IBM WebSphere Developer for zSeries Version 6.0.1

Common Access Repository Manager Developer's Guide

SC31-6914-00

IBM WebSphere Developer for zSeries Version 6.0.1

Common Access Repository Manager Developer's Guide

SC31-6914-00

Note

Before using this document, read the general information under "Notices" on page 75.

Second edition (November 2005)

This edition applies to Common Access Repository Manager for version 6.0.1 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.

Conventions used in this book

Throughout this book there are several references to data sets and members that have the high-level qualifier `CRA`. 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 `CRA.SCRASAM` in this book could actually be named `MYCORP.TEST.SCRASAM` on your host system. Thus, depending on the configuration of your host system, the `CRA` 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 WebSphere Developer for zSeries (WD/z) ships with such a software bridge to allow the WD/z CARMA plug-in to access CARMA hosts.

Figure 1 on page 2 illustrates an example CARMA environment.

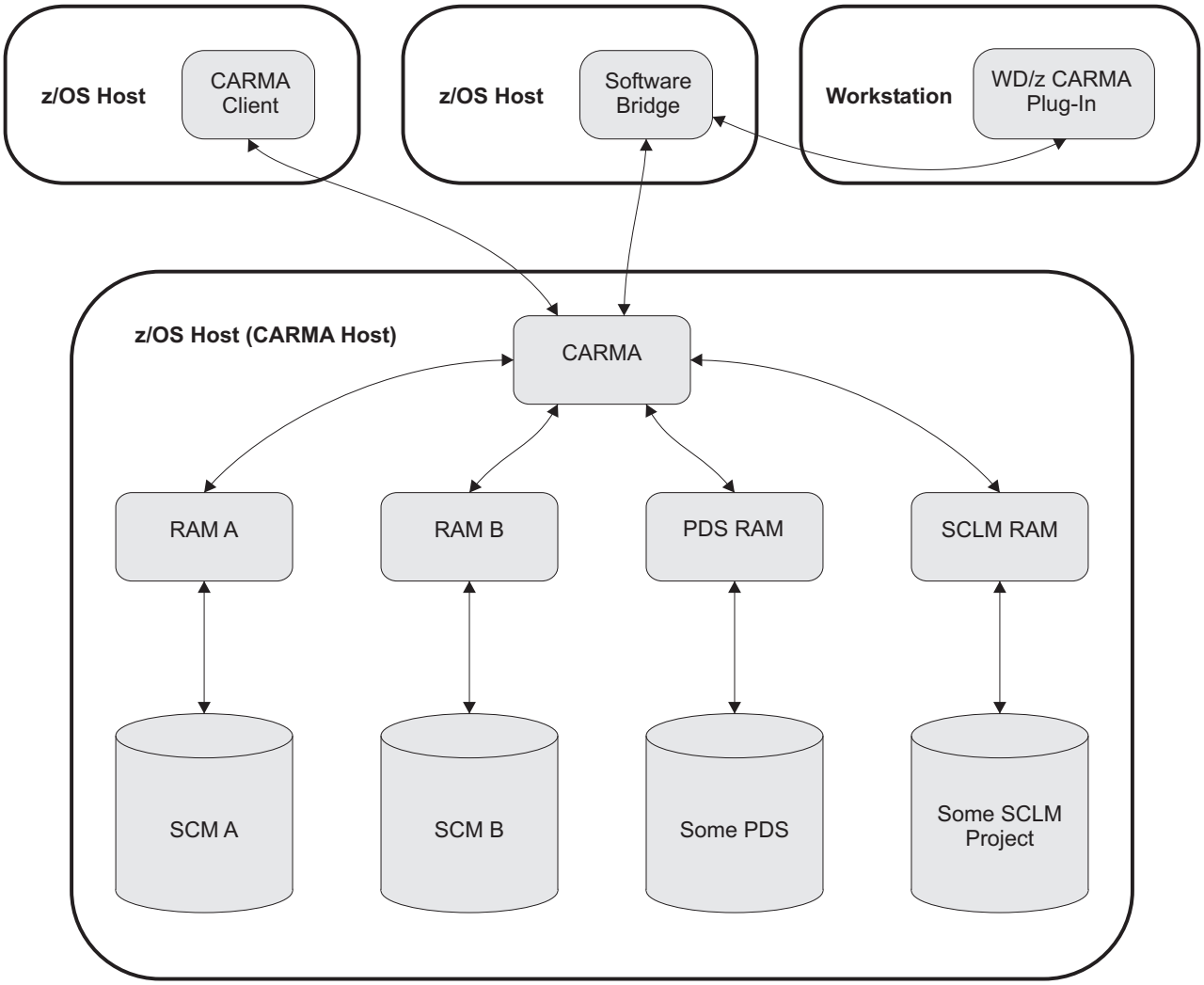


Figure 1. Example CARMA environment

CARMA currently ships with four sample RAMs:

- Sample PDS RAM - Provides access to the Partitioned Data Sets (PDSs) available on the CARMA host
- 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 to develop their own RAM

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 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 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 13.

