IBM WebSphere® Developer for zSeries® Version 6.0

IBM

Common Access Repository Manager Developer's Guide

SC31-6914-00

IBM WebSphere® Developer for zSeries® Version 6.0

Common Access Repository Manager Developer’s Guide

SC31-6914-00

Note

Before using this document, read the general information under "Notices" on page 39.

First edition (July 2005)

This edition applies to Common Access Repository Manager for version 6.0 of IBM WebSphere Developer for zSeries (product number 5724-L44) and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications by phone or fax. IBM Software Manufacturing Solutions takes publication orders between 8:30 a.m. and 7:00 p.m. eastern standard time (EST). The phone number is (800) 879-2755. The fax number is (800) 445-9269. Faxes should be sent Attn: Publications, 3rd floor.

You can also order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

IBM welcomes your comments. You can send your comments by mail to the following address:

IBM Corporation, Attn: Information Development, Department 53NA Building 501, P.O. Box 12195, Research Triangle Park, NC 27709-2195.

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 you are developing a RAM for. To use this book for CARMA client development, you need to understand generic SCM concepts.

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 used 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 WebSphere Developer for zSeries (WD/z) ships with such a software bridge to allow its CARMA client plug-in to access CARMA hosts.

CARMA currently ships with one RAM for accessing Partitioned Data Sets (PDSs). To access other SCMs using CARMA, you will need to obtain or develop additional RAMs.

The following diagram illustrates the composition of the CARMA environment:

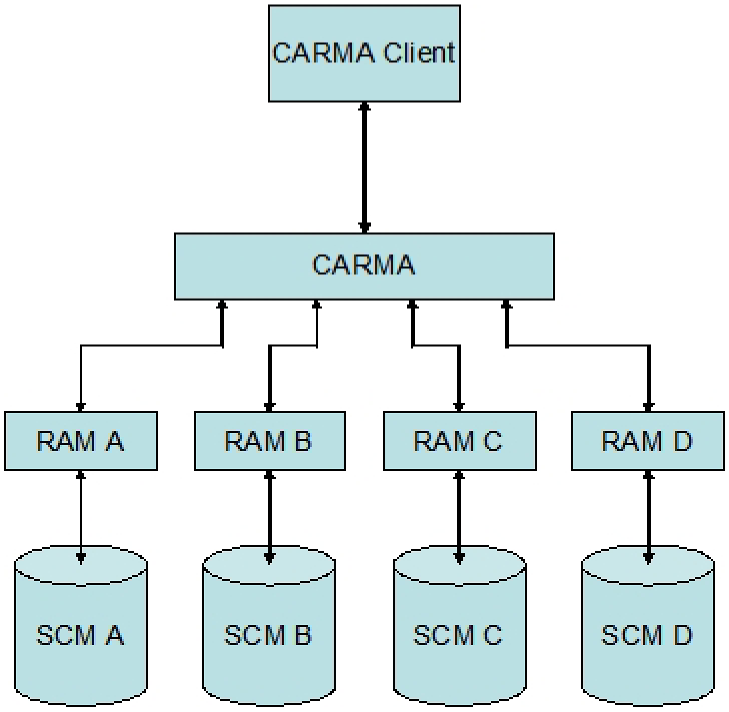

Figure 1. Example CARMA environment

Supported operations

CARMA currently supports the following generic SCM operations:

- Browse an SCM
- Extract an SCM member
- Create and update an SCM member
- Get and update SCM member metadata
- Lock, unlock, check in, and check out a member

Although CARMA supports all of these actions, it is quite possible that a given SCM may not support one or more of these operations 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 12.

Locating the sample files

The sample files are shipped with the CARMA host installation packages. After your CARMA host has been successfully set up, the default location of these sample files should be in the sample library (`CRA.SCRASAM`). However, depending on how your CARMA environment has been configured, any referenced dataset name beginning with "CRA" may use a different middle qualifier. For example, a CARMA environment could be configured to place its sample library in `CRA.TEST.SCRASAM` instead of `CRA.SCRASAM`.

