

IBM Rational Developer for System z
8.5

*Guia do Desenvolvedor do Common
Access Repository Manager*



IBM Rational Developer for System z
8.5

*Guia do Desenvolvedor do Common
Access Repository Manager*



Nota

Antes de utilizar este documento, leia as informações gerais em “Notas de Documentação para IBM Rational Developer for System z” na página 125.

Sétima Edição (Junho de 2012)

Esta edição se aplica ao Common Access Repository Manager para versão 8.0.3 do IBM Rational Developer for System z (número do produto 5724-T07) e a todas as liberações e modificações subsequentes até que seja indicado de outra forma em novas edições.

Solicite as publicações pelo telefone ou fax. O IBM Software Manufacturing Solutions recebe os pedidos de publicações entre 8h30 e 19h, horário padrão na costa leste dos Estados Unidos. O número de telefone é (800) 879-2755. O número de fax é (800) 445-9269. O fax deve ser enviado para: Publications, 3rd floor.

Você também pode solicitar as publicações através de um representante IBM ou da filial da IBM que atende em sua região. As publicações não são guardadas no endereço abaixo.

A IBM agradece pelo seu comentário. Você pode enviar os comentários por correio ao seguinte endereço:

IBM Corporation
Rodovia SP 101 Km 09
CEP 13185-900
Hortolândia,
SP

Ao enviar informações à IBM, você concede à IBM o direito não-exclusivo de utilizar ou distribuir as informações da forma que julgar apropriada, sem incorrer em qualquer obrigação para com o Cliente.

Nota sobre Direitos Restritos para Usuários do Governo dos Estados Unidos - Uso, duplicação e divulgação restritos pelo documento GSA ADP Schedule Contract com a IBM Corp.

© Copyright IBM Corporation 2000, 2012.

Índice

Sobre este Manual v

Quem Deve Ler Este Manual v

Convenções Usadas Neste Manual v

Capítulo 1. Introdução ao CARMA . . . 1

Operações Suportadas 3

Localizando Arquivos de Amostra 3

Capítulo 2. Conceitos Gerais. 5

Navegação 5

Registro de Entrada e Saída 6

Alocação de Memória 6

Conteúdo de Membro 8

Buffers de Caracteres 8

Códigos de Retorno 9

Log 9

Parâmetros e Valores de Retorno Customizados . . . 10

Capítulo 3. Desenvolvendo um RAM . . 13

Construção do RAM 13

Construção de um PDS 13

Construção de um PDS/E 14

Usando o Módulo de Utilitários do RAM 14

utilInitMemberList 14

utilGetNextMember 15

utilCloseMemberList 15

utilGetAllMemberInfo 15

utilGetMemberInfo 16

utilSetMemberInfo 16

utilGetAllPDSInfo 16

utilCopyPDStoPDS 17

utilCopyPDStoSDS 17

utilCopySDStoPDS 17

utilCopySDStoSDS 17

utilPutMemberInit 17

utilPutMemberRecs 18

utilPutMemberRec 18

utilPutMemberClose 18

utilExtractMemberInit 18

utilExtractMemberRec 19

utilExtractMemberClose 19

Definindo o RAM para o CARMA. 19

Exportando Funções 20

IDs versus Nomes 20

Estruturas de Dados Predefinidas pelo RAM . . . 20

Log 20

Lidando com Operações Não Suportadas 21

Manipulando Parâmetros e Valores de Retorno

Customizados 21

Metadados Definidos pelo CARMA 22

Extensão de Arquivo Especificada pelo RAM . . 22

Versão do CARMA 23

Funções de Estado 24

initRAM. 24

terminateRAM. 25

reset 25

Funções de Navegação 25

getInstances 25

getInstancesWithInfo 26

getMembers 27

getMembersWithInfo 28

isMemberContainer 30

getContainerContents 30

getContainerContentsWithInfo 31

Criar/Excluir 32

Funções de Transferência de Arquivo. 34

extractMember 34

putMember 37

Extrair para Externo 39

Transferência de Arquivo Binário 40

Funções de Metadados 41

getAllMemberInfo 41

getMemberInfo 42

updateMemberInfo 43

Outras Operações 44

lock 44

unlock 44

check_in 45

check_out 45

performAction 46

getVersionList 47

Desenvolvimento do RAM Usando COBOL 48

Estrutura de Programa do RAM COBOL 49

Passando Valores de C para COBOL 50

Passando Dados de COBOL para C 53

Lidando com Operações de Ponteiro 54

Variáveis Compartilhadas entre Programas . . . 55

Manipulando Dados do Custom Action

Framework 56

Diferenças entre “DLL de Utilitário” e “Origem

de Utilitário COBOL para C” 58

Depurando e Evitando Finalização Anormal . . . 59

Capítulo 4. Customizando uma API do RAM Usando o CAF 61

Tipos de Objeto CAF 61

RAM 61

Parâmetro 62

Valor de Retorno 62

Ações 63

Campo 64

Desenvolvendo o Modelo de RAM para um RAM

Customizado 64

Criando Registros VSAM de um Modelo do RAM

70

CRADF 70

CRASTRS. 73

Registros VSAM do RAM SAMP 75

Acesso a Clusters do VSAM. 77

Capítulo 5. Desenvolvendo um Cliente

CARMA 79

Compilando o Cliente CARMA. 79

Executando o Cliente 79

Armazenando Resultados para Uso Posterior . . . 80

Estruturas de Dados Predefinidas pelo Cliente. . . 80

Log 84

Manipulando Parâmetros e Valores de Retorno

Customizados 85

Metadados Definidos pelo CARMA 85

Extensão de Arquivo Especificada pelo RAM . . 85

Extraír para Externo 86

copyFromExternal 86

copyToExternal 86

Funções de Estado 87

initCarma 87

getRAMList 88

getRAMList2 88

initRAM. 88

reset 89

terminateRAM. 89

terminateCarma 89

Funções de Navegação 90

getInstances. 90

getInstancesWithInfo. 90

getMembers 91

getMembersWithInfo 92

isMemberContainer 93

getContainerContents. 94

getContainerContentsWithInfo. 94

Criar/Excluir. 95

Funções de Transferência de Arquivo. 98

extractMember 98

putMember 100

Transferência de Arquivo Binário. 102

Funções de Metadados 103

getAllMemberInfo 103

getFieldsData 104

getFieldsData2. 105

getMemberInfo 105

updateMemberInfo 106

Outras Operações 106

lock 106

unlock. 107

checkin 108

checkout 108

performAction 109

getCAFDData 110

getCAFDData2. 110

getVersionList 111

Apêndice A. Códigos de Retorno . . . 115

Apêndice B. IDs de Ação 117

Apêndice C. RAMs de Amostra 119

PDS RAM 119

Descrição do RAM 119

Estrutura de Navegação 119

Ações Suportadas 119

Ações Não-Suportadas 119

RAM do SCLM. 119

Descrição do RAM 119

Estrutura de Navegação 120

Ações Suportadas 120

Ações Não Suportadas 122

RAM COBOL 122

Descrição do RAM 122

Estrutura de Navegação 122

Recursos Suportados 122

RAM Esqueleto. 123

Descrição do RAM 123

Notas de Documentação para IBM

Rational Developer for System z . . . 125

Licença de Direitos Autorais 126

Reconhecimentos de Marca Registrada 127

Índice Remissivo 131

Sobre este Manual

Este manual explica como desenvolver gerenciadores de acesso a repositório (RAMs) e clientes Common Access Repository Manager (CARMA). Ele inclui os seguintes tópicos:

- Como desenvolver um RAM capaz de conectar-se a um gerenciador de configuração de software (SCM)
- Como desenvolver um cliente CARMA capaz de acessar vários SCMs por meio do CARMA usando RAMs

É possível usar este documento como um guia para essas tarefas ou como uma referência de programação.

Quem Deve Ler Este Manual

Este manual destina-se a programadores de aplicativo ou a qualquer pessoa que deseja saber como RAMs e clientes são desenvolvidos.

Para usar este manual como guia para desenvolvimento de RAM, é preciso estar familiarizado com o SCM para o qual você está desenvolvendo um RAM. Para usar este manual para desenvolvimento de cliente CARMA, você deve entender os conceitos genéricos do SCM.

Convenções Usadas Neste Manual

Em todo este manual, há diversas referências a conjuntos de dados e membros com o qualificador de alto nível FEK. Dependendo de como o host do CARMA foi configurado, esses conjuntos de dados podem realmente ter nomes de arquivo diferentes. Por exemplo, a biblioteca de amostra referida como FEK.SFEKSAMP neste manual pode na verdade ser chamada de MYCORP.TEST.SFEKSAMP no sistema host. Assim, dependendo da configuração do sistema host, o FEK nos nomes de conjuntos de dados referenciados neste manual podem ser substituídos por alguma outra sequência. Entre em contato com o programador de sistema para determinar se esses conjuntos de dados estão de fato localizados no sistema host.

Capítulo 1. Introdução ao CARMA

CARMA é uma biblioteca que fornece uma interface genérica aos gerenciadores de configuração de software (SCMs) do z/OS. Os desenvolvedores podem construir sobre o CARMA desenvolvendo gerenciadores de acesso a repositório (RAMs) que se conectam ao ambiente do CARMA. Os RAMs definem como o CARMA deve se comunicar com vários SCMs. Por exemplo, um host do CARMA (uma máquina host do z/OS com o CARMA nela) pode ser configurado para usar um RAM para se comunicar com repositórios IBM® Source Code Library Manager (SCLM) e outro RAM para se comunicar com seu próprio SCM customizado.

Usando o CARMA, os desenvolvedores de software cliente podem evitar escrever código especializado para acessar SCMs e permitir facilmente o suporte para qualquer SCM ao qual um RAM está disponível. O CARMA é uma DLL armazenada em um MVS PDS. Somente clientes z/OS podem acessar o CARMA diretamente. Para acessar o CARMA a partir de uma estação de trabalho, uma ponte de software entre a estação de trabalho e o host deve ser desenvolvida. Esse software ponte deve agir como um cliente para o host do CARMA e como um servidor para as estações de trabalho. Essa ponte de software acompanha o IBM Rational Developer for System z para permitir que o plug-in do CARMA acesse os hosts do CARMA.

A Figura 1 na página 2 ilustra um exemplo de ambiente do CARMA.