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 37 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 (`CRA.SCRASAM`). The following table summarizes these members:

Table 1. Sample CARMA development files

Member in `CRA.SCRASAM`	Description
`CRA390H`	Header needed for clients
`CRA390SD`	CARMA/390 DLL side deck
`CRACLICM`	JCL to compile a CARMA client
`CRACLIRN`	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
`CRAMREPR`	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
`CRARAMCM`	JCL to compile Skeleton RAM
`CRARAMCS`	JCL to compile the C source for the sample SCLM RAM
`CRARAMSA`	Skeleton RAM source code
`CRAREPR`	JCL to REPRO `CRADEF`
`CRAREPRP`	JCL to REPRO the sample PDS RAM's messages
`CRAREPRS`	JCL to REPRO the sample SCLM RAM's messages

Table 1. Sample CARMA development files (continued)

Member in **CRA.SCRASAM**	Description
CRARREXX	JCL to compile the sample SCLM RAM's required REXX modules
CRASBLD	REXX source file for the SCLM BUILD API call
CRASLCK	REXX source file for the SCLM LOCK API call
CRASPDS	Source code for the sample PDS RAM
CRASPRM	REXX source file for the SCLM PROMOTE API call
CRASREPR	JCL to REPRO CRASTRS
CRASSCLM	Source code for the sample SCLM RAM
CRASSV	REXX source file for the SCLM SAVE API call (for putMember)
CRASUL	REXX source file for the SCLM UNLOCK API call

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 sample 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.

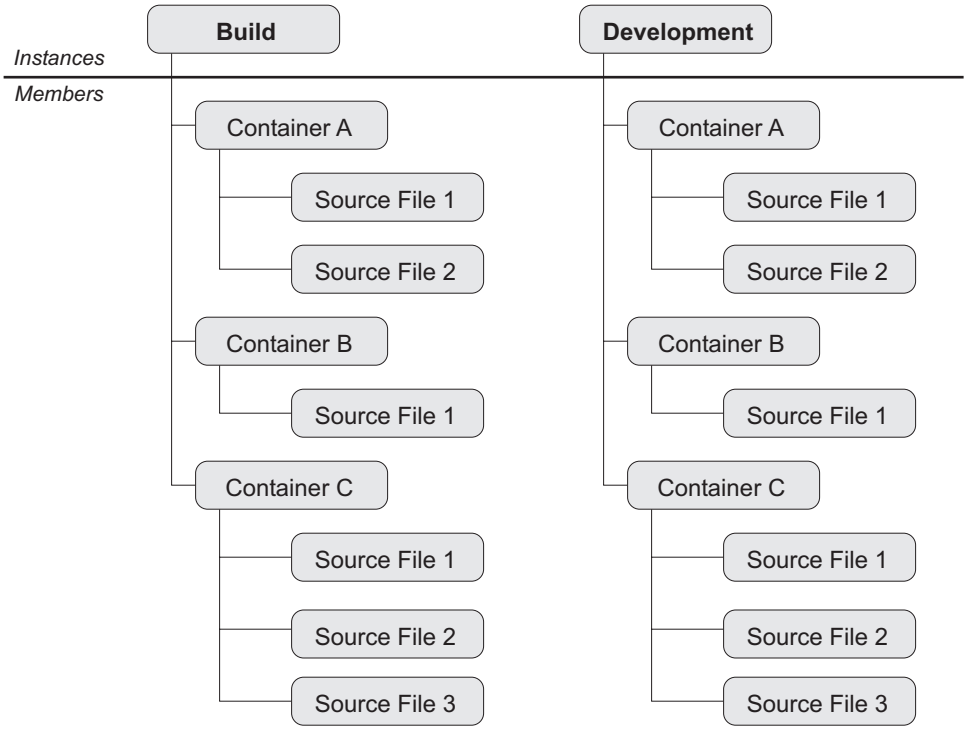


Figure 2. Example SCM hierarchy

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 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. CARMA clients should always expect to perform the check in action before performing the update action, and to perform the extract action immediately after performing the check out action.

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:

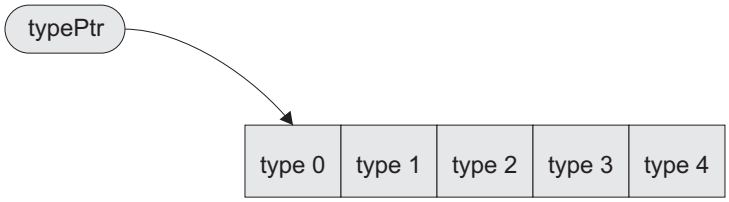


Figure 3. Simple one dimensional array as would be allocated by a RAM

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:

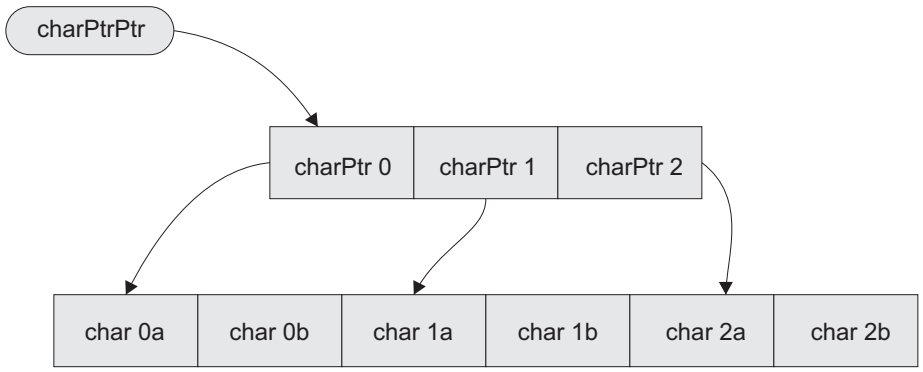


Figure 4. Two-dimensional character array as used in `extractMember` and `putMember`

`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.

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 Appendix A, “Return codes,” on page 71 for a list of the predefined error codes. When an error results in a return code between 500 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 2. Trace levels. Messages at the "None" trace level are not logged.