The following table summarizes the available sample files, available in the sample library:

File	Description
`CRA390SD`	CARMA/390 DLL side deck
`CRACLISA`	Sample client source code
`CRASPDS`	PDS RAM source code
`CRARAMSA`	Sample (skeleton) RAM source code
`CRAADDRM`	JCL to rebuild the VSAM cluster
`CRACLICM`	JCL to compile a CARMA client
`CRA390H`	Header needed for clients
`CRADSDEF`	Header needed for clients and RAMs
`CRAFCDEF`	Header needed for RAMs
`CRACLIRN`	JCL to run a host-based client
`CRARAMCM`	JCL to compile a RAM

Chapter 2. General concepts

Browsing

CARMA views all entities within an SCM as instances, members, and metadata. Instances are the entities at the highest level within an SCM. For example, the PDS RAM uses the PDSs themselves as instances. Instances could be different libraries of code, different levels of code, or whatever the RAM developer thinks would make the most sense for client users. For most SCMs, an instance should represent a project or component in the SCM.

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 instances, the components are containers, and the source files are simple members.

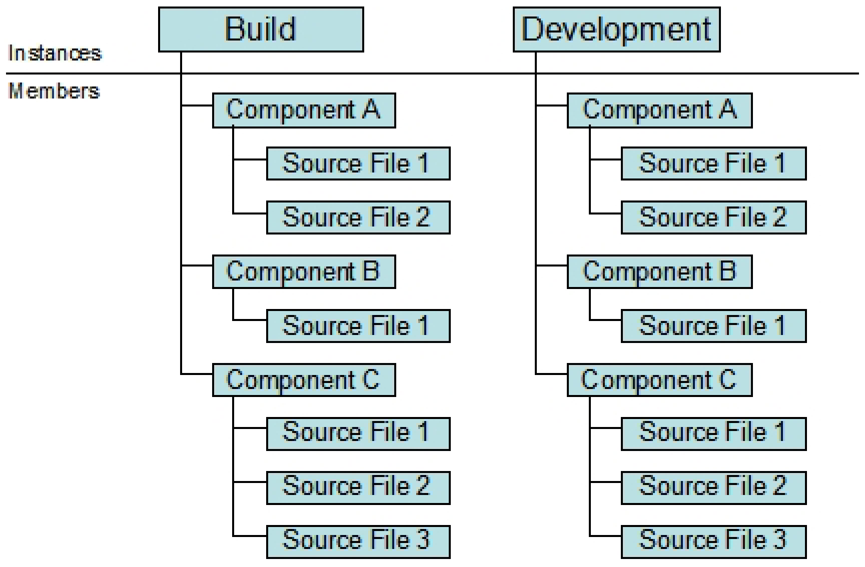

Figure 2. Example SCM hierarchy

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 function for any given SCM will respectively expect or return a member's contents, CARMA has been designed such that check in and check out are only 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 operations.

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. CARMA clients should always expect to perform a check in operation before an update operation, and to perform an extract operation immediately after a check out operation.

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, etc. The following diagram illustrates how this memory is allocated:

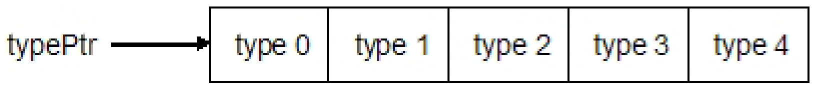

Figure 3. Simple one dimensional array as would be allocated by a RAM

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 the array. In both cases, the array should be allocated as illustrated in the following diagram:

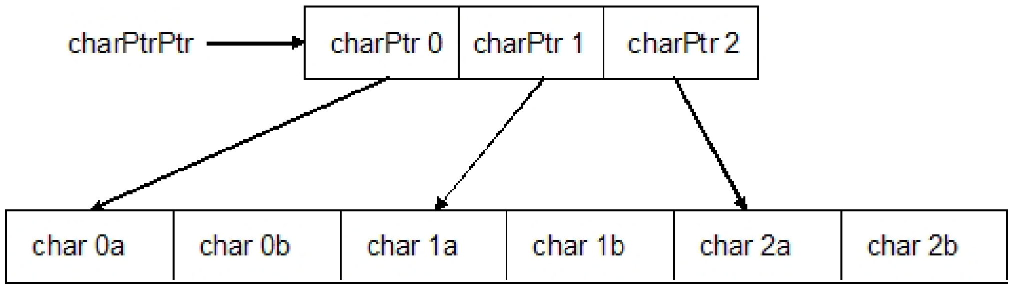

Figure 4. Two-dimensional character array as used in `extractMember` and `putMember`