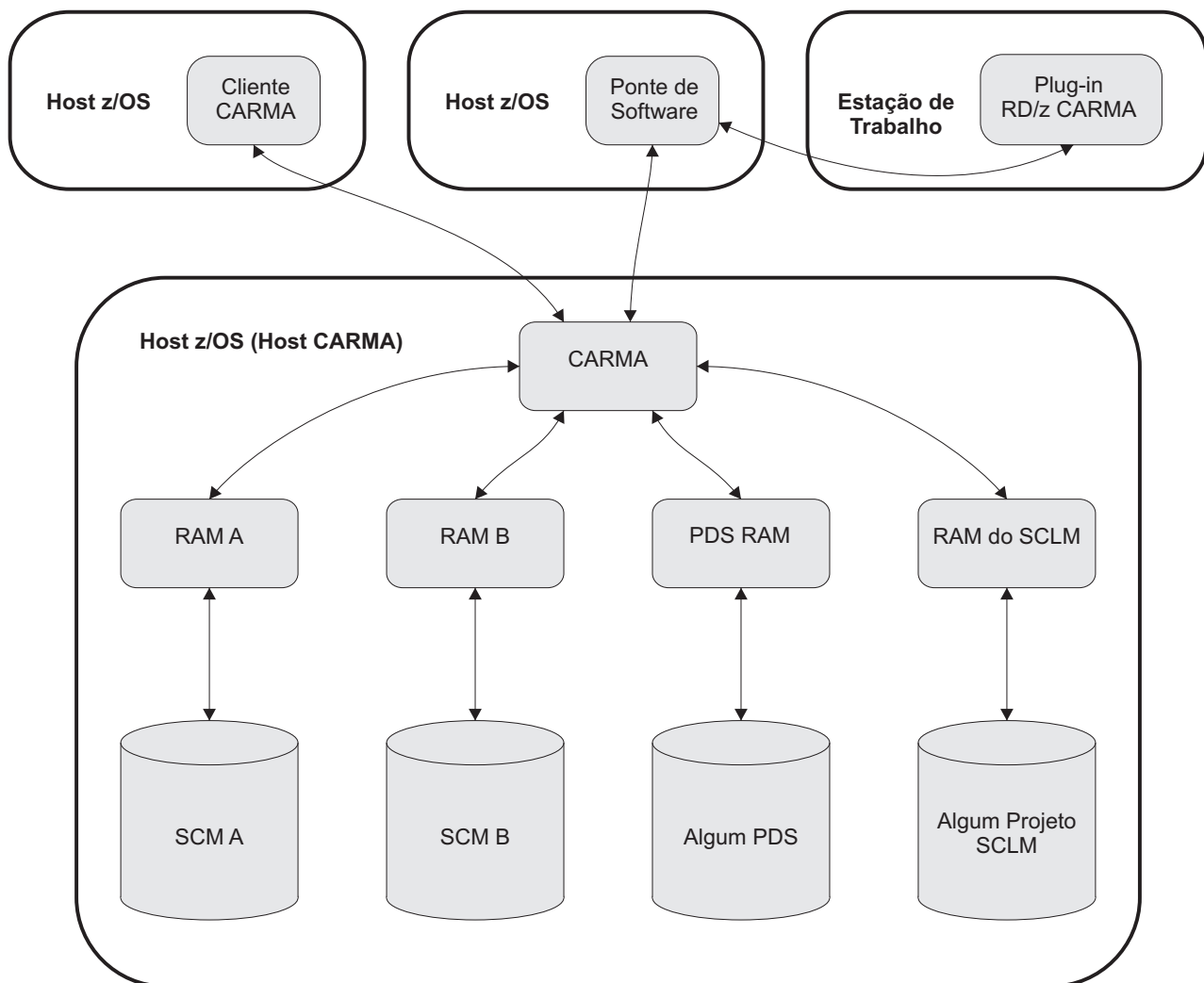


Figura 1. Exemplo de ambiente do CARMA

Quatro RAMs de amostra acompanham o CARMA atualmente:

- RAM TSO/ISPF PDS de amostra - Fornece acesso aos Conjuntos de Dados Particionados (PDS) por intermédio do uso da API de Gerenciamento de Biblioteca do TSO.
- RAM SCLM de amostra - Fornece acesso aos projetos Software Configuration Library Manager (SCLM).
- RAM COBOL de amostra - Fornece código COBOL de exemplo que demonstra o tratamento de problemas de ILC específicos do desenvolvimento de RAM baseado em COBOL.
- RAM esqueleto - Fornece um ponto de início para os desenvolvedores de RAM.

Nota: Os RAMs de amostra são fornecidos com o propósito de testar a configuração do ambiente CARMA e como exemplos para que você desenvolva seus próprios RAMs. **NÃO utilize os RAMs de amostra fornecidos em um ambiente de produção.**

Para acessar seus próprios SCMs usando o CARMA, você precisará obter ou desenvolver RAMs adicionais. Consulte o Capítulo 2, “Conceitos Gerais”, na página 5

página 5 e o Capítulo 3, “Desenvolvendo um RAM”, na página 13 para obter mais informações sobre como desenvolver um RAM para acessar seu próprio SCM.

Operações Suportadas

O CARMA suporta atualmente os seguintes conjuntos de ações genéricas:

- Navegar em um SCM
- Extrair um membro do SCM
- Criar um membro do SCM
- Atualizar um membro do SCM
- Obter metadados de membro do SCM
- Atualizar metadados de membro do SCM
- Copiar um membro para um PDS ou SDS
- Copiar um membro de um PDS ou SDS
- Excluir um membro ou contêiner
- Bloquear, desbloquear, efetuar o registro de saída e de entrada de um membro
- Navegar no histórico de um membro do SCM

Embora o CARMA suporte todas essas ações, é bem possível que um determinado SCM não possa suportar uma ou mais dessas ações devido ao seu design. Os desenvolvedores de RAMs que acessam tais SCMs devem seguir as diretrizes de manipulação de operações não suportadas em “Lidando com Operações Não Suportadas” na página 21.

O CARMA fornece também uma estrutura chamada Custom Action Framework (CAF) para a customização das ações que um RAM pode executar (consulte o Capítulo 4, “Customizando uma API do RAM Usando o CAF”, na página 61 para obter mais informações).

Localizando Arquivos de Amostra

Arquivos de amostra foram incluídos nos pacotes de instalação do host do CARMA. Depois que o host do CARMA tiver sido configurado com êxito, você conseguirá localizar esses arquivos de amostra como membros dentro da biblioteca de amostra (FEK.SFEKSAMP). A tabela a seguir resume esses membros:

Tabela 1. Arquivos de desenvolvimento de amostra do CARMA

Membro em FEK.SFEKSAMP	Descrição
CRA390H	Cabeçalho necessário para clientes
CRA390SD	Deck paralelo de DLL CARMA/390
CRA#CCLT	JCL para compilar um cliente CARMA para um PDS/E
CRA#PCLT	JCL para compilar um cliente CARMA para um PDS
CRA#XCLT	JCL para executar um cliente baseado em host
CRACLISA	Código de origem do cliente de amostra
CRADSDEF	Cabeçalho C necessário para clientes e RAMs
CRAFCDEF	Cabeçalho C necessário para RAMs
CRASUTIL	Código de origem para as funções de utilitário de RAM
CRAHUTIL	Cabeçalho necessário para funções de utilitário de RAM
CRA\$VMSG	IDCAMS JCL para REPRO CRAMSG

Tabela 1. Arquivos de desenvolvimento de amostra do CARMA (continuação)

Membro em FEK.SFEKSAMP	Descrição
CRAMSGH	Arquivo de cabeçalho comum aos RAMs de PDS e SCLM de amostra
CRAMSGO	Módulo de objeto comum aos RAMs de PDS e SCLM de amostra
CRA#CCOB	JCL para compilar o RAM COBOL de Amostra para um PDS/E
CRA#PCOB	JCL para compilar o RAM COBOL de Amostra para um PDS
CRA#CRAM	JCL para compilar o RAM Esqueleto para um PDS/E
CRA#PRAM	JCL para compilar o RAM Esqueleto para um PDS
CRA#CSLM	JCL para compilar o RAM SCLM de amostra para um PDS/E
CRA#PSLM	JCL para compilar o RAM SCLM de amostra para um PDS
CRA#CPDS	RAM PDS para um PDS/E
CRA#PPDS	RAM PDS para um PDS
CRARAMSA	Código de origem do RAM Esqueleto
CRA\$VDEF	JCL para REPRO CRADEF
CRA#VPDS	JCL para REPRO das mensagens do RAM PDS de amostra
CRA#VSLM	JCL para REPRO das mensagens do RAM SCLM de amostra
CRASPDS	Código de amostra para o RAM PDS de amostra
CRA\$VSTR	JCL para REPRO CRASTRS
CRASSCLM	Código de amostra para o RAM SCLM de amostra

Nota: Os membros CRA\$* foram copiados para FEK.#CUST.JCL para customização durante a configuração do Developer for System z. Peça ao programador de sistema uma cópia dessas JCLs customizadas para usar como seu próprio ponto de início.

Capítulo 2. Conceitos Gerais

Esta seção descreve os conceitos gerais que são essenciais para entender como o CARMA funciona. Para obter uma visão geral mais profunda desses conceitos, leia *Integrating Source Code Management Systems into WebSphere Developer for zSeries CARMA* (SC23-5817-00) localizado na biblioteca do IBM Rational Developer for System z (<http://www.ibm.com/software/awdtools/devzseries/library/>)

Navegação

O CARMA visualiza todas as entidades dentro de um SCM como **Instâncias de Repositório** (ou RIs), membros e metadados. Instâncias de Repositório são as entidades no nível mais alto dentro de um SCM. Por exemplo, o RAM PDS de amostra usa PDSs como RIs. Os RIs podem ser bibliotecas e níveis de código diferentes, ou o que o desenvolvedor de RAM achar que faria mais sentido para clientes CARMA. Para a maioria dos SCMs, um RI deveria representar um projeto ou componente no SCM. As Instâncias de Repositório são referidas mais comumente como instâncias durante discussão mais adiante.

Membros são entidades contidas em instâncias ou em outros membros. Os membros que contêm outros membros são conhecidos como **contêineres**, enquanto membros que não contêm outros membros são conhecidos como membros simples.

A Figura 2 ilustra uma hierarquia simples. "Construção" e "Desenvolvimento" são instâncias de repositório, os componentes são contêineres e os arquivos de origem são membros simples.

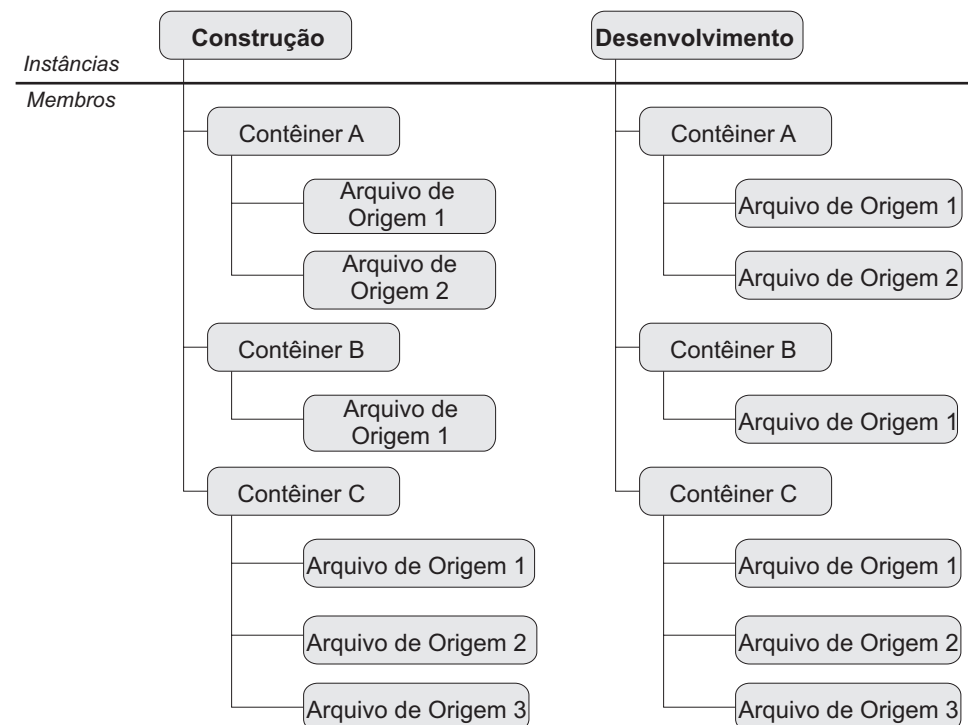


Figura 2. Exemplo de hierarquia do SCM

Registro de Entrada e Saída

O CARMA fornece uma interface genérica entre diversos SCMs, cada um podendo manipular operações de formas diferentes. Como não é possível prever se a operação de registro de entrada ou saída de um determinado SCM respectivamente esperará ou retornará o conteúdo de um membro, o CARMA foi projetado de um modo que as ações de registro de entrada e saída sejam operações de configuração de sinalizador. Ou seja, nenhum conteúdo de membro é passado ao SCM ou retornado dele como parte das ações de registro de entrada e saída.