Enumeration	Trace Level
-1	None
0	Error
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 13 (for RAM development) and “Logging” on page 56 (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:

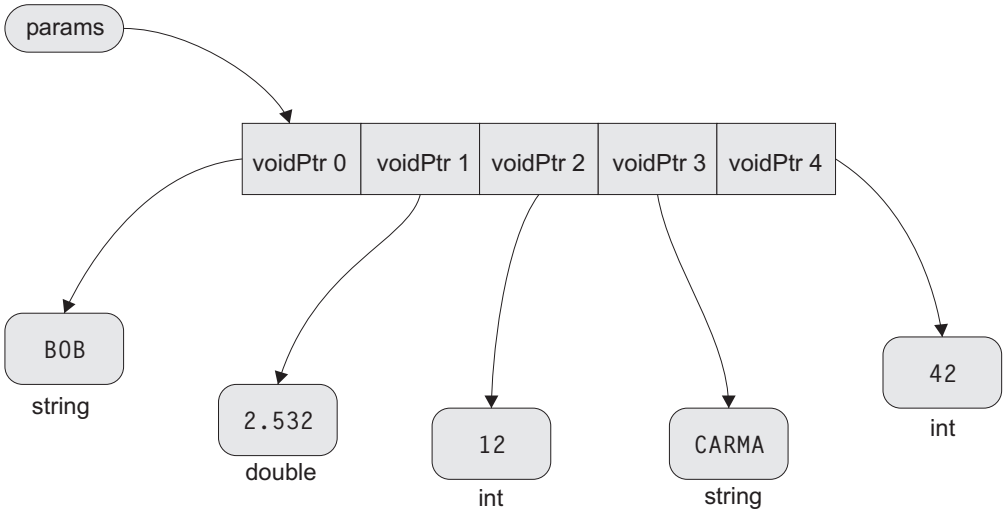


Figure 7. Custom parameter array example. Each element in the array is a pointer to a parameter. The value of each parameter is shown and labeled with its data type.

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 45). 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` CARMA function to retrieve information on the custom actions, parameters, and return values for a RAM (see "`getCAFData`" on page 69 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 14

Compiling a RAM

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

Data Set 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 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. `CRADSDEF` contains several data structures necessary to the RAM's operation. If a RAM is being developed in a language other than C, the

Descriptor and KeyValPair data structures must be implemented before the RAM can properly communicate its results to CARMA.

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 37 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 3 and Table 4.

Table 3. Descriptor data structure

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

Table 4. 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 2 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 log file will be created as a sequential data set in the CARMA user's data sets. It will be of the format *USERNAME*.CRA*TIMESTAMP*, where *USERNAME* is the user name of the user running CARMA, and *TIMESTAMP* is a numeric time stamp indicating the creation time of the log. For example, if user BOB is running CARMA at 3:42 PM, the log could be named BOB.CRA1542.

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 37). 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:

1. 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 16 to any client that requests that operation from your RAM.
2. 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 37 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. Use the following C code as an example:

```
int param0;
char* param1;
double param2;

param0= *( (int*) params[0]);
memcpy(param1 (char*), 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.

If the result values of param0, param1, and param2 from the example above needed to be returned, it could be done using the following C code:

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

/* Program body */
*return0 = 5;
memcpy(*return1, "THE STRING", 10);
*return2 = 3.41

/* 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.

State functions

The RAM has three 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.

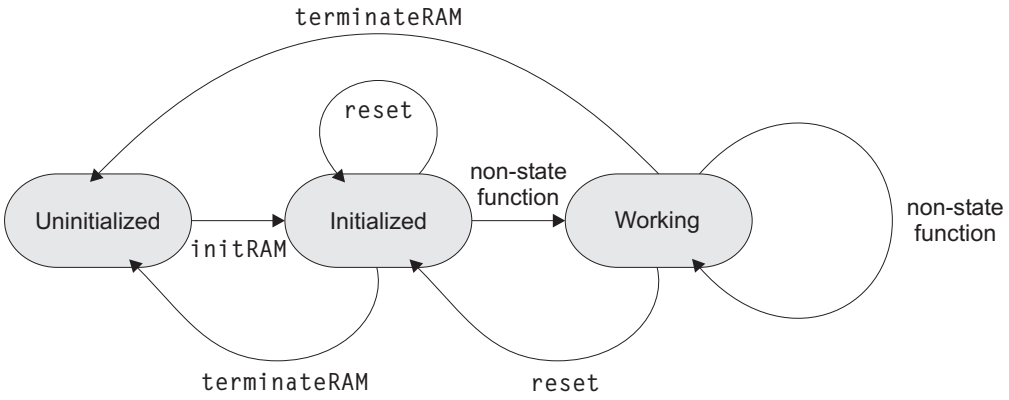


Figure 8. RAM state diagram

initRAM

```
int initRAM(Log_Func logFunc, FILE* log, int traceLev,
            char 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 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	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 13)
`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 13)
`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	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 13)

`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 13)
`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	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 13)
`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 13)
`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 13)
`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 13)
`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	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 13)
`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 13)
`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	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 13)
`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 13)
`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	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 13)
`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 13)
`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 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 13)
void** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 13)
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.