The above illustration depicts the standard way to represent a two-dimensional array in C. `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 in 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 in the extractMember function) an extra step is required before the array can be allocated: charPtrPtr needs to be passed by reference to the RAM; that is, a pointer to charPtrPtr needs to be passed. This is necessary so that charPtrPtr can serve has a handle to the two-dimensional array after the RAM has allocated the array. Suppose that the RAM has a parameter named contents of type char*** in the RAM function that will allocate the two-dimensional array. The address of charPtrPtr should be passed as the value for contents. 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);
```

It is necessary to free the data array before freeing the char pointer array to avoid producing 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.

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 (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	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

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 between 100 and 200 or between 500 and 900. Codes ranging from 100 to 200 are reserved for generic errors that all RAMs may face. Codes ranging from 500 to 900 should be used for any errors that are specific to a certain RAM. Likewise, CARMA may return error codes between 4 and 100, 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 900 and 999. See "Return codes," on page 37 for a list of the predefined error codes. When an error occurs, the RAM should fill the error buffer with the details of the error.

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 1. Trace levels

Enumeration	Trace Level
0	Error

Table 1. Trace levels (continued)

Enumeration	Trace Level
1	Warning
2	Information
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 11 (for RAM development) and “Logging” on page 24 (for client development).

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. References for the API functions are 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` in the sample library) as a starting point for your RAM. 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.

Compiling the RAM

The RAM should be compiled as a DLL into a load library. The file `CRARAMCM` in the sample library can be modified to compile your RAM code into a DLL. Specifically, the `OUTFILE`, `INFILE`, `SYSLIB`, and `SYSDEFSD` dataset name symbolics need to be modified to point to your dataset locations. The following table summarizes these symbolics:

Dataset Name Smybolic	Description
`OUTFILE`	The load library your RAM should be compiled into.
`INFILE`	The source file for the RAM to compile.
`SYSLIB`	The library or libraries containing all of your headers.
`SYSDEFSD`	Specifies where the DLL's side deck should be built.

Since CARMA loads RAMs explicitly, the DLL does not necessarily require a side deck in order for the RAM to work properly. However, it should still be created in order for the JCL procedure to work properly.

To compile a RAM written in C, the `CRADSDEF` header file (located in the sample library) must be included. Currently, equivalent header files for use with other languages do not exist. However, they should be available by the time WD/z 6.0.1 ships.

Defining the RAM to CARMA

CARMA keeps its RAM information in a VSAM cluster, which should be located at `CRA.VSAMV.CRADEF` by default. The RAM records in the cluster have the format illustrated in Figure 7.

	Key					Data			
Section	Record Type (R)	Locale	RAM ID	Empty (000)	Reserved (2 spaces)	Name	Version Reposi-tory Level CARMA Level	Execut-able Name	Descrip-tion
Bytes	1	5	2	3	2	16	8 each	32	2048
Starting Index	0	1	6	8	11	13	29	53	85

Figure 7. RAM information VSAM cluster memory map.

An example VSAM cluster initialization file is available at `CRA.VSAMV.INIT` by default. This file defines several RAMs to CARMA. Edit the file or create a new sequential dataset with the same allocation settings (be sure to use 2133 as the value for `lrecl`) and fill in the fields as the samples have been filled.

The RAM ID is stored in bytes 0 through 12. It must follow the format shown in Figure 7. It should have two spaces in the reserved area, with "0"s in the three bytes before the reserved area. The two bytes preceding the "0"s should be unique within the cluster and in sequential order (RAM IDs should start at 00 and increment towards 99). The preceding five characters define a locale, and the first character in the ID must be an "R". For example, a CARMA environment could have three RAMs, defined by the following example:

```
REN_US00000
REN_US01000
RFR_FR01000
```

Note that each line in the above example ends with two space characters. The RAM's locale should only describe the locale for which the strings describing the RAM were written. RAM IDs should be constant for RAMs with different locales. In the above example, ″REN_US01000 ″ should point to the same RAM as ″RFR_FR01000 ″. The only difference between the two VSAM records is the locale of the strings used to describe the RAM to client users.

After creating or customizing your VSAM cluster initialization file, update the cluster using the JCL provided (`CRAADDRM` in the sample library). This JCL will overwrite the data in the VSAM cluster with the data in the VSAM cluster initialization file.

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 2 and Table 3.

The header file (`CRAFCDEF` in the sample library) must be included within the RAM.

Table 2. `Descriptor` data structure

Field Name	Bytes	Description
`ID`	256	Unique ID to describe the entity
`Name`	64	Display Name

Table 3. `KeyValPair` data structure

Field Name	Bytes	Description
`Key`	64	An index.
`Value`	256	The data.

Logging

CARMA provides RAMs with a pointer to a logging function, a pointer to a log file, and a trace level (see Table 1 on page 6) 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 file pointer that is passed in at initialization. An example call in C is shown below:

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

The log file will be created as a sequential dataset in the CARMA user's datasets. It will be of the format `*USERNAME*.CRA*TIMESTAMP*`, where *`USERNAME`* is the username of the user running CARMA, and *`TIMESTAMP`* is a numeric timestamp indicating the creation time of the log. For example, if user BOB is running CARMA, the log could be named `BOB.CRA15343`.

Dealing with unsupported operations

If you are developing a RAM that communicates with an SCM that does not support a CARMA operation, you can use one of the two following procedures to safely indicate to the CARMA client that an operation is unsupported:

1. Do not implement the function for that operation and do not include a `pragma export` statement for the function. This will cause CARMA to return a return code of 16 to any client that requests that operation from your RAM.
2. Implement the function for the operation to simply return a return code of 107 and include the `#pragma export` statement for the function as you normally would.

State functions

The RAM has 3 state functions: `initRAM`, `terminateRAM`, and `reset`, as illustrated in Figure 8. `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`.

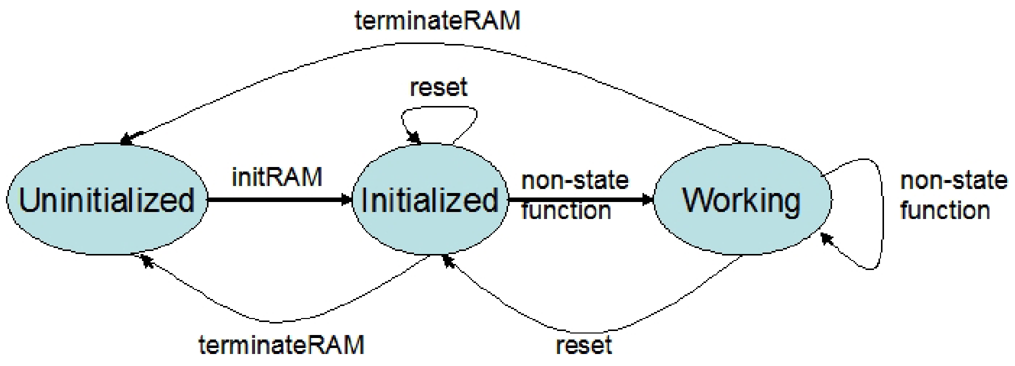

Figure 8. RAM state diagram

initRAM

```
int initRAM(Log_Func logFunc, FILE* log, int traceLev,
            char 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 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 should be 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	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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.

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	Reserved for future use
void*** customReturn	Output	Reserved for future use
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.

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	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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.

Metadata functions

getAllMemberInfo

Retrieves all of a member'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 instance containing the member

`char memberID[256]`	Input	The ID of the member whose metadata is being retrieved
`KeyValPair** contents`	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	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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'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 instance containing the member
`char memberID[256]`	Input	The ID of the member whose metadata is being retrieved
`char key[64]`	Input	The key for the value to be returned
`char value[256]`	Output	The requested value
`void** params`	Input	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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'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 instance containing the member

`char memberID[256]`	Input	The ID of the member whose metadata is being set
`char key[64]`	Input	The key for the value to be set
`char value[256]`	Input	The value to set
`void** params`	Input	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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 member operations

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 dataset and array
`int* numRecords`	Output	The number of records in the dataset/rows in the array
`char recFM[4]`	Output	Will contain the dataset'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 dataset that wasn't extracted if `*moreData` is set to 1; otherwise, undefined

`void** params`	Input	Reserved for future use
`void** customReturn`	Output	Reserved for future use
`char error[256]`	Output	If an error occurs, this should be filled with a description of the error.