Alguns SCMs podem esperar que o conteúdo de um membro seja passado durante uma operação de registro de entrada para esse membro. Um RAM para um SCM desse tipo deve tratar esse caso armazenando o conteúdo do membro em um local temporário antes de fazer a chamada de registro de entrada para o SCM.

Da mesma forma, alguns SCMs poderão retornar o conteúdo de um membro durante uma operação de registro de saída para esse membro. Um RAM para um SCM desse tipo deve tratar esse caso armazenando o conteúdo do membro em um local temporário até que o cliente recupere o conteúdo.

Alocação de Memória

Muitas das funções de API do CARMA exigem que o RAM ou o cliente CARMA aloque memória para armazenar resultados ou parâmetros de função que são passados entre o RAM e o cliente CARMA. Para todas as funções diferentes de `extractMember` e `putMember`, uma matriz unidimensional precisará ser alocada pelo RAM e liberada pelo cliente para armazenar conjuntos de informações de instância, membro e outras. O diagrama a seguir ilustra como o RAM deve alocar essa matriz:

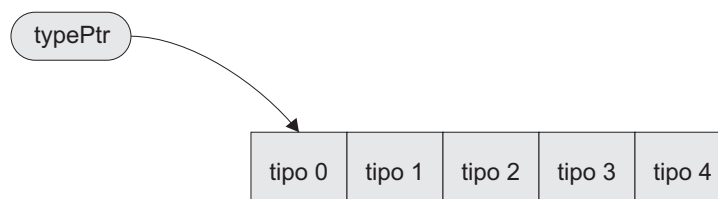


Figura 3. Matriz unidimensional simples conforme seria alocada por um RAM

Cada elemento na matriz representada acima é do tipo de estrutura de dados `type`. `typePtr` é um ponteiro `type` (do tipo `type*`) que atua como uma manipulação para a memória alocada recentemente. Em C, essa memória pode ser alocada com o seguinte código:

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

em que `numElements` é o número de índices de matriz que precisam ser criados. A memória para a qual `typePtr` aponta deve ser liberada pelo cliente depois que não é mais necessária.

As funções `putMember` e `extractMember` usam matrizes bidimensionais para transferir conteúdo de membro, com cada linha de matriz contendo um dos registros de membro. Para `extractMember`, o RAM deve alocar a matriz e o cliente CARMA deve liberar a matriz. Para `putMember`, o cliente CARMA deve alocar e liberar a matriz. Em ambos os casos, a matriz deve ser alocada conforme ilustrado

no diagrama a seguir:

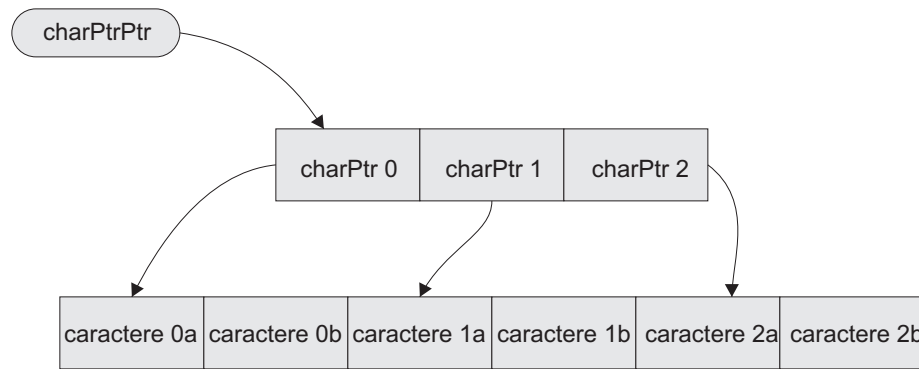


Figura 4. Matriz de caracteres bidimensional conforme usada em *extractMember* e *putMember*

`charPtrPtr` é um ponteiro para um ponteiro `char` (é do tipo `char**`) que serve de manipulação para uma matriz de ponteiros `char` (elementos do tipo `char*`). Os dados da matriz de caracteres bidimensional na verdade são armazenados em uma matriz de caracteres unidimensional; a ideia de linhas e colunas é puramente conceitual. A matriz de ponteiros `char` é usada para fornecer manipulações ao primeiro elemento de cada linha da matriz "bidimensional". Assim, na ilustração, a primeira linha da matriz bidimensional consiste nos elementos 0a e 0b, com 0a sendo o primeiro elemento dessa linha; a segunda linha consiste nos elementos 1a e 1b, com 1a sendo o primeiro elemento dessa linha; e assim por diante.

Para alocar uma matriz bidimensional como as que são requeridas para as funções *extractMember* e *putMember*, o cliente CARMA deve primeiro criar `charPtrPtr`. Em C, use a seguinte declaração:

```
char** charPtrPtr;
```

Se o cliente CARMA estiver alocando a matriz de caracteres bidimensional (como é o caso da função *putMember*) a matriz agora poderá ser alocada. Em C, o cliente CARMA deve usar o seguinte código:

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

em que `numRows` é o número de linhas e `numColumns` é o número de colunas na matriz bidimensional. A primeira linha aloca a matriz de ponteiros `char` (um ponteiro para cada linha na matriz bidimensional), a segunda linha aloca a matriz que mantém os dados da matriz bidimensional e o loop `for` designa cada ponteiro `char` na matriz de ponteiros `char` a uma linha na matriz bidimensional.

Se o RAM estiver alocando a matriz de caracteres bidimensional (como é o caso da função *extractMember*) uma etapa adicional será requerida para que a matriz possa ser alocada: `charPtrPtr` precisa ser passado por referência ao RAM como o parâmetro `contents` de *extractMember*, ou seja, um ponteiro para `charPtrPtr` precisa ser passado. Isso é necessário para que o cliente tenha uma manipulação para a matriz bidimensional depois que o RAM alocou a matriz. Suponha que o RAM receba um parâmetro denominado `contents` do tipo `char***` na função do RAM que alocará a matriz bidimensional. O RAM deve então alocar a matriz bidimensional, usando `contents` como manipulação para a matriz. Em C, o RAM deve usar o seguinte código para alocar a matriz bidimensional:

Tabela 3. Níveis de rastreio. Mensagens no nível de rastreio "Nenhum" não são registradas. (continuação)

Enumeração	Nível de Rastreio
1	Informações
3	Depurar

Todas as mensagens no nível escolhido e abaixo dele serão registradas. Por exemplo, se o nível de rastreio "Informações" for escolhido, os seguintes tipos de mensagens serão registrados: informações, aviso e erro. Informações adicionais sobre criação de log são discutidas em "Log" na página 20 (para desenvolvimento do RAM) e em "Log" na página 84 (para desenvolvimento do cliente CARMA).

Parâmetros e Valores de Retorno Customizados

Parâmetros e valores de retorno customizados são referenciados por elementos nas matrizes de ponteiros void. Como parâmetros e valores de retorno podem ser de vários tipos de dados, os ponteiros para eles são estereotipados como void* e, em seguida, armazenados em uma única matriz. Cada matriz assim mantém o parâmetro ou os valores de retorno customizados, mas nunca ambos. O diagrama a seguir ilustra a estrutura de uma matriz de parâmetros customizados de exemplo:

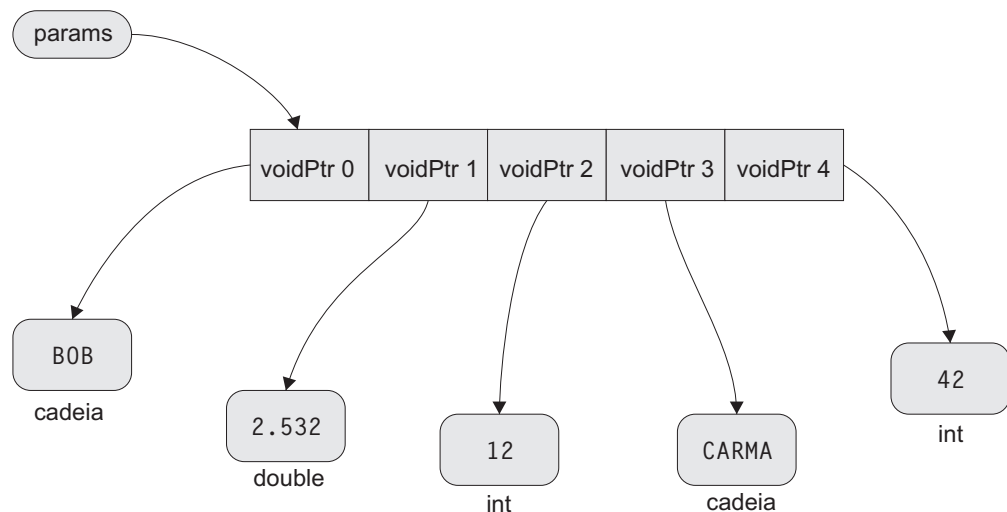


Figura 7. Exemplo de matriz de parâmetro customizado. Cada elemento na matriz é um ponteiro para um parâmetro. O valor de cada parâmetro é mostrado e rotulado com seu tipo de dados.

em que params é um ponteiro para uma matriz void e cada voidPtr na matriz é um ponteiro void que aponta para um parâmetro. As matrizes de valor de retorno customizado devem ser estruturadas de forma semelhante.

O número de elementos que devem estar em uma matriz de parâmetro ou valor de retorno customizado depende das informações de CAF nos clusters do VSAM do CARMA (consulte "Criando Registros VSAM de um Modelo do RAM" na página 70). Como é responsabilidade do desenvolvedor do RAM incluir informações sobre parâmetros e valores de retorno customizados nos clusters do VSAM, o desenvolvedor do RAM já deverá saber quantos elementos incluir nas matrizes de parâmetros e valores de retorno customizados. Os desenvolvedores do cliente CARMA podem usar a função getCAFData ou getCAFData2 do CARMA para

recuperar informações sobre ações, parâmetros e valores de retorno customizados para um RAM (consulte “getCAFData” na página 110 para obter mais informações). Usando essas informações, os desenvolvedores do cliente CARMA podem determinar quantos parâmetros e valores de retorno customizados são necessários para cada ação do RAM.

Capítulo 3. Desenvolvendo um RAM

Os gerenciadores de acesso a repositório (RAMs) fornecem ao CARMA acesso a SCMs específicos. Um RAM é uma biblioteca vinculada dinamicamente (DLL) que exporta pontos de entrada para todas as funções de API que ele implementa. Uma referência de função de API foi incluída no final deste capítulo.

A maioria das funções do RAM tem o seguinte padrão:

1. Determinar a qual instância e/ou membro a solicitação se aplica
2. Contatar o SCM para executar a operação solicitada
3. Alocar qualquer memória necessária para retornar o resultado
4. Preencher a memória alocada com o resultado
5. Retornar o resultado ao CARMA

Você pode usar o arquivo de origem do RAM esqueleto, CRARAMSA (localizado na biblioteca de amostra), como ponto de início para o RAM se estiver desenvolvendo o RAM em C. Lembre-se de que o RAM deve seguir as diretrizes de estado, alocação de memória e implementação de API fornecidas neste documento; caso contrário, problemas graves poderão se desenvolver: o CARMA pode não se comunicar apropriadamente com o RAM; fugas de memória poderão se desenvolver ou, na pior das hipóteses, o CARMA ou o RAM poderá ser encerrado de forma anormal. Especificamente, leia as seguintes seções com atenção:

- “Alocação de Memória” na página 6
- “Funções de Estado” na página 24

Construção do RAM

A construção do RAM é um processo que difere do processo de construção de um módulo de carregamento ou objeto de programa normal. Por causa dos requisitos de suporte de DLL, o processo de criar o RAM para um PDS requer maior esforço do que para um PDS/E.