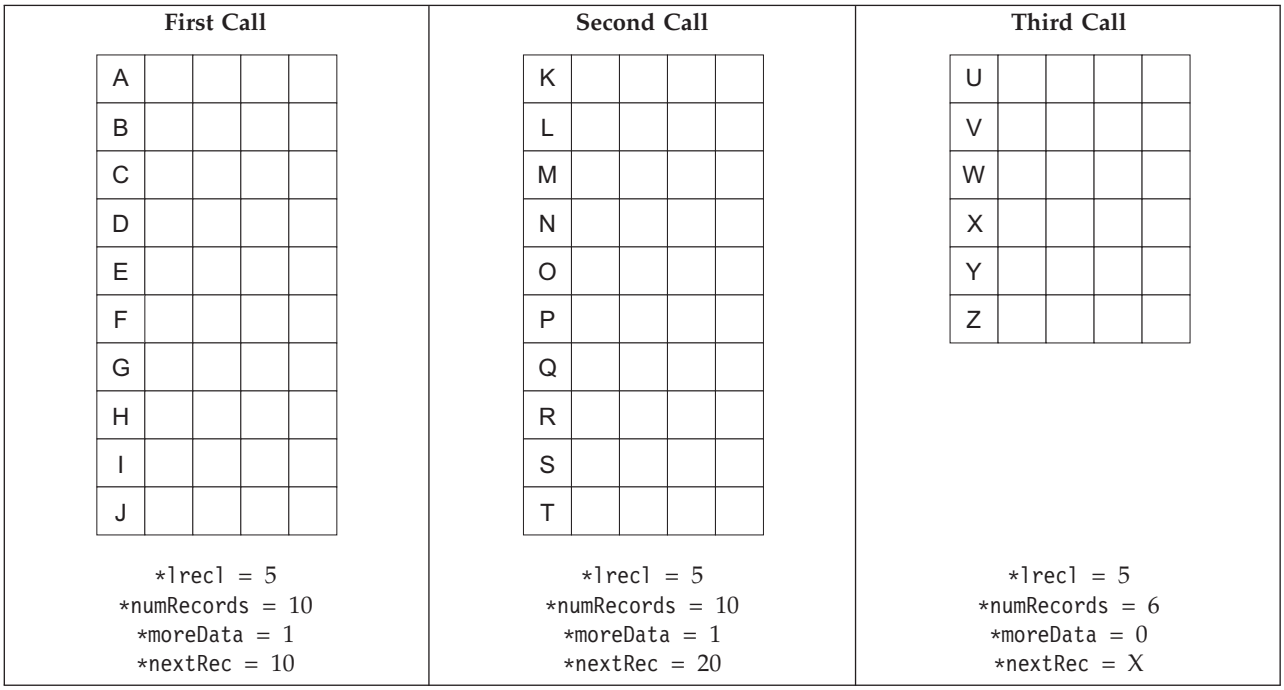


*Figure 9. Example of return values for subsequent calls to `extractMember`. Notice that during the third call, *nextRec has a listed value of X. This means that the value of `*nextRec` is not significant and will not need to be altered.*

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 13)
`void** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 13)
`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 3 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.

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 13)
`void** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 13)
`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 13)
`void** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 13)

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 13)
void** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 13)
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 13)
void** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 13)
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

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 CUTILS DLL.
- Code must be added to the RGVA DLL in order to share variables between RAM function programs.
- 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. 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.

Compilation

Compiling the sample COBOL RAM

The JCL for compiling the sample COBOL RAM (`CRACOBJ1`, located in the sample library) can be modified as necessary to add new source code to the sample COBOL RAM, compile source code for your own COBOL RAM, include side decks for new helper DLLs, or change the destination PDS for COBOL RAM builds.

Compiling helper DLLs

Source code is provided for the two helper DLLs used by the sample COBOL RAM. Each helper DLL can be compiled from a single source file. The following table summarizes these source files (all of the source files and JCL listed can be found in the sample library):

Source File Name	Language	JCL to Compile Source	DLL Name
CRACOB14	COBOL	CRACOBJ3	RGVA
CRACOBC1	C	CRACOBJ2	CUTILS

You can use these source files and JCL scripts as a starting point for developing any helper DLLs you may need for the development of your own COBOL-based RAM.

Note: When using this JCL to compile a COBOL RAM, this JCL must reference the definition side deck for any code utilized by the RAM.

COBOL RAM program structure

Coding the program ID

RAMs developed in C implement several CARMA functions, such as `initRAM` or `getMembers`. RAMs developed in COBOL instead implement each of these functions as individual COBOL programs (called *RAM function programs*). At compile time, the source code for each program is included and compiled into a single DLL, which exports each program ID to a definition side deck for use by CARMA. 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).

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

```
77  GIP-RECORDS                  POINTER.
77  GIP-NUMRECS                  POINTER.
77  GIP-PARAMS                   POINTER.
77  GIP-RETURN                   POINTER.
77  GIP-FILTER                   POINTER.
77  GIP-ERROR                    POINTER.
```

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 code for this error buffer:

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

Note: The error buffer cannot be used within the program until addressability has been established using `SET`. Refer to "Dealing with pointer operations" on page 29 for more information on establishing addressability.

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.

Note: For simplicity and consistency, the provided example code passes parameters by value as often as possible. However, a few examples pass parameters by reference as needed in certain situations.