`extractMember` returns the contents of the dataset 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 datasets are considered to be indexed with the first record being record 0.

Operation:

1. Determine how many records are in the dataset, 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 on page 19 shows what `extractMember` should return for each call needed to extract all the contents.

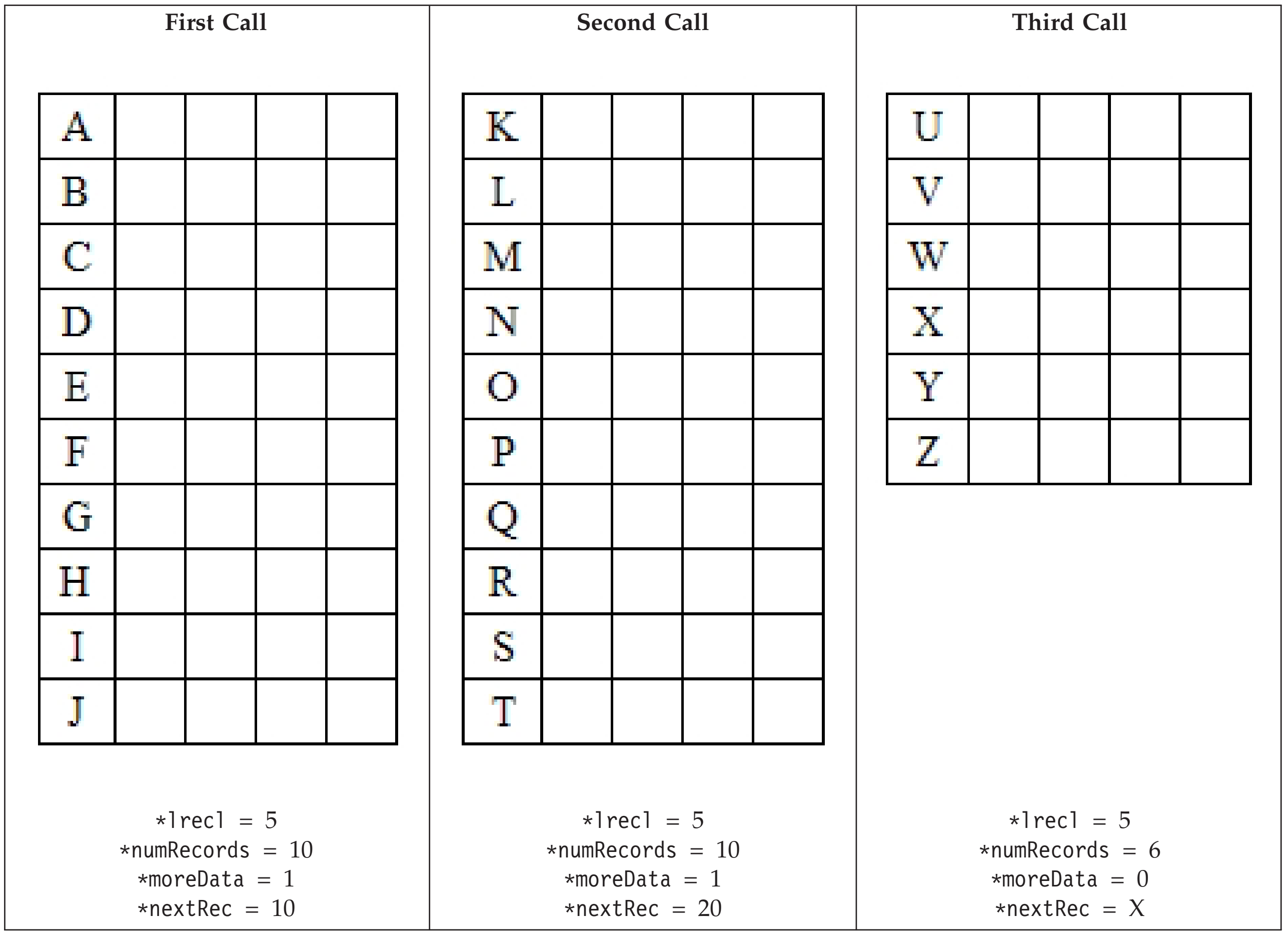

*Figure 9. Example of return values for subsequent calls to `extractMember`. Notice that during the third call, *nextRec has a listed value of X. This means that the value of `*nextRec` is not significant and will not need to be altered.*

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 dataset and array
`int* numRecords`	Input/Output	The number of records in the dataset/rows in the array.
`char recFM[4]`	Input	Contains the dataset'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 dataset 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 dataset; 0 otherwise
`void** params`	Input	Reserved for future use
`void** customReturn`	Output	Reserved for future use
`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 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 dataset. If `moreData` is equal to 1, either leave the dataset open if its state cannot be maintained between calls, or close the dataset and make sure that it can be reopened to the appropriate place with the values being passed in next time `putMember` is called.