Construção de um PDS

O processo de criar um RAM em um PDS requer o uso do Prelinker. As etapas descritas na criação de um RAM para um PDS são estas:

1. Compilação
2. Pré-link
3. Link

A etapa de compilação requer que sejam fornecidas a cada origem as opções de compilação adequadas para produzir o código de objeto de DLL. A etapa de pré-link envolve alimentar o código do objeto no Prelinker. A saída do Prelinker é o código do objeto que é entrada válida para o vinculador. O Prelinker criará um deck paralelo que pode ser uma entrada obrigatória para que o vinculador resolva referências externas. A etapa de link final requer que o código do objeto e os decks paralelos sejam criados pelas etapas anteriores como entrada.

Para ajudar na execução de compilações envolvendo C, o procedimento da JCL CRACPL é fornecido na biblioteca de amostra do CARMA. A JCL de amostra para criar um RAM em um PDS também é fornecida nos membros CRA#CRAM e

CRA#PRAM. CRA#CRAM é compilado para PDS/E, enquanto CRA#PRAM pode ser compilado para PDS ou PDS/E. Somente CRA#PRAM (ou outra compilação para a JCL do PDS) requer CRACPL.

Construção de um PDS/E

O processo de criar um RAM em um PDS/E envolve duas etapas. A saída desse processo é um objeto de programa.

1. Compilação
2. Ligação

A primeira etapa envolve o uso do compilador para gerar o código do objeto para o RAM. Depois que o código do objeto de todas as origens foi criado, ele pode ser alimentado no componente de ligação como entrada para gerar o objeto de programa do RAM.

O processo de criar um RAM em um PDS/E é mais simples do que o de criar um PDS. A JCL de exemplo é fornecida para criar os RAMs PDS, SCLM e COBOL em um PDS/E. A JCL de amostra faz uso de procedimentos padrão para executar os processos de compilação e ligação.

Notas:

1. RAMs escritos em C destinam-se apenas ao uso com o compilador z/OS XL C
2. RAMs escritos em COBOL destinam-se apenas ao uso com o compilador Enterprise COBOL para z/OS.

Usando o Módulo de Utilitários do RAM

As funções de utilitários do RAM são fornecidas como origem de amostra que pode ser compilada para uso de qualquer RAM designado para trabalhar com o CARMA. Ela fornece acesso a métodos que são frequentemente requeridos pelos designers de RAM e com frequência são chamados diversas vezes dentro de um único RAM. Usando o módulo de utilitários do RAM e sua biblioteca de funções, os desenvolvedores poderão economizar um tempo enorme e simplificar a execução das operações do CARMA nos membros PDS.

Os métodos a seguir estão incluídos no módulo de utilitários do RAM:

utilInitMemberList

Esse método inicializa uma lista de membros do PDS especificado. Ele deve ser chamado antes de chamadas para utilGetNextMember serem feitas. Uma chamada para utilCloseMemberList também deverá ser feita antes da próxima chamada para utilInitMemberList se utilInitMemberList retornar 0 para sucesso.

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

char pds[44]	Entrada	O PDS especificado cujos membros listar
int* count	Saída	O número de membros no PDS
void** tempDataPtr	Saída	Informações de estado armazenadas para uso do módulo, criadas por essa chamada

utilGetNextMember

Esse método coloca o próximo membro no PDS especificado por utilInitMemberList no membro. utilGetNextMember retorna 0 para sucesso, 1 para nenhum membro restante e qualquer outro valor no caso de erro.

utilCloseMemberList deve ser chamado ao concluir a leitura da lista de membros para evitar fugas de memória. Se utilGetNextMember retornar algo diferente de 0 ou 1, você não terá de chamar utilCloseMemberList.

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

char member[8]	Saída	O próximo membro no PDS (espaço preenchido se nenhum membro existir)
void** tempDataPtr	Saída	Informações de estado armazenadas para uso do módulo, modificadas por essa chamada

utilCloseMemberList

Esse método limpa a lista de membros PDS criada por utilInitMemberList. Ele deve ser chamado antes de outro utilInitMemberList ser chamado.

```
void utilCloseMemberList(void** tempDataPtr)
```

void** tempDataPtr	Entrada	Informações de estado armazenadas para uso do módulo, limpas por essa chamada
--------------------	---------	---

utilGetAllMemberInfo

Esse método retorna os seguintes metadados mantidos pelo ISPF disponíveis para o determinado membro PDS. Esses metadados incluem:

- Conjunto de Dados
- Versão
- Nível de Modificação
- Data de Criação
- Dados de Modificação
- Hora de Modificação
- Tamanho Atual
- Tamanho Inicial
- Número de Registros Modificados

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

char pds[44]	Entrada	O PDS que contém o membro
char member[8]	Entrada	O nome do membro
memberInfo* output	Saída	As informações do membro são colocadas nessa estrutura

utilGetMemberInfo

Esse método retorna parte dos metadados mantidos pelo ISPF disponíveis para o determinado membro PDS, incluindo os tipos listados no método "utilGetAllMemberInfo".

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

char pds[44]	Entrada	O PDS que contém o membro
char member[8]	Entrada	O nome do membro
char* info	Saída	Um buffer grande o suficiente para conter as informações. U_ISPF_MI_SIZE[ukey] indicará o tamanho necessário para uma determinada chave. Não será por terminação NULA, mas o espaço deve ser preenchido com o tamanho especificado em U_ISPF_MI_SIZE.
int ukey	Entrada	Chave para as informações desejadas. Consulte o arquivo de cabeçalho Módulo de Utilitários do RAM para obter uma lista completa de chaves.

utilSetMemberInfo

Esse método permite que todos os metadados mantidos pelo ISPF sejam definidos. Os metadados que podem ser configurados incluem os tipos listados no método "utilGetAllMemberInfo".

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

char pds[44]	Entrada	O PDS que contém o membro
char member[8]	Entrada	O nome do membro
char info[10]	Entrada	As novas informações. Ukey_ZLLIB e Ukey_ZLMSEC não são suportados
int ukey	Entrada	Chave para as informações desejadas. Consulte o arquivo de cabeçalho Módulo de Utilitários do RAM para obter uma lista completa de chaves.

utilGetAllPDSInfo

Esse método retorna todos os metadados do ISPF disponíveis para o determinado PDS.

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

char pds[44]	Entrada	O PDS sobre o qual obter todas as informações
--------------	---------	---

pdsInfo* output	Saída	As informações do PDS serão colocadas nessa estrutura
-----------------	-------	---

utilCopyPDStoPDS

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

char fromInstanceID[44]	Entrada	O PDS do qual copiar
char frommemberID[8],	Entrada	O membro PDS a ser copiado
char toInstanceID[44]	Entrada	O PDS para o qual copiar
char tomemberID[8]	Entrada	O membro PDS a ser substituído (ou criado se não existir)

utilCopyPDStoSDS

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

char fromInstanceID[44]	Entrada	O PDS do qual copiar
char frommemberID[8],	Entrada	O membro PDS a ser copiado
char toInstanceID[44]	Entrada	O SDS para o qual copiar (este deve existir)

utilCopySDStoPDS

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

char fromInstanceID[44]	Entrada	O SDS do qual copiar
char toInstanceID[44]	Entrada	O PDS para o qual copiar
char tomemberID[8]	Entrada	O membro PDS a ser substituído (ou criado se não existir)

utilCopySDStoSDS

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

char fromInstanceID[44]	Entrada	O SDS do qual copiar
char toInstanceID[44]	Entrada	O SDS para o qual copiar (este deve existir)

utilPutMemberInit

Iniciará uma colocação em um membro PDS. Chame utilPutMemberRecs ou utilPutMemberRec até que todos os registros requeridos sejam colocados.

```
int utilPutMemberInit(char pds[44], char member[8], int* lrec1)
```

char pds[44]	Entrada	O PDS de destino
--------------	---------	------------------

char member[8]	Entrada	O membro PDS de destino
int* lrecl	Saída	O lrecl, ou o tamanho do registro para o determinado PDS. (Para VB, esse será o tamanho de registro máximo.)

utilPutMemberRecs

Coloque diversos registros em um comprimento fixo.

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

char** contents	Entrada	Matriz 2-D de registros (de tamanho lrecl) a ser colocada.
int numRecords	Entrada	O número de registros em um conteúdo de membros

utilPutMemberRec

Coloque um único registro de comprimento variável.

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

char* contents	Entrada	Um único registro a ser colocado
int length	Entrada	O comprimento do registro a ser colocado. (máximo de lrecl)

utilPutMemberClose

Deve ser chamado para cada utilPutMemberInit, exceto no caso de uma condição de erro em utilPutMemberInit, utilPutMemberRec ou utilPutMemberRecs.

```
int utilPutMemberClose()
```

utilExtractMemberInit

Configure o membro PDS do qual extrair.

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

char pds[44]	Entrada	O PDS de origem
char member[8]	Entrada	O membro PDS de origem
int* lrecl	Saída	O lrecl, ou o tamanho do registro para o determinado PDS. (Para VB, esse será o tamanho de registro máximo.)
int* recFM	Saída	Um Sinalizador representando o formato de registro. As opções são U_RECFM_VB, U_RECFM_FB e U_RECFM_U.

int* numRecords	Saída	O número de registros no membro PDS. Como isso usa estatísticas do ISPF para determinar o número de registros, o valor máximo é 65535 e só será exato se as estatísticas estiverem corretas. utilExtractMemberRec retorna 1 se fora dos registros e deve ser usado para determinar com precisão quando parar de extrair.
-----------------	-------	---

Para um valor igual a 65535, o membro PDS pode ter realmente mais registros.

utilExtractMemberRec

Extrai o próximo registro.

Retorna 0 para sucesso e 1 para sem mais registro.

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

char* record	Saída	Um buffer de caracteres de tamanho lrecl, para o qual o próximo registro será extraído.
int* length	Saída	O número de caracteres gravados no registro.

Um valor de retorno igual a 1 indica não haver mais registros e nenhum registro foi retornado nessa chamada.

utilExtractMemberClose

Deve ser chamado para cada utilExtractMemberInit, exceto no caso de uma condição de erro em utilExtractMemberInit ou utilExtractMemberRec

```
int utilExtractMemberClose()
```

Informações adicionais sobre os métodos listados anteriormente podem ser encontradas no arquivo de cabeçalho do módulo de utilitários do RAM.

Definindo o RAM para o CARMA

O CARMA mantém suas informações do RAM em diversos clusters do VSAM, que devem ser preenchidos com registros para cada RAM no ambiente. Consulte Capítulo 4, “Customizando uma API do RAM Usando o CAF”, na página 61 para saber como inserir os registros apropriados ao RAM nesses clusters do VSAM. Se você não precisar customizar a API do RAM, o único registro que será necessário incluir no cluster do VSAM é o registro do RAM; você não precisará incluir registros de parâmetro, valor de retorno ou ação.