The following example procedure division declaration illustrates how you might designate parameters to be passed by value.

```
PROCEDURE DIVISION USING BY VALUE GIP-RECORDS
                         BY VALUE GIP-NUMRECS
                         BY VALUE GIP-PARAMS
                         BY VALUE GIP-RETURN
                         BY VALUE GIP-FILTER
                         BY VALUE GIP-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.

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 associated with each RAM function are provided to the COBOL compiler as a concatenated DD statement. Failing to provide END PROGRAM directives will cause programs to be treated as nested, which will yield compiler error messages.

ILC data type equivalents between C and COBOL

Because CARMA is intended to be ILC-compatible, COBOL data type equivalents must exist for C data types. A thorough treatment of this topic is given within Language Environment documentation (*Language Environment Writing Interlanguage Communication Applications*). However, for your convenience, mappings for common data types are listed in the following table:

C Data Type	COBOL Equivalent
int	PIC S9(9) BINARY
char *	POINTER
char **	POINTER
void *	POINTER
void ***	POINTER

Since COBOL is not as strongly-typed as C, any pointer passed to a COBOL program is effectively similar to a void pointer in C. In other words, once a pointer is passed to a COBOL program, it is the responsibility of the COBOL programmer to ensure that the pointer is handled properly for the correct data type. There is no distinct data type associated with the COBOL pointer as there is in C.

Since double and triple pointers may be passed into COBOL-based RAMs, it may be necessary to dereference these pointers multiple times. Although this task is more elegantly accomplished in C, it is possible to do in COBOL. However, to avoid the complexities associated with pointer operations in COBOL, sample code has been provided for a C utility DLL that performs pointer operations that would be overly complicated to implement in COBOL. The C utility DLL is further discussed in "Dealing with pointer operations."

Dealing with pointer operations

Simple pointer operations

For most parameters passed to COBOL RAM function programs, a small amount of pointer dereferencing code is necessary 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, it must be dereferenced using `SET`.

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 C utility DLL. 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.

Alternatively, complex pointer operations can be performed within COBOL, but can decrease the code readability. To deal with dynamic structures, pointer arithmetic is necessary through using 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 to allocate memory. The C utility DLL has a C function called `CMALLOC` to provide access to `malloc` from within COBOL code. The `CMALLOC` function accepts as an argument an integer containing the number of bytes to be allocated and returns a pointer to the chunk 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 chunk of allocated memory.

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

```
01  MALLOC-SIZE                PIC S9(9) BINARY.
01  VOID-POINTER-RETURNED      POINTER.

CALL "CMALLOC" USING BY VALUE MALLOC-SIZE
               RETURNING VOID-POINTER-RETURNED.
```

Variables shared between programs

Many RAM functions for a RAM written in C may have variables defined with a global scope. As such, each function can access and modify the values of these variables and, in turn, make these values accessible to other functions. However, because each RAM function is implemented in COBOL as a separate COBOL program, there is no way to directly share variables between programs. Because of this problem, a DLL consisting of sample code (`CRACOB14`, located in the sample library) has been shipped, which demonstrates a way to store and retrieve variables that are intended to have global scope. This code is also known as RGVA.

RGVA (RAM Global Variables Accessor) provides a transactional interface for retrieving data that is global within the RAM's scope. Within RGVA's code, each variable is associated with an index value. When invoking RGVA, pass a flag indicating whether or not you want to store or retrieve the variable's value, the index of the variable to store or retrieve, and a buffer that contains the variable itself. The code for RGVA is intended to be extensible, meaning that it can be updated to store a variety of variables with minimal coding. Note that it is necessary to update the code for RGVA to provide storage for additional variables.

Retrieving variables

In order to retrieve a variable, one must pass several pieces of information to RGVA grouped under a 1-level item. The first is a flag that indicates whether to retrieve ("R") or store ("S") the variable's value. The second flag indicates the index of the variable to be retrieved. The third portion that must be allocated is a sufficient amount of space to store the item being retrieved.

CAUTION:
If enough space is not allocated, the results of this operation will be unpredictable and could result in the abnormal termination of your RAM.

For example, to retrieve the variable with index 1, which is a pointer to the logging function within the distributed sample, one would set up an area in working storage with code similar to the following:

```
01  RGVA-ARGUMENTS.
    05  RGVA-FLAG                      PIC X VALUE  'R'.
    05  RGVA-WHICH-ONE                 PIC X VALUE  '1'.
    05  RGVA-SOME-POINTER              POINTER.
```

In this example, the information in the working storage section is hard-coded to retrieve the variable with index 1, which happens to be of type `POINTER`. To retrieve the data, you would call RGVA using the following code:

```
CALL 'RGVA' USING RGVA-ARGUMENTS.
```

After this call, the value of the pointer being fetched from RGVA will be contained in `RGVA-SOME-POINTER`. You could then utilize this value as necessary.

Storing variables

Storing variables works in a similar manner to retrieving variables. As an example, the following code illustrates another entry for the working storage section that is used to store a value:

```
01  RGVA-ARGUMENTS.
    05  RGVA-FLAG                      PIC X VALUE  'S'.
    05  RGVA-WHICH-ONE                 PIC X VALUE  '8'.
    05  RGVA-TEST-AVG                  PIC 9(3).
```

This entry is hard-coded to store the variable with the index 8, which is a test average consisting of 3 numeric digits. RGVA is then invoked as before:

```
MOVE 100 TO RGVA-TEST-AVG.
CALL 'RGVA' USING RGVA-ARGUMENTS.
```

After the call has been made, RGVA will have stored the variable for later retrieval (provided that RGVA has been updated with the code for storing this value).

Adding to RGVA

In order to customize RGVA, you will have to make additions to the RGVA code. First, the variable being added will need to be included as a 5-level item to the `RAM-GLOBAL-VARIABLES` group within RGVA's working storage.

Secondly, `100-STORE-VARIABLE` and `200-RETRIEVE-VARIABLE` will need to have their `EVALUATE` statements updated to perform the action that will store the new variable. Depending on your coding practices, it may be necessary to create a paragraph to perform the task of storing or retrieving the variable value. This code is not present in the sample, but it would be necessary to implement this sort of logic so that you could implement multiple custom actions. The sample code treats `performAction` as a custom action.

Lastly, it may be necessary to update `RGVA-PARMS-IN` within the linkage section to provide a section that redefines `RGVA-PARM-SPACE`. For instance, if a new variable named `RGVA-TEST-AVG` were being added to RGVA, the updated group would be defined as follows:

```
01  RGVA-PARMS-IN.
    05  RGVA-RS-FLAG               PIC X.
    05  RGVA-WHICH-ONE             PIC X.
    05  RGVA-PARM-SPACE            PIC X(256).
    05  FPTR-LOG-FUNC              REDEFINES RGVA-PARM-SPACE
                                   POINTER.
    05  FILE-PTR-LOG               REDEFINES RGVA-PARM-SPACE
                                   POINTER.
```