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	Reserved for future use
`void** customReturn`	Output	Reserved for future use
`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	Reserved for future use
`void** customReturn`	Output	Reserved for future use
`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	Reserved for future use

`void** customReturn`	Output	Reserved for future use
`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	Reserved for future use
`void** customReturn`	Output	Reserved for future use
`char error[256]`	Output	If an error occurs, this should be filled with a description of the error.

Chapter 4. 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 WebSphere Developer for zSeries (WD/z). WD/z was designed to support the basic functions all RAMs have in common, so a RAM fitting perfectly into the CARMA specification would work with WD/z right out of the box. WD/z 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" and "Memory allocation" on page 4 carefully, as memory leaks and abnormal program termination can easily result from not following the recommendations on handling memory for each function.

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 IDs, and names. The RAM IDs are required by virtually every function that uses a RAM to carry out an operation.

Client Predefined Data Structures

Most RAM functions use predefined structures to pass information back to CARMA and then the RAM. The `RAMRecord` consists of a 13–byte `ID` character field, a 16–byte `name` character field, and several other character fields that describe the RAM. 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. The applicable structures are summarized in Table 4 on page 24, Table 5 on page 24, and Table 6 on page 24.

These structures are available in the CRADSDEF header file located in the sample library. These structures 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.

Table 4. `RAMRecord` data structure

Field Name	Bytes	Description
`ID`	13	Unique ID to describe the RAM
`Name`	16	Display name
`Version`	8	RAM version
`SCMLevel`	8	The level of the SCM the RAM accesses.
`Language`	8	Language in which the RAM is written
`CARMALevel`	8	The level of CARMA for which the RAM was designed.
`Module Name`	8	Name of the RAM module to load
`Description`	2048	Displayed as a RAM description by the client.

Table 5. `Descriptor` data structure

Field Name	Bytes	Description
`ID`	256	Unique ID to describe the entity
`Namename`	64	Display name

Table 6. `KeyValPair` data structure

Field Name	Bytes	Description
`Key`	64	An index.
`Value`	256	The data.

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 1 on page 6.

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`. Regardless of how the client loads CARMA, the JCL to run the client must have the location of the CARMA DLL and the RAMs in the `STEPLIB` data definition. Also, the `CRADEF` and `MSGPDS` data definitions must point to the locations of the VSAM cluster and `MSG` file PDSs, respectively. By default, the `MSG` file PDS will be created and named `CRA.MSG` during installation and the VSAM cluster will be created and named `CRA.VSAMV.CRADEF` after running the sample file `CRAADDRM`.

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` — 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 24 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.

initRAM

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

```
int initRAM(char RAMid[13], char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be initialized. This ID was obtained after running `getRAMList`.
`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(char RAMid[14], char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be reset. This ID was obtained after running `getRAMList`.
`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(char RAMid[13], char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be terminated. This ID was obtained after running `getRAMList`.
`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(char RAMid[13], Descriptor** RIrecords,int* numRecords,
                 void** params, void*** customReturn, char filter[256],
                 char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList`.
`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	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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

`getMembers`

Retrieves the list of members available within the specified instance

```
int getMembers(char RAMid[13], char instanceID[256],
               Descriptor** memberArr, int* numRecords, void** params,
               void*** customReturn, char filter[256], char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList`.
`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	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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.

isMemberContainer

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

```
int isMemberContainer(char RAMid[13], char instanceID[256],
                      char memberID[256], int* isContainer,
                      void** params, void*** customReturn,
                      char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList`.
`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	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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(char RAMid[13], char instanceID[256],
                         char memberID[256], Descriptor** contents,
                         int* numMembers, void** params,
                         void*** customReturn, char filter[256],
                         char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList`.
`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	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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.

Metadata functions

getAllMemberInfo

Retrieves all metadata for the given member