Exportando Funções

Quando o CARMA tenta carregar um RAM, ele espera poder carregar as funções da API do RAM explicitamente usando a função C `dllqueryfn`. Se estiver usando C, uma instrução `#pragma export` como aquela a seguir será usada para exportar cada função do RAM. O exemplo a seguir exporta a função `initRAM`:

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

IDs versus Nomes

Quando um membro, uma instância ou outro tipo de dados está sendo retornado do RAM para o CARMA, seu ID e nome de exibição normalmente são retornados. O ID deve identificar exclusivamente a entidade para o RAM. Seria sensato retornar o caminho absoluto de um membro (começando pelo contêiner de nível superior) no campo de ID para que o membro possa ser acessado facilmente pelo RAM quando futuras solicitações forem feitas. O nome de exibição é simplesmente o nome que deve ser exibido no cliente.

Estruturas de Dados Predefinidas pelo RAM

Em sua maioria, as funções do RAM são estruturas predefinidas para passar informações de volta ao CARMA.

A estrutura `Descriptor` consiste em um campo de caractere de nome de 64 bytes e um campo de caractere de ID de 256 bytes. É usado para descrever instâncias, contêineres e membros simples. A estrutura `KeyValPair` consiste em um campo de chave de 64 bytes e um campo de valor de 256 bytes. É usado para pares de valores de chaves de metadados. Essas estruturas são resumidas na Tabela 4 e na Tabela 5.

Tabela 4. Estrutura de dados `Descriptor`

Campo	Descrição
<code>char id[256]</code>	ID exclusivo para descrever a entidade
<code>char name[64]</code>	Nome de Exibição

Tabela 5. Estrutura de dados `KeyValPair`

Campo	Descrição
<code>char key[64]</code>	Um índice
<code>char value[256]</code>	Os dados

`CRAFCDEF`, um arquivo de cabeçalho C na biblioteca de amostra, deve ser incluído no código do RAM para que você possa usar essas estruturas de dados.

Log

O CARMA fornece aos RAMs um ponteiro para uma função de criação de log, um ponteiro para um arquivo de log e um nível de rastreo (consulte a Tabela 3 na página 9) na inicialização. O nível de rastreo deve ser usado para filtrar algumas mensagens que possam não interessar aos usuários. A função de criação de log obtém um buffer de caracteres de remetente de 16 bytes, um buffer de caracteres de mensagem de 256 bytes e o ponteiro de arquivo de log que é passado na inicialização. Segue um exemplo de chamada em C:

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

A tarefa em spool indicará o nome do log criado.

Lidando com Operações Não Suportadas

Se você estiver desenvolvendo um RAM que se comunica com um SCM que não suporta uma operação do CARMA, informe ao cliente que ela está desativada, modificando apropriadamente as informações do CAF do RAM (consulte o Capítulo 4, “Customizando uma API do RAM Usando o CAF”, na página 61). Você pode assumir que os clientes CARMA não chamarão ações marcadas como desativadas. Entretanto, conte ainda com a possibilidade de um cliente chamar uma ação desativada executando uma das duas seguintes ações:

1. Não implemente a função para a ação desativada e não inclua uma instrução `pragma export` para a função. Isso fará com que o CARMA retorne um código de retorno igual a 8 para qualquer cliente que solicita essa operação do RAM.
2. Implemente a função para a ação desativada para retornar simplesmente um código de retorno igual a 107. Inclua a instrução `#pragma export` para a função como normalmente você faria.

Manipulando Parâmetros e Valores de Retorno Customizados

Parâmetros customizados são passados ao RAM usando o parâmetro `void** params`. `params` é uma matriz de ponteiros `void` que apontam para variáveis de diversos tipos. Se esses parâmetros customizados tiverem de ser definidos como parâmetros obrigatórios para uma determinada função nos clusters do VSAM do CARMA (consulte Capítulo 4, “Customizando uma API do RAM Usando o CAF”, na página 61 para obter mais informações), deverá ser assumido que o cliente configurou `params` corretamente. Para recuperar os parâmetros, basta estereotipar as variáveis em `params` novamente aos seus tipos adequados. Observe como `params` usa um `char*` para sequências, em vez de um `char**`. Use o seguinte código C como exemplo:

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

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

Um ponteiro para uma matriz de valores de retorno customizados não alocados é passado ao RAM como `void*** customReturn`. Se valores de retorno customizados forem definidos nos clusters do VSAM do CARMA, o RAM deverá alocar memória para `customReturn` e preenchê-la apropriadamente. Como o cliente deve liberar a memória criada no RAM, é importante que os desenvolvedores do RAM aloquem memória para cada valor de retorno separadamente. O seguinte código C demonstra o retorno de `int`, `string` e `double`:

```
/* Estes são definidos no início */
int* return0;
char* return1;
double* return2;

/* Corpo do programa */

return0 = malloc(sizeof(int));
```

```

*return0 = 5;
return1 = malloc(sizeof(char) * 10);
memcpy(return1, "THE STRING", 10);
return2 = malloc(sizeof(double));
*return2 = 3.41;
/* Alocar e preencher a estrutura de valor de retorno */
*customReturn = malloc(sizeof(void*) * 3);
(*customReturn)[0] = (void*) return0;
(*customReturn)[1] = (void*) return1;
(*customReturn)[2] = (void*) return2;

```

Se nenhum valor de retorno estiver definido nos clusters do VSAM do CARMA, customReturn deverá ser configurado como NULL.

Metadados Definidos pelo CARMA

Extensão de Arquivo Especificada pelo RAM

O RAM fornece a capacidade de sugerir extensões de arquivo para recursos do CARMA nos clientes CARMA que usam o RAM. As extensões de arquivo fornecem ao cliente insight sobre o editor apropriado para uso com um recurso do CARMA específico. Permitir que o RAM especifique a extensão de arquivo elimina a necessidade de o usuário especificar extensões em cada recurso.

As extensões podem ser adquiridas de três fontes diferentes:

- O RAM
- O cliente
- Um contêiner-pai

O RAM pode ser configurado para sugerir extensões de arquivo ao cliente que possam ser usadas em conjunto com os recursos do CARMA. Por exemplo, supondo que a propriedade de metadados do RAM "carma.file-extension" esteja configurada como "foo" e o cliente esteja configurado para consultar o RAM em busca de uma extensão. O nome do arquivo para o recurso do CARMA "Name" seria exibido no cliente como "Name.foo". Isso se deve ao fato de que o CARMA consultará o RAM em busca de uma extensão de arquivo se o cliente estiver configurado para aceitar uma extensão do RAM. Por padrão, o RAM não sugere a extensão de arquivo. Entretanto, pode-se assumir que o cliente fornecerá uma extensão se uma não tiver sido fornecida ainda pelo RAM.

Tabela 6. Extensão de arquivo sugerida pelo RAM

Nome de Exibição	Propriedade de Metadados do RAM (carma.file-extension)	Propriedade de Extensão de Cliente (configurada para aceitar a sugestão do RAM)	Nome do Arquivo no Cliente
Nome	.foo	<unset>	Name.foo

Entretanto, depois que o RAM tiver especificado a extensão de arquivo, fica a critério do cliente aceitar a extensão de arquivo sugerida ou usar uma definida no cliente. No exemplo fornecido na Tabela 6, a extensão fornecida pelo RAM foi "foo", de modo que o recurso do CARMA "Name" foi exibido no cliente como "Name.foo". Supondo agora que o cliente foi configurado para não usar a extensão fornecida pelo RAM e aplicar uma sua própria. O arquivo "Name.foo" seria alterado para exibir "Name.ext", em que "ext" é a nova extensão especificada no

cliente. Se o nome de exibição já tiver uma extensão de arquivo associada a ele, o cliente não poderá removê-la; ele só poderá anexar uma nova extensão ao nome de arquivo existente.

Tabela 7. Extensão de arquivo especificada pelo cliente. O cliente substitui a extensão de arquivo sugerida pelo RAM e aplica a sua própria.

Nome de Exibição	Propriedade de Metadados do RAM (carma.file-extension)	Propriedade de Extensão de Cliente (configurada para ignorar a sugestão do RAM)	Nome do Arquivo no Cliente
Nome	.foo	.ext	Name.ext
Name.foo	<unset>	.ext	Name.foo.ext

Se uma extensão de arquivo não estiver predefinida na propriedade de metadados do RAM (carma.file-extension = <unset>), um recurso do CARMA direcionará a si mesmo para o cliente para obter uma extensão. Se o cliente não especificar uma extensão de arquivo também, o recurso do CARMA herdará a extensão padrão de seu contêiner-pai.

Tabela 8. Herança de extensão de arquivo. Uma extensão de arquivo não é especificada em nenhum nível, por isso o recurso herda uma extensão de seu pai. Uma extensão "dft" representa a extensão padrão de um pai conforme determinada pelo cliente CARMA.

Nome de Exibição	Propriedade de Metadados do RAM (carma.file-extension)	Propriedade de Extensão de Cliente	Nome do Arquivo no Cliente
Nome	<unset>	<unset>	Name.dft

Versão do CARMA

O RAM fornece a capacidade de rastrear todas as versões disponíveis dos membros do CARMA com o uso de uma chave de metadados específica: carma.version. Ao fornecer a chave carma.version na lista de informações do membro, o CARMA possibilita oferecer funcionalidade específica para recursos com versão. Por exemplo, os membros do CARMA que suportam rastreamento de versão podem diferir dos membros que não suportam. As ações disponíveis nos membros ativados para versão dependem do SCM do qual o membro se origina, bem como do RAM usado para conectar-se ao SCM. Cabe ao desenvolvedor do RAM decidir que ações ativar, como tornar as versões editáveis, somente leitura, ou fornecer acesso a versões anteriores. Quando o CARMA executa funções nos membros que foram ativados para versão, por padrão, as funções sempre farão referência à versão mais recente de um membro, a menos que o contrário seja especificado.

O uso da chave de informações do membro não identifica exclusivamente o membro do CARMA. Cada membro do CARMA com versão deve ter um ID de membro exclusivo a fim de indicar qual versão está sendo considerada especificamente. Por exemplo, um membro do CARMA com o ID "member1" tem 2 versões – 1 e 2. As versões do membro podem ser identificadas exclusivamente, anexando um número de versão ao ID. Consulte o exemplo na Tabela 9 na página 24. O RAM deve estar apto a identificar exclusivamente a versão do membro com base no ID a fim de oferecer funcionalidade para suportar membros com versão – como checkin, extractMember, performAction, etc.

Tabela 9. Exemplo de versão de membro do CARMA.

Versão do Membro	Exemplo de ID do Membro
Versão 1	member1_v1
Versão 1.1	member1_v1.1
Versão 2	member1_v2

Para obter informações detalhadas sobre a chamada de listas de versões para membros do CARMA, consulte a seção “getVersionList” na página 47.

Funções de Estado

O RAM tem três funções de estado: `initRAM`, `terminateRAM` e `reset`, conforme ilustrado na Figura 8. `initRAM` inicializa as variáveis globais do RAM e estabelece a conexão com o repositório. Não será possível chamá-la novamente em uma sessão até que o RAM tenha sido finalizado. `reset` restaura a conexão do repositório com seu estado inicial. É possível chamá-la a qualquer momento, exceto imediatamente após `terminateRAM`. `terminateRAM` também pode ser chamada a qualquer momento, mas a única função que pode ser chamada com êxito imediatamente após `terminateRAM` é `initRAM`.