```
    05  INT-TRACE-LEVEL             REDEFINES RGVA-PARM-SPACE
                                     PIC S9(9) BINARY.
    05  STRING-ERROR                REDEFINES RGVA-PARM-SPACE
                                     PIC X(256).
    05  RGVA-TEST-AVG-IN            REDEFINES RGVA-PARM-SPACE
                                     PIC 9(3).
```

In order to accommodate the new variable, RAM-GLOBAL-VARIABLES would be defined as follows after the update:

```
01  RAM-GLOBAL-VARIABLES.
    05  RGV-LOG-FUNC-WRITETOLOG     POINTER.
    05  RGV-FILE-PTR-LOGPTR         POINTER.
    05  RGV-INT-TRACELEVEL          PIC S9(9) USAGE IS BINARY.
    05  RGV-CALL-COUNT              PIC 9(9).
    05  RGV-TEST-AVG                PIC 9(3).
```

Once these additions have been made, the variable may be referenced by name for storing or retrieving within the code for RGVA.

Overview of the function programs of the sample COBOL RAM

The sample COBOL RAM uses the following function programs. Depending on which CARMA functions your COBOL RAM supports, you may need to write a different set of function programs for your COBOL RAM.

extractMember
: Extracts ten records from a data set and returns them to the client. Regardless of the instanceID or memberID passed to the program, the same data set will be extracted. The data set must be defined by a DD statement for CBLIN within the CLIST for starting CARMA. You should point CBLIN to a data set with an LRECL of 80 and RECFM of FB. This program serves as an example of how to perform extraction and deal with memory allocation issues in COBOL.

putMember
: Writes a single chunk of data to the same data set that is extracted by extractMember. The size of the chunk written is the number of records passed to putMember. The incoming data is expected to have a RECFM of FB and an LRECL of 80. The value for nextRec should be zero.

getInstances
: Returns a list of three instances to the client. These instances are static, hard-coded values.

getMembers
: Returns a list of two members regardless of the instanceID passed to it. Thus, when viewed from the client, each instance will have exactly two members under it.

initRAM
: Contains example code for making calls to the RGVA DLL.

performAction
: Contains example code for dealing with custom parameters and custom return values.

Handling Custom Action Framework data

The sample COBOL source file (CRACOB17, located in the sample library) implements a simple custom action that accepts an integer and a string as custom parameters, displays them, and then sends these parameters back as custom

returns in reverse order. Thus, this sample code illustrates the techniques for creating custom actions, utilizing custom parameters, and passing custom return values.

Custom actions

Custom actions may be created by using CRACOB17 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 PA-ACTIONID:

```
EVALUATE PA-ACTIONID
     WHEN 100 PERFORM CUSTOM-ACTION-100
     WHEN 200 PERFORM CUSTOM-ACTION-200
     WHEN OTHER
           PERFORM-LOG-ERROR-AND-EXIT
END-EVALUATE.
```

Custom parameters

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.

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

Custom returns

Accessing custom return values within a COBOL RAM requires more caution than dealing with custom returns. 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:

```
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.
```

Debugging and avoiding abnormal termination

There are several utilities 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. It is possible to write a routine that converts a pointer's value to hexadecimal and then displays the converted value, but this is left as an exercise for the reader.

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 the enclave containing both CARMA and the COBOL RAM. It is recommended that you do not use `STOP RUN` statements unless you wish for the RAM to exhibit 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

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 four types of objects used in CAF information: RAMs, parameters, and return values, and actions.

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 0 and 999. Parameter IDs for a RAM *must* be sequential, starting at 0. For example, you cannot only define parameters with the following IDs: 0, 1, and 3. You must also define a parameter with a parameter ID of 2.

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	Arbitrary (this value does not matter)
long	Arbitrary (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 0 and 999. Return value IDs for a RAM *must* be sequential, starting at 0. For example, you cannot only define return values with the following return value IDs: 0, 1, and 3. You must also define a return value with a return value ID of 2.

RAM ID
The ID of the RAM the return value belongs to

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	Arbitrary (this value does not matter)
long	Arbitrary (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 73 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.

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:

- 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:
  1. Instance ID
  2. Reason

  and returns the following values:
  1. Lock type
  2. Return code
- Has a "disenflaguate" action, which removes a flag from a member within the SCM. This action requires the following parameters:
  1. Instance ID
  2. Member ID
  3. Reason

  and returns the following values:
  1. Return code

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

  and returns the following values:
  1. New instance ID
  2. New member ID
  3. 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 "Compiling a RAM" 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 37). The following table summarizes this information:

Table 5. 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
CARMA Version	2.5

At this time, you may find it helpful to tabulate the information (as described in "Action" on page 39) 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 73) 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 6. Information about SAMP RAM's lock instance action

Name	Lock instance
Description	Locks an instance within the SCM

Table 6. Information about SAMP RAM's lock instance action (continued)

Action ID	100
RAM ID	1
Parameter List	Instance ID Reason
Return Value List	Return code Lock type

Table 7. 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 8. 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 9. 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 10. 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 39), 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 39). 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 on page 45 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.

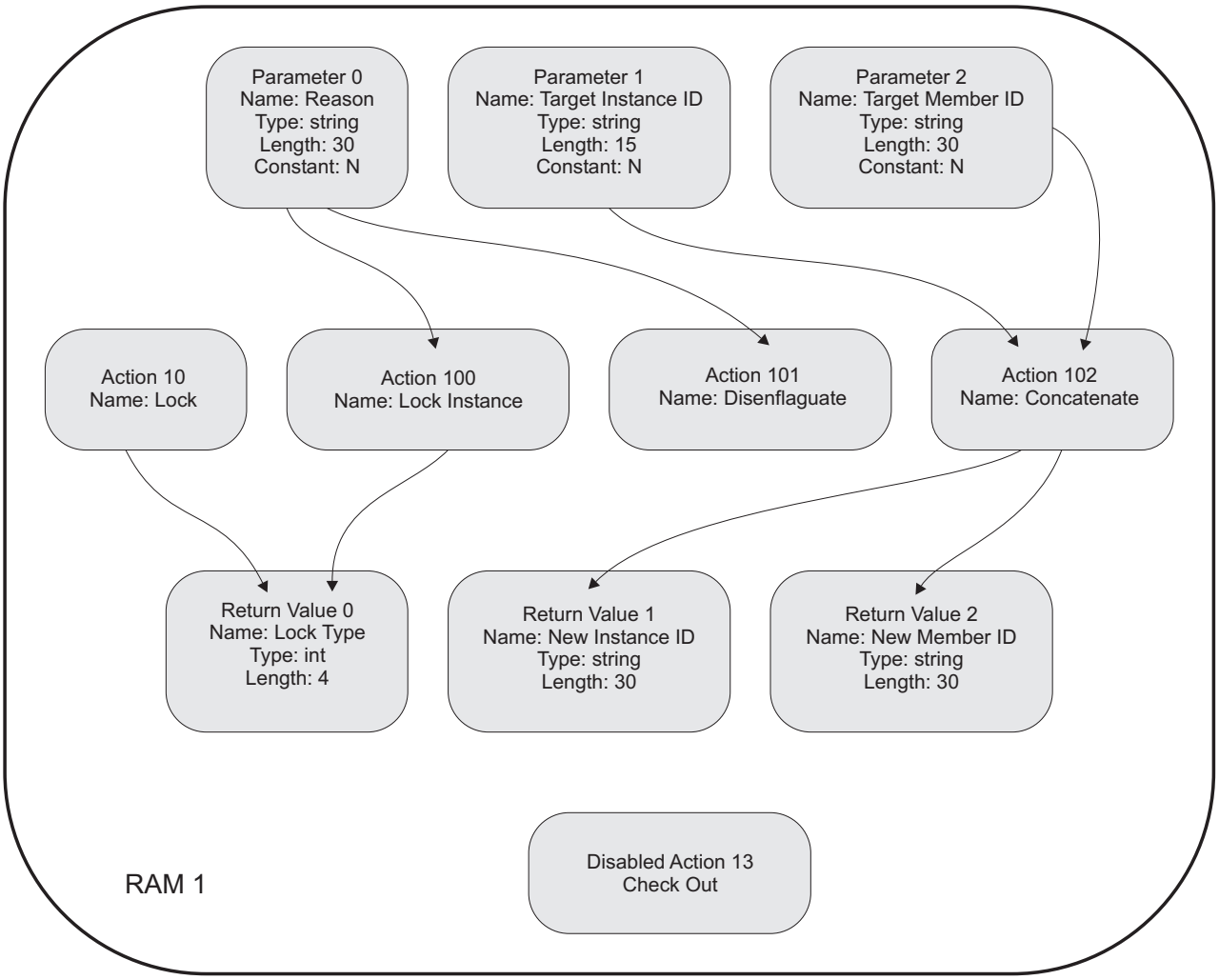


Figure 10. Visual representation of the SAMP RAM model. Only information relevant to the relationship between the objects is shown.

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 do 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 11. CRADEF record format

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

Record keys

CRADEF record keys are composed of the following fields:

1. (1 byte) The type character (″A″ for action, ″D″ for disabled action, ″P″ for parameter, ″R″ for RAM, and ″T″ for return value)
2. (2 bytes) The two-digit RAM ID left-padded with 0s (a unique identification number between ″00″ and ″99″)
3. (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 and return values, you should use sequential IDs starting at ″000″.
4. (2 bytes) Unused (reserved for future use). Fill these bytes with spaces.

The following table summarizes the CRADEF key format.

Table 12. 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

1. (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".

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

Table 13. 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

Table 13. 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. (continued)*

Type	1024-Byte Data	
T	(16 bytes) Type	(16 bytes) Length

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 uses a record width of 2101 bytes. CRASTRS uses a 21-byte key and reserves the remaining 2080 bytes for data. The following table summarizes the composition of a generic record in CRASTRS:

Table 14. CRASTRS record format

2101-Byte Record	
(21 bytes) Key	(2080 bytes) Data

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:

1. (8 bytes) The locale of the record (for example, "EN_US ")
2. (5 bytes) The code page of the record (for example, "00037")
3. (8 bytes) The key to the CRADEF record to which this CRASTRS record corresponds

The following table summarizes the CRASTRS key format.

Table 15. 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 bytes in each record are used for the record data. These 2080 bytes contain different fields depending on the record type:

RAM, action, and return type

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

Parameter

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

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 16. 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. The number of bytes reserved for each field is specified in parentheses.

Type	2080-Byte Data			
A **R** **T**	(16 bytes) Name		(1024 bytes) Description	
P	(16 bytes) Name	(16 bytes) Default Value	(1024 bytes) Prompt	(1024 bytes) 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 17. SAMP RAM records (one per row) in CRADEF. Each cell represents a field. Refer to "CRADEF" on page 45 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 CRA.SCRAVSAY(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, CRA.SCRAVSAY(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 CRA.SCRAVSAY(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 CRA.SCRALIB(CRAREPR) to REPRO CRA.SCRAVSAY(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 18. SAMP RAM records (one per row) in CRASTRS. Each cell represents a field. Refer to "CRASTRS" on page 48 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 Instância		Bloqueia a instância	
PT_BR	01047	A01101	Tirar sinalizador		Remove um sinalizador	
PT_BR	01047	A01102	Concatenar		Concatena dois conjuntos de dados	

Table 18. SAMP RAM records (one per row) in CRASTRS. Each cell represents a field. Refer to "CRASTRS" on page 48 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	P01000	Motivo	Por que não?	Por que você deseja que eu execute a ação?	O motivo para executar a ação
PT_BR	01047	P01001	ID de Instância de Destino	MyInstance	Em qual instância o membro está localizado?	A instância que contém o membro a ser concatenado
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 Instância		O ID de instância de concatenação	
PT_BR	01047	T01002	Novo ID do Membro		O ID do membro de concatenação	

Please refer to CRA.SCRAVSAY(CRASINIT) for an example of the proper column format. This sequential data set is used to initialize CRASTRS during CARMA installation. Like CRA.SCRAVSAY(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, CRA.SCRAVSAY(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 CRA.SCRAVSAY(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 CRA.SCRALIB(CRASREPR) to REPRO CRA.SCRAVSAY(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.

Cluster editing tool

Currently, there is a tool under development that will provide RAM developers with an easy-to-use dialog-based interface from which they can add all the CAF information for their RAMs to the VSAM clusters. Once development of this tool is complete, developers will no longer need to edit the clusters directly since the tool will automatically convert RAM models into cluster records.

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 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 RAM API 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" on page 54 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. The `RAMRecord` consists of an integer RAM ID, 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 `Parameter` structure consists 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. The `returnValue` structure consists of an integer ID, a 16-byte name, a 16-byte type, an integer length, and a 1024-byte description. The `Action` structure consists 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.

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. 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 19. `RAMRecord` data structure

Field	Description
`int id`	Unique ID to describe the RAM
`char name[16]`	Display name
`char version[8]`	RAM version
`char reposLevel[8]`	The level of the SCM the RAM accesses.
`char language[8]`	Language in which the RAM is written
`char CRALevel[8]`	The level of CARMA for which the RAM was designed.
`char moduleName[8]`	Name of the RAM module to load
`char description[2048]`	Displayed as a RAM description by the client.

Table 20. `Descriptor` data structure

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

Table 21. `KeyValPair` data structure

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

Table 22. `Action` data structure

Field	Description
`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]`	The action's name
`int* paramArr`	A list of the IDs for the parameters the action uses
`int numParams`	The number of elements in the `paramArr` array
`int* returnArr`	A list of the IDs for the return values the action returns
`int numReturn`	The number of elements in the `returnArr` array
`char description[1024]`	A short description of the action

Table 23. `Parameter` data structure

Field	Description
`int id`	A numeric identifier for the parameter between 0 and 999

Table 23. `Parameter` data structure (continued)

Field	Description
`char name[16]`	The parameter's name
`char type[16]`	The data type of the parameter ("INT", "LONG", "DOUBLE", or "STRING")
`char defaultValue[16]`	The parameter's default value
`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`	Whether or not the parameter will always contain the same value
`char prompt[1024]`	The prompt that the CARMA client should display when requesting a value for the parameter from users
`char description[1024]`	A short description of the parameter

Table 24. `returnValue` data structure

Field	Description
`int id`	A numeric identifier for the return value between 0 and 999
`char name[16]`	The return value's name
`char type[16]`	The data type of the return value ("INT", "LONG", "DOUBLE", or "STRING")
`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.
`char description[1024]`	A short description of the return value

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 2 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` 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;
params[2] = (void*) &param2;
```