```
int getAllMemberInfo(char RAMid[13], char instanceID[256],
                     char memberID[256], KeyValPair** metadata,
                     int* num, void** params, void*** customReturn,
                     char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList`.
`char instanceID[256]`	Input	The instance the member is within
`char memberID[256]`	Input	The member for which metadata is being returned

`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	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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.

getMemberInfo

Retrieves a specific piece of metadata for the given member

```
int getMemberInfo(char RAMid[13], char instanceID[256],
                  char memberID[256], char key[64], char value[256],
                  void** params, void*** customReturn, char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList`.
`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	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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(char RAMid[13], char instanceID[256],
                     char memberID[256], char key[64], char value[256],
                     void** params, void*** customReturn,
                     char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList`.
`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	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`char error[256]`	Output	If an error occurs, this should be filled with a description of the error.

Other member operations

extractMember

```
int extractMember(char RAMid[13], 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 RAMid[13]`	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList`.
`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 dataset and array
`int* numRecords`	Output	The number of records in the dataset/rows in the array
`char recFM[4]`	Output	Will contain the dataset'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 dataset that was not extracted if `*moreData` is set to 1; otherwise, undefined
`void** params`	Input	Reserved for future use
`void** customReturn`	Output	Reserved for future use
`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 9 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 member ID is not found within the specified instance

```
int putMember(char RAMid[13], 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 RAMid[13]`	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList`.
`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 dataset and array

`int* numRecords`	Input/Output	The number of records in the dataset/rows in the array
`char recFM[4]`	Input	Contains the dataset'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 dataset 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 dataset; 0 otherwise.
`void** params`	Input	Reserved for future use
`void** customReturn`	Output	Reserved for future use
`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)));
```

and should be freed after the call in a manner similar to the following (the example is in C):

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

lock

Locks the member

```
int lock(char RAMid[13], char instanceID[256], char memberID[256],
        void** params, void*** customReturn, char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList`.
`char instanceID[256]`	Input	The instance the member is within
`char memberID[256]`	Input	The member to be locked
`void** params`	Input	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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 RAMid[13], char instanceID[256], char memberID[256],
           void** params, void*** customReturn, char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList`.
`char instanceID[256]`	Input	The instance the member is within
`char memberID[256]`	Input	The member to be unlocked
`void** params`	Input	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`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(char RAMid[13], char instanceID[256], char memberID[256],
            void** params, void*** customReturn, char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList`.
`char instanceID[256]`	Input	The instance the member is within
`char memberID[256]`	Input	The member to be checked in
`void** params`	Input	Reserved for future use
`void*** customReturn`	Output	Reserved for future use

`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(char RAMid[13], char instanceID[256], char memberID[256],
             void** params, void*** customReturn, char error[256])
```

`char RAMid[13]`	Input	Tells CARMA which RAM should be worked on. This ID was obtained after running `getRAMList`.
`char instanceID[256]`	Input	The instance the member is within
`char memberID[256]`	Input	The member to be checked out
`void** params`	Input	Reserved for future use
`void*** customReturn`	Output	Reserved for future use
`char error[256]`	Output	If an error occurs, this should be filled with a description of the error.

Appendix. Return codes

Return Code	Description
20	Internal error
22	VSAM cluster contains no records
24	VSAM cluster not found
26	No RAMs defined for this locale
28	VSAM Cluster read error
30	No RAM records found in the VSAM cluster
32	Invalid RAM record found in the VSAM cluster
34	Requested RAM not found
36	Could not load RAM module
38	Could not load pointer to RAM function
40	Requested RAM has not been loaded
44	Error in MSG file
46	Failed to initialize CARMA
48	Failed to load the RAM list

Notices

Trademarks and service marks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, or other countries, or both:

- IBM
- WebSphere
- zSeries

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.

Readers' Comments — We'd Like to Hear from You

IBM WebSphere® Developer for zSeries® Version 6.0
Common Access Repository Manager Developer's Guide

Publication No. SC31-6914-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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.

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SC31-6914-00

Cut or Fold
Along Line

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

SC31-6914-00

Cut or Fold
Along Line

Program Number: 5724-L44

Printed in USA

SC31-6914-00