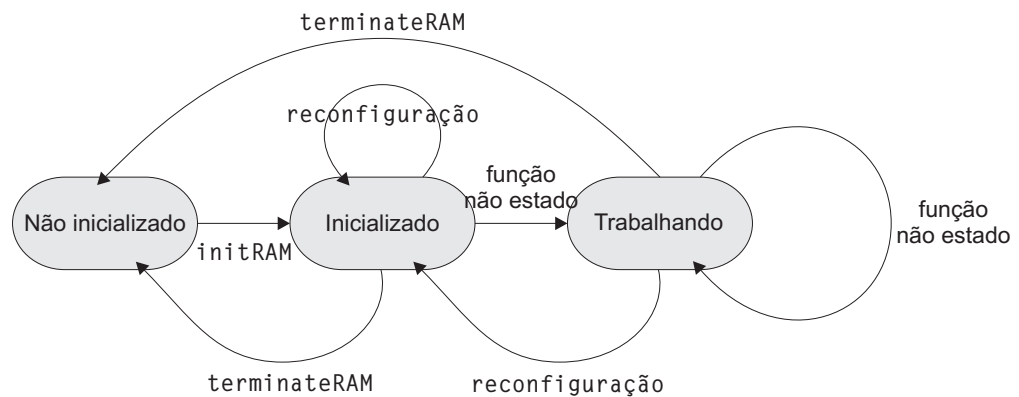


Figura 8. Diagrama de estado do RAM

initRAM

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

Log_Func logFunc	Entrada	Um ponteiro de função para a função de criação de log do CARMA. Isso deve ser armazenado para uso em outras funções do RAM.
FILE* log	Entrada	Um ponteiro de arquivo para o log do CARMA. Isso deve ser armazenado para uso com a função de criação de log.
int traceLev	Entrada	O nível de rastreo de criação de log a ser usado em toda a sessão.

char locale[8]	Entrada	Informa ao CARMA o código de idioma das sequências que serão retornadas ao cliente
char codepage[5]	Entrada	Informa ao CARMA a página de códigos das sequências que serão retornadas ao cliente
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

initRAM deve ser chamado antes que todas as outras operações do RAM ocorram. Deve ser usado para inicializar a conexão do SCM e configurar qualquer variável global usada no programa. Entre essas variáveis globais devem estar aquelas usadas para armazenar as três variáveis passadas nessa função.

terminateRAM

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

char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.
-----------------	-------	--

terminateRAM deve ser usado para fechar a conexão do SCM e liberar os recursos usados pelo RAM (como memória e arquivos).

reset

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

char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.
-----------------	-------	--

reset é usado para restaurar a conexão SCM com seu estado inicial.

Funções de Navegação

getInstances

Recupera a lista de instâncias disponíveis no SCM

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

Descriptor** records	Saída	Isso deve ser alocado e preenchido com os IDs e os nomes das instâncias disponíveis.
int* numRecords	Saída	O número de registros que foram alocados e retornados

<code>void** params</code>	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
<code>void*** customReturn</code>	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
<code>char filter[256]</code>	Entrada	Isso pode ser passado do cliente para filtrar conjuntos de instâncias
<code>char error[256]</code>	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Operação:

1. Consulte o SCM a respeito de sua lista de instâncias, possivelmente aplicando um filtro.
2. Aloque a matriz `records`. Caso esteja desenvolvendo um RAM em C, use o seguinte código:

```
*records = (Descriptor*) malloc(sizeof(Descriptor) * *numRecords);
```
3. Preencha a matriz `records` com os IDs e os nomes.

Se não for possível consultar o SCM a respeito das instâncias, pode ser útil o cliente passar uma lista de instâncias conhecidas usando o buffer `filter`. O RAM deve então verificar a lista e retornar as instâncias na matriz de registros. As instâncias poderão ser codificadas permanentemente se foram constantes para o SCM.

getInstancesWithInfo

Recupera a lista de instâncias disponíveis no SCM e os metadados associados a essas instâncias. Essa é uma função opcional. O RAM deverá implementar essa função se o SCM fornecer os metadados com a lista de instâncias. O RAM é obrigado a implementar `getInstances` mesmo que `getInstancesWithInfo` esteja implementado.

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

<code>DescriptorWithInfo** records</code>	Saída	Isso deve ser alocado e preenchido com os IDs e os nomes das instâncias disponíveis. Para cada instância, as informações de metadados devem ser alocadas e preenchidas.
<code>int* numRecords</code>	Saída	O número de registros que foram alocados e retornados

<code>void** params</code>	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
<code>void*** customReturn</code>	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
<code>char filter[256]</code>	Entrada	Isso pode ser passado do cliente para filtrar conjuntos de instâncias
<code>char error[256]</code>	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Operação:

1. Consulte o SCM a respeito de sua lista de instâncias e metadados, possivelmente aplicando um filtro.
2. Aloque a matriz de registros. Caso esteja desenvolvendo um RAM em C, use o seguinte código:

```
*records = (DescriptorWithInfo*)
            malloc(sizeof(DescriptorWithInfo) * *numRecords);
```

3. Para cada registro na matriz, proceda da seguinte forma:
 - a. Preencha o ID e o nome da instância
 - b. Configure o `infoCount` com o número de pares de valores de chaves de metadados associados a essa instância.
 - c. Aloque a matriz `info`. Caso esteja desenvolvendo um RAM em C, use o seguinte código, em que `i` é o índice baseado em zero da instância com a qual você está trabalhando:

```
(*records)[i].info = (KeyValPair *)
                    malloc(sizeof(KeyValPair) * (*records)[i].infocount);
```
 - d. Preencha a matriz de informações com os pares de valores de chaves de metadados para a instância.

Se não for possível consultar o SCM a respeito das instâncias, pode ser útil o cliente passar uma lista de instâncias conhecidas usando o buffer `filter`. O RAM deve então verificar a lista e retornar as instâncias na matriz de registros. As instâncias poderão ser codificadas permanentemente se foram constantes para o SCM.

Nota: Se a função `getInstancesWithInfo` precisar de parâmetros customizados ou devoluções customizadas, configure a função para usar os mesmos parâmetros customizados e devoluções customizadas como a função `getInstances`. Isso permite que o cliente CARMA chame a função apropriada sem entrada adicional do usuário.

getMembers

Recupera an lista de membros em uma instância

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

char instanceID[256]	Entrada	A instância para a qual os membros devem ser retornados
Descriptor** members	Saída	Isso deve ser alocado e preenchido com os IDs e os nomes dos membros dentro da instância.
int* numRecords	Saída	O número de membros para os quais a matriz foi alocada
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char filter[256]	Entrada	Isso pode ser passado do cliente para filtrar conjuntos de membros.
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Operação:

1. Consulte o SCM a respeito dos membros da instância determinada, possivelmente aplicando um filtro.
2. Aloque a matriz members. Caso esteja desenvolvendo um RAM em C, use o seguinte código:

```
*members = malloc(sizeof(Descriptor) * *numRecords);
```
3. Preencha a matriz members com os IDs e nomes dos membros.

getMembersWithInfo

Recupera a lista de membros disponíveis na instância especificada e os metadados associados a esses membros. Essa é uma função opcional. O cliente deverá chamar essa função se desejar recuperar uma lista de membros e os metadados associados a todos esses membros. Se o RAM não suportar essa função, o cliente deverá fazer fallback para chamar getMembers.

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

char instanceID[256]	Entrada	A instância para a qual os membros devem ser retornados
----------------------	---------	---

MemberDescriptorWithInfo** members	Saída	Isso deve ser alocado e preenchido com os IDs e os nomes dos membros dentro da instância. Para cada membro, as informações de metadados devem ser alocadas e preenchidas. Deve ser indicado também se o membro é um contêiner.
int* numRecords	Saída	O número de membros para os quais a matriz foi alocada
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char filter[256]	Entrada	Isso pode ser passado do cliente para filtrar conjuntos de membros.
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Operação:

1. Consulte o SCM a respeito dos membros da instância determinada e os metadados associados, possivelmente aplicando um filtro.
2. Aloque a matriz members. Caso esteja desenvolvendo um RAM em C, use o seguinte código:

```
*members = (MemberDescriptorWithInfo*)
             malloc(sizeof(MemberDescriptorWithInfo) * *numRecords);
```
3. Para cada membro na matriz, proceda da seguinte forma:
 - a. Preencha o ID e o nome do membro
 - b. Configure se o membro é um contêiner. Configure o campo isContainer como 1 se for um contêiner; e 0 se não for.
 - c. Configure o campo infoCount com o número de pares de valores de chaves de metadados associados a esse membro.
 - d. Aloque a matriz info. Caso esteja desenvolvendo um RAM em C, use o seguinte código, em que i é o índice baseado em zero do membro com o qual você está trabalhando:

```
*members[i].info = (KeyValPair *)
                   malloc(sizeof(KeyValPair) * (*members)[i].infoCount);
```
 - e. Preencha a matriz de informações com os pares de valores de chaves de metadados para o membro.

Nota: Se a função getMembersWithInfo precisar de parâmetros customizados ou devoluções customizadas, configure a função para usar os mesmos

parâmetros customizados e devoluções customizadas como a função `getMembers`. Isso permite que o cliente CARMA chame a função apropriada sem entrada adicional do usuário.

isMemberContainer

Configura `isContainer` como `true` se um membro for um contêiner; caso contrário, `false`

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

char instanceID[256]	Entrada	A instância que contém o membro sendo verificado
char memberID[256]	Entrada	A instância que está sendo verificada
int* isContainer	Saída	Deve ser configurado como 1 se o membro for um contêiner; e como 0, se não for
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Configure `*isContainer` como 1 se o membro for um contêiner; ou 0, se não for.

getContainerContents

Recupera a lista de membros disponíveis em um contêiner

```
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]	Entrada	A instância que mantém o contêiner
char memberID[256]	Entrada	O ID do contêiner
Descriptor** contents	Saída	Deverá ser alocado e preenchido com os IDs e os nomes dos membros dentro do contêiner
int* numRecords	Saída	O número de membros para os quais a matriz foi alocada

<code>void** params</code>	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
<code>void*** customReturn</code>	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
<code>char filter[256]</code>	Entrada	Isso pode ser passado do cliente para filtrar conjuntos de membros.
<code>char error[256]</code>	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Operação:

1. Consulte o SCM a respeito dos membros do contêiner determinado, possivelmente aplicando um filtro.
2. Aloque a matriz `contents`. Caso esteja desenvolvendo um RAM em C, use o seguinte código:

```
*contents = malloc(sizeof(Descriptor) * *numMembers);
```
3. Preencha a matriz `contents` com os IDs e nomes dos membros.

getContainerContentsWithInfo

Recupera a lista de membros em um contêiner e os metadados associados a esses membros. Essa é uma função opcional. O RAM deverá implementar essa função se o SCM fornecer os metadados com a lista de membros. O RAM é obrigado a implementar `getContainerContents` mesmo que `getContainerContentsWithInfo` esteja implementado.

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

<code>char instanceID[256]</code>	Entrada	A instância que mantém o contêiner
<code>char memberID[256]</code>	Entrada	O ID do contêiner
<code>MemberDescriptorWithInfo** members</code>	Saída	Isso deve ser alocado e preenchido com os IDs e os nomes dos membros dentro do contêiner. Para cada membro, as informações de metadados devem ser alocadas e preenchidas. Deve ser indicado também se o membro é um contêiner
<code>int* numRecords</code>	Saída	O número de membros para os quais a matriz foi alocada

<code>void** params</code>	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
<code>void*** customReturn</code>	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
<code>char filter[256]</code>	Entrada	Isso pode ser passado do cliente para filtrar conjuntos de membros.
<code>char error[256]</code>	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Operação:

1. Consulte o SCM a respeito dos membros do contêiner determinado e os metadados associados, possivelmente aplicando um filtro.
2. Aloque a matriz de membros. Caso esteja desenvolvendo um RAM em C, use o seguinte código:

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

3. Para cada membro na matriz, proceda da seguinte forma
 - a. Preencha o ID e o nome do membro
 - b. Configure se o membro é um contêiner. Configure o campo `isContainer` como 1 se for um contêiner; e 0 se não for.
 - c. Configure o campo `infoCount` com o número de pares de valores de chaves de metadados associados a esse membro.
 - d. Aloque a matriz de informações. Caso esteja desenvolvendo um RAM em C, use o seguinte código, em que `i` é o índice baseado em zero do membro com o qual você está trabalhando:


```
(*members)[i].info = (KeyValPair *)
                    malloc(sizeof(KeyValPair) * (*members)[i].infocount);
```
 - e. Preencha a matriz `info` com os pares de valores de chaves de metadados para o membro.

Nota: Se a função `getContainerContentsWithInfo` precisar de parâmetros customizados ou devoluções customizadas, configure a função para usar os mesmos parâmetros customizados e devoluções customizadas como a função `getContainerContents`. Isso permite que o cliente CARMA chame a função apropriada sem entrada adicional do usuário.

Criar/Excluir

Criar e excluir fornecem a funcionalidade para criar e excluir membros e contêineres em um ambiente do CARMA.

createMember

Cria um novo membro


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

char instanceID[256]	Entrada	A instância que contém o membro sendo criado
char memberID[256]	Saída	O ID do membro que está sendo criado
char name[64]	Entrada/Saída	O ID do membro sendo criado
char parentID[256]	Entrada	O ID do contêiner-pai (Se nenhum pai existir, deve ser preenchido com espaço)
int* lrecl	Saída	O número de colunas no conjunto de dados e na matriz
char recFM[4]	Saída	Contém o formato de registro do conjunto de dados (FB, VB, ect)
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

createContainer

Cria um novo contêiner

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

char instanceID[256]	Entrada	A instância que contém o contêiner sendo criado
char memberID[256]	Saída	O ID do contêiner que está sendo criado
char name[64]	Entrada/Saída	O ID do contêiner sendo criado
char parentID[256]	Entrada	O ID do contêiner-pai (Se nenhum pai existir, deve ser preenchido com espaço)

void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

delete

Exclui um membro ou contêiner

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

char instanceID[256]	Entrada	A instância que contém o membro ou o contêiner sendo excluído
char memberID[256]	Entrada	O ID do membro que está sendo excluído
int force	Entrada	Usado para forçar uma exclusão. Um valor igual a 1 forçará uma exclusão
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

A função de exclusão pode ser usada para excluir membros e contêineres; entretanto, não deve ser usada para excluir uma Instância do RAM.

Funções de Transferência de Arquivo

extractMember

Recupera o conteúdo do membro

```
int extractMember(char instanceID[256], char memberID[256],
    char*** contents, int* lrec1, int* numRecords,
    char recFM[4], int* moreData, int* nextRec,
    void** params, void*** customReturn, char error[256])
```

char instanceID[256]	Entrada	A instância que contém o membro
char memberID[256]	Entrada	O ID do membro que está sendo extraído
char*** contents	Saída	Será alocado como uma matriz bidimensional para manter o conteúdo do membro
int* lrec1	Saída	O número de colunas no conjunto de dados e na matriz
int* numRecords	Saída	O número de registros no conjunto de dados ou o número de linhas na matriz
char recFM[4]	Saída	Conterá o formato de registro do conjunto de dados (FB, VB, etc.)
int* moreData	Saída	Configure como 1 o valor da variável para o qual isso aponta, se a extração tiver de ser chamada novamente (por ainda haver mais dados a serem extraídos). Caso contrário, designe como 0 o valor para o qual ela aponta.
int* nextRec	Entrada/Saída	Entrada: O registro do membro no qual o RAM deve iniciar a extração Saída: O primeiro registro no conjunto de dados que não foi extraído se *moreData estiver configurado como 1; caso contrário, indefinido
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

extractMember retorna o conteúdo do conjunto de dados em uma matriz bidimensional. A função foi projetada para suportar o envio dos dados em chunks, por isso a matriz não precisa ser alocada para o tamanho inteiro do arquivo. Os registros nos conjuntos de dados são considerados para serem indexados com o primeiro registro sendo 0.

Operação:

1. Determine quantos registros estão no conjunto de dados, qual é o `lrec1` e os formatos de registro e configure `*lrec1` e `recFM`.
 - a. Se `*numRecords - nextRec` for maior que o tamanho do chunk de dados do RAM, configure `*numRecords` com o número de registros do chunk de dados e configure `*moreData` como 1; por último, aloque a matriz.
 - b. Caso contrário, configure `*numRecords` como `*numRecords - *nextRec` e aloque a matriz. Caso esteja desenvolvendo um RAM em C, use o seguinte código:

```
*contents = (char**) malloc(sizeof(char*) * (*numRecords));
**contents = (char*) malloc(sizeof(char) * (*lrec1) * (*numRecords));
for(i = 0; i < *numRecords; i++)
    (*contents)[i] = ( (**contents) + (i * (*lrec1)) );
```
2. Preencha a matriz com o conjunto esperado de registros. Assegure-se de que os registros não tenham terminação nula. Se houver mais dados a serem retornados, configure `*nextRec` como índice baseado em 0 do próximo registro.

Exemplo

Configuração: O membro contém 26 registros, cada um contendo o próximo caractere alfabético, começando com "A" no registro 0. Seu valor `*lrec1` é 5, seu valor `recFM` é "FB" e o tamanho do chunk de dados do RAM é 10.

A Figura 9 na página 37 mostra o que `extractMember` deve retornar para cada chamada necessária para extrair todo o conteúdo.

Primeira Chamada	Segunda Chamada	Terceira Chamada																																																																																																																																		
<table><tr><td>A</td><td></td><td></td><td></td><td></td></tr><tr><td>B</td><td></td><td></td><td></td><td></td></tr><tr><td>C</td><td></td><td></td><td></td><td></td></tr><tr><td>D</td><td></td><td></td><td></td><td></td></tr><tr><td>E</td><td></td><td></td><td></td><td></td></tr><tr><td>F</td><td></td><td></td><td></td><td></td></tr><tr><td>G</td><td></td><td></td><td></td><td></td></tr><tr><td>H</td><td></td><td></td><td></td><td></td></tr><tr><td>I</td><td></td><td></td><td></td><td></td></tr><tr><td>J</td><td></td><td></td><td></td><td></td></tr></table> <p>*lrec1 = 5 *numRecords = 10 *moreData = 1 *nextRec = 10</p>	A					B					C					D					E					F					G					H					I					J					<table><tr><td>K</td><td></td><td></td><td></td><td></td></tr><tr><td>L</td><td></td><td></td><td></td><td></td></tr><tr><td>M</td><td></td><td></td><td></td><td></td></tr><tr><td>N</td><td></td><td></td><td></td><td></td></tr><tr><td>O</td><td></td><td></td><td></td><td></td></tr><tr><td>P</td><td></td><td></td><td></td><td></td></tr><tr><td>Q</td><td></td><td></td><td></td><td></td></tr><tr><td>R</td><td></td><td></td><td></td><td></td></tr><tr><td>S</td><td></td><td></td><td></td><td></td></tr><tr><td>T</td><td></td><td></td><td></td><td></td></tr></table> <p>*lrec1 = 5 *numRecords = 10 *moreData = 1 *nextRec = 20</p>	K					L					M					N					O					P					Q					R					S					T					<table><tr><td>U</td><td></td><td></td><td></td><td></td></tr><tr><td>V</td><td></td><td></td><td></td><td></td></tr><tr><td>W</td><td></td><td></td><td></td><td></td></tr><tr><td>X</td><td></td><td></td><td></td><td></td></tr><tr><td>Y</td><td></td><td></td><td></td><td></td></tr><tr><td>Z</td><td></td><td></td><td></td><td></td></tr></table> <p>*lrec1 = 5 *numRecords = 6 *moreData = 0 *nextRec = X</p>	U					V					W					X					Y					Z				
A																																																																																																																																				
B																																																																																																																																				
C																																																																																																																																				
D																																																																																																																																				
E																																																																																																																																				
F																																																																																																																																				
G																																																																																																																																				
H																																																																																																																																				
I																																																																																																																																				
J																																																																																																																																				
K																																																																																																																																				
L																																																																																																																																				
M																																																																																																																																				
N																																																																																																																																				
O																																																																																																																																				
P																																																																																																																																				
Q																																																																																																																																				
R																																																																																																																																				
S																																																																																																																																				
T																																																																																																																																				
U																																																																																																																																				
V																																																																																																																																				
W																																																																																																																																				
X																																																																																																																																				
Y																																																																																																																																				
Z																																																																																																																																				

Figura 9. Exemplo de valores de retorno para chamadas subsequentes para `extractMember`. Observe que durante a terceira chamada, `*nextRec` tem um valor listado X. Isso significa que o valor de `*nextRec` não é significativo e não precisará ser alterado.

putMember

Atualiza o conteúdo de um membro ou cria um novo membro se o `memberID` especificado não existir na instância

