets global variables and demonstrates the COBOL-to-C logging function.

- `performAction`

  Contains sample code for performing a custom action. The sample for `performAction` accepts custom parameters and then provides them as custom returns in reverse order. Information for configuring the CAF to use the custom action may be found within the program documentation for `performAction`.

Skeleton RAM

RAM Description

The Skeleton RAM is the most basic out of all of the samples. It provides a simple framework that can be used to developed a RAM that will meet your needs. This RAM should be used as a starting point for developing your own custom RAM.

Documentation notices for IBM Rational Developer for System z

© Copyright IBM Corporation - 2010

U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
3-2-12, Roppongi, Minato-ku, Tokyo 106-8711 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation

3039 Cornwallis Road, PO Box 12195
Research Triangle Park, NC 27709
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Copyright license

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademark acknowledgments

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at `www.ibm.com/legal/copytrade.shtml`.

Rational® are trademarks of International Business Machines Corporation and Rational Software Corporation, in the United States, other countries, or both.

Intel and Pentium are trademarks of Intel Corporation in the United States, or other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Readers' Comments — We'd Like to Hear from You

IBM Rational Developer for System z
Common Access Repository Manager Developer's Guide
8.5

Publication No. SC23-7660-06

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: 1-800-227-5088 (US and Canada)

If you would like a response from IBM, please fill in the following information:

Name   Address

Company or Organization

Phone No.   Email address

IBM®

Fold and Tape   **Please do not staple**   Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL   PERMIT NO. 40   ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Information Development
Department G7IA / Bldg. 503
P.O. Box 12195
Research Triangle Park, NC
 27709-2195

Fold and Tape   **Please do not staple**   Fold and Tape

SC23-7660-06

Index

V

IBM ®

Product Number: 5724-T07

Printed in USA