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:

```
/* Declared at top */
int return0;
char* return1;

/* Call the CARMA function */
/* ... */

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

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 56 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(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`.
`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`.
`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`.
`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`.

`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 56)
`void*** customReturn`	Output	Used to reference an array of custom return values (see “Handling custom parameters and return values” on page 56)
`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(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`.
`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 56)
`void*** customReturn`	Output	Used to reference an array of custom return values (see “Handling custom parameters and return values” on page 56)

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(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.
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 56)
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 56)
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.

`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 56)
`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 56)
`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(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`.
`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	Pointer to an array of custom parameters (see "Handling custom parameters and return values" on page 56)
`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 56)
`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(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`.
`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 56)
`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 56)
`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.
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 56)
void*** customReturn	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 56)
char error[256]	Output	If an error occurs, this should be filled with a description of the error.

Other operations

extractMember

```
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.
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 56)
`void** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 56)
`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 member ID is not found within the specified 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`.
`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 56)
`void** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 56)
`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(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`.
`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 56)
`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 56)
`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`.

`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 56)
`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 56)
`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`.
`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 56)
`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 56)
`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`.
`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 56)
`void*** customReturn`	Output	Used to reference an array of custom return values (see "Handling custom parameters and return values" on page 56)
`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 25.

`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

Table 25. (continued)

`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 37 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.

Appendix A. Return codes

Return Code	Description
20	Internal error
22	No RAMs defined for this locale
24	CRADEF not found
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
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
109	Invalid parameter value
110	Member cannot be updated
111	Member cannot be created
112	Not authorized
113	Could not initialize

Return Code	Description
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
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
80	initCarma
81	terminateCarma
82	getRAMList
83	getCAFData

Notices

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

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

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:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

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

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 OR CONDITIONS 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.

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:

IBM Corporation
P.O. Box 12195, Dept. TL3B/B503/B313
3039 Cornwallis Rd.
Research Triangle Park, NC 27709-2195
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.

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.

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.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may 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 illustrates 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. 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 IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(C) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (C) Copyright IBM Corp. 2000, 2004. All rights reserved.

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 6.0.1
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

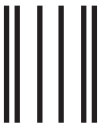
Company or Organization

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