```
int putMember(char instanceID[256],
              char memberID[256], char** contents, int lrec1,
              int* numRecords, char recFM[4], int moreData,
              int nextRec, int eof, void** params,
              void*** customReturn, char error[256])
```

char instanceID[256]	Entrada	A instância que contém o membro
char memberID[256]	Entrada	O ID do membro que está sendo atualizado/criado
char** contents	Entrada	Mantém o novo conteúdo do membro
int lrec1	Entrada	O número de colunas no conjunto de dados e na matriz
int* numRecords	Entrada/Saída	O número de registros no conjunto de dados ou o número de linhas na matriz
char recFM[4]	Entrada	Contém o formato de registro do conjunto de dados (FB, VB etc.)
int moreData	Entrada	Será 1 se o cliente tiver mais chunks de dados para enviar; caso contrário, 0

int nextRec	Entrada	O registro no conjunto de dados para o qual o registro 0 da matriz de conteúdo é mapeado
int eof	Entrada	Se 1, indica que a última linha da matriz deverá marcar a última linha no conjunto de dados; caso contrário, 0
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Como `extractMember`, `putMember` suporta os dados sendo enviados em chunks. `putMember` também deve suportar clientes que desejam passar chunks de dados que não estão em ordem sequencial. Por exemplo, um cliente podem enviar os registros de 10 a 19, de 20 a 29 e depois de 0 a 9. O RAM deve manipular tal situação e atualizar adequadamente o membro, ou retornar um código de erro e preencher o buffer de erros com uma sequência declarando que ele não pode manipular tal situação.

`numRecords` descreve quantos registros o cliente gostaria de atualizar/gravar na entrada, enquanto o RAM deve configurá-lo com o número de registros que foram de fato gravados para saída. Se houver uma diferença entre os dois, o cliente tentará inserir os membros que não foram gravados. Portanto, depois de receber uma resposta do RAM, o cliente irá configurar `nextRec` com o novo valor `numRecords` mais `nextRec` em sua próxima chamada `putMember`.

Para `putMember`, `nextRec` informa ao RAM onde começar a gravar o buffer de conteúdo que foi passado. Por exemplo, se `nextRec` for 0, o RAM deverá começar no início do membro.

`moreData` significa que o cliente estará chamando `putMember` novamente com outro chunk. Cabe ao desenvolvedor do RAM decidir como manipular uma situação em que `moreData` está configurado e a próxima chamada para o RAM não é uma chamada para a função `putMember` que fornece o próximo chunk de dados. Nesse caso, o RAM pode simplesmente retornar um erro. Como alternativa, ele pode tratar o problema e seguir em frente.

`eof` significa que o buffer de conteúdo atual tem os últimos registros de um membro. Se um membro de 40 registros precisasse ser reduzido a 5 registros, `eof` seria configurado como 1 quando o quinto registro estivesse sendo passado. Isso nunca deve ser configurado quando `moreData` é igual a 1.

Consulte a origem para o RAM Esqueleto e o RAM PDS de amostra para obter mais ajuda (consulte “Localizando Arquivos de Amostra” na página 3 para obter informações sobre como localizar esses arquivos de origem).

Operação:

1. Assegure-se de que os valores `lrec1`, `numRecords` e `nextRec` que foram passados sejam válidos.
2. Abra dataset e grave do registro `nextRec` para o registro `nextRec + numRecords`.
3. Se `eof` for especificado, assegure-se de que todos os registros que começam com o registro no índice `nextRec + numRecords` sejam removidos.
4. Se `moreData` for igual a 0, feche o conjunto de dados. Se `moreData` for igual a 1, deixe o conjunto de dados aberto se seu estado não puder ser mantido entre as chamadas, ou feche o conjunto de dados e verifique se ele pode ser reaberto no local apropriado com os valores sendo passados da próxima vez que `putMember` for chamado.

Extrair para Externo

O CARMA fornece aos RAMs a capacidade de extrair arquivos de um SCM em um ambiente normal do host de PDSs e arquivos sequenciais.

copyFromExternal

Copia um membro de um PDS ou um SDS.

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

char instanceID[256]	Entrada	A instância que contém o membro sendo copiado
char memberID[256]	Entrada	O ID do membro sendo copiado
char external[256]	Entrada	O local do qual copiar. Um membro PDS ou um membro SDS. Exemplos: FEK.#CUST.EXT.STOR FEK.#CUST.EXT.PDS(MEMBER)
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

copyToExternal

Copia um membro para um PDS ou um SDS.

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

char instanceID[256]	Entrada	A instância que contém o membro sendo copiado
char memberID[256]	Entrada	O ID do membro sendo copiado
char target[256]	Entrada	O local para o qual copiar. Um membro PDS ou um membro SDS. Exemplos: FEK.#CUST.EXT.STOR FEK.#CUST.EXT.PDS(MEMBER)

void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Transferência de Arquivo Binário

Para transferir com êxito arquivos que contêm dados binários sem incorrer em danos, o RAM usa um conjunto projetado de funções para extrair e colocar arquivos binários de um SCM. Depois que um membro binário foi extraído de um SCM, o RAM entrega o membro para o CARMA390, que continua a passá-lo até que o membro atinja a máquina do usuário. Em cada estágio do processo de transferência, o membro é reconhecido como contendo dados binários e nenhuma mudança é aplicada ao membro porque isso danificaria os dados.

extractBinMember

Recupera o conteúdo de um membro binário.

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

char instanceID[256]	Entrada	A instância que contém o membro sendo extraído.
char memberID[256]	Entrada	O ID do membro que está sendo extraído.
char** contents	Saída	Ponteiro para o conteúdo do membro
int* length	Saída	O comprimento do conteúdo do membro.
int* moreData	Saída	Se a extração precisar ser chamada novamente porque há mais dados, configure como 1 o valor da variável para o qual isso aponta, ou então designe como 0 o valor para o qual ela aponta.
int start	Entrada	O local de byte do arquivo do qual iniciar a extração.
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)

void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

putBinMember

Atualiza o conteúdo de um membro binário ou cria um novo membro se o memberID especificado não existir na instância.

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

char instanceID[256]	Entrada	A instância que contém o membro sendo atualizado/criado.
char memberID[256]	Entrada	O ID do membro que está sendo atualizado/criado.
char* contents	Entrada	Mantém o novo conteúdo dos membros.
int length	Entrada	Ponteiro para o comprimento dos dados a serem gravados.
int moreData	Entrada	Será 1 se o cliente tiver mais chunks de dados para enviar; caso contrário, 0.
int start	Entrada	O local de byte do arquivo para iniciar a colocação de dados.
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Funções de Metadados

getAllMemberInfo

Recupera todos os metadados de um membro ou de uma instância

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

char instanceID[256]	Entrada	O ID da instância que contém o membro
char memberID[256]	Entrada	O ID do membro para o qual metadados estão sendo retornados. O ID poderá ficar vazio (configurado como todos os espaços) se as informações do membro forem recuperadas para a instância e não para um membro específico.
KeyValPair** metadata	Saída	Isso deve ser alocado e preenchido com todos os pares de valores de chaves de metadados para o membro especificado
int* num	Saída	O número de pares de valores de chaves para os quais a matriz foi alocada
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Operação:

1. Consulte o SCM a respeito dos metadados do membro determinado.
2. Aloque a matriz contents. Caso esteja desenvolvendo um RAM em C, use o seguinte código:

```
*metadata = malloc(sizeof(KeyValPair) * *num);
```
3. Preencha a matriz contents com os pares de valores de chaves.

getMemberInfo

Recupera uma parte específica dos metadados de um membro ou de uma instância.

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

char instanceID[256]	Entrada	O ID da instância que contém o membro
char memberID[256]	Entrada	O ID do membro cujos metadados estão sendo recuperados. Se configurado como todos os espaços, os metadados da instância deverão ser retornados.
char key[64]	Entrada	A chave do valor a ser retornado
char value[256]	Saída	O valor solicitado
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

getMemberInfo retorna o valor da chave especificada para o membro determinado.

updateMemberInfo

Atualiza uma parte específica dos metadados de um membro ou de uma instância.

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

char instanceID[256]	Entrada	O ID da instância que contém o membro
char memberID[256]	Entrada	O ID do membro cujos metadados estão sendo configurados. Se configurado como todos os espaços, os metadados da instância deverão ser configurados.
char key[64]	Entrada	A chave do valor a ser configurado
char value[256]	Entrada	O valor a ser configurado
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)

void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

updateMemberInfo tenta atualizar os metadados de um membro (especificados pela chave determinada) com o valor determinado.

Outras Operações

lock

Bloqueia o membro

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

char instanceID[256]	Entrada	A instância que contém o membro
char memberID[256]	Entrada	O ID do membro que está sendo bloqueado
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

unlock

Desbloqueia o membro

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

char instanceID[256]	Entrada	A instância que contém o membro
char memberID[256]	Entrada	O ID do membro que está sendo desbloqueado

void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

check_in

Efetua o registro de entrada do membro. Isso consiste apenas em configurar um sinalizador para marcar que o registro de entrada é efetuado.

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

char instanceID[256]	Entrada	A instância que contém o membro
char memberID[256]	Entrada	O ID do membro cujo registro de entrada está sendo efetuado
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

check_out

Efetua o registro de saída do membro. Isso consiste apenas em configurar um sinalizador para marcar que o registro de saída é efetuado.

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

char instanceID[256]	Entrada	A instância que contém o membro
----------------------	---------	---------------------------------

char memberID[256]	Entrada	O ID do membro cujo registro de saída está sendo efetuado
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

performAction

Executa a ação identificada no actionID usando os parâmetros fornecidos e os valores de retorno em customReturn (quando aplicável).

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

int actionID	Entrada	A ação customizada que está sendo solicitada, conforme definido no CRADEF VSAM.
char instanceID[256]	Entrada	A instância na qual a ação está sendo executada. Se este e memberID estiverem configurados como todos os espaços, isso indicará que a ação deve ser executada no RAM.
char memberID[256]	Entrada	O membro no qual a ação está sendo executada. Se isso estiver configurado como todos os espaços, indicará que a ação deve ser executada na instância.
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)

char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.
-----------------	-------	--

getVersionList

Fornece uma lista de versões disponíveis para um determinado membro

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

char instanceID[256]	Entrada	A instância na qual o membro está
char memberID[256]	Entrada	O membro para o qual obter uma lista de versões
int* num	Saída	O número de versões
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

VersionIdent será identificado pela seguinte estrutura:

```
typedef struct {
char memberID[256]; /*Um memberID com versão, como
    baseMemberID_VerNum*/
char versionKey[64]; /* Uma maneira de se referir à versão, como
    "1.2.3"...deve ser igual ao valor
    para a chave de metadados carma.version*/
    char comments[256]; /* Os comentários fornecidos pelo RAM sobre a versão,
    podem ser registro de data/hora, mudanças, etc.. */
} VersionIdent;
```

A lista de versões deverá ser uma lista completa de versões ordenadas, mas o Desenvolvedor do RAM pode optar por usar um ID ‘com versão’ para a versão atual ou usar o ID inalterado. Como exemplo, a versão atual de um membro pode ser acessível por meio de “location(Member)” ou “location(Member)_1.4”, em que o arquivo está na versão 1.4. O desenvolvedor do RAM pode portanto optar por retornar “location(Member)_1.4” ou “location(Member)” como a versão mais recente na lista.

Ao retornar uma lista de membros por meio das funções de navegação, como getMembers, os memberIDs retornados NÃO DEVEM incluir a versão. Alterar o memberID para um membro impede que um cliente CARMA rastreie esse membro adequadamente.

Para suportar versão, os Desenvolvedores do RAM devem manipular as chamadas do CARMA quando apresentadas com um ID 'com versão' para o memberID.

Se um desenvolvedor do RAM desejar suportar versão para alguns, mas não para todos os membros, um código de retorno 130, que significa "O membro não suporta versão" pode ser usado.

Desenvolvimento do RAM Usando COBOL

Embora a linguagem de programação C seja uma opção suficiente para o desenvolvimento da maioria dos RAMs, em determinadas ocasiões, você poderá achar benéfico desenvolver um RAM em COBOL. Saiba que existem algumas vantagens em usar COBOL para desenvolvimento do RAM, além de haver algumas desvantagens também:

Vantagens do desenvolvimento do RAM em COBOL

- O código entre as funções é separado de maneira mais clara, forçando um design rigoroso e obrigando a um inventário cuidadoso dos recursos compartilhados entre as funções de programa do RAM.
- Como o COBOL está profundamente associado ao host, os recursos para o desenvolvimento em COBOL podem estar mais prontamente disponível no sistema.
- Como a manipulação de sequência em COBOL não depende de delimitadores NULL, a probabilidade de ocorrer exceções de proteção é menor do que durante o desenvolvimento em C.
- Os RAMs que envolvem a incorporação da implementação de lógica de negócios ou grandes quantidades de dados sendo ordenados aleatoriamente são mais simples de desenvolver em COBOL.
- O código COBOL tem a propriedade de ser documentado automaticamente.

Desvantagens do desenvolvimento do RAM em COBOL

- As estruturas dinâmicas usadas pelo CARMA são complicadas de se lidar em COBOL.
- O uso de recursos adicionais no estilo C envolve a inclusão de código C na origem de COBOL para C.
- O tipo de dados disponível em C não está disponível em COBOL. Você deve ter mais cuidado ao lidar com ponteiros.

O CARMA é fornecido com um RAM de amostra desenvolvido em COBOL, apropriadamente chamado de RAM COBOL de amostra. Para funcionar corretamente, esse RAM COBOL de amostra requer a origem de COBOL para C. Você pode usar esse RAM como ponto de início para seu próprio RAM escrito em COBOL, mas o RAM COBOL de amostra fornecido não deve ser usado em um ambiente de produção.

Nota: Para usar o RAM COBOL de amostra, você deve atualizar as informações do Custom Action Framework (CAF) nos clusters do VSAM. Os detalhes de como fazer isso podem ser encontrados no IBM Rational Developer for System z Host Configuration Guide (SC31-6930-02).

Estrutura de Programa do RAM COBOL

Codificando o ID do Programa

Os RAMs desenvolvidos em C implementam as funções de API do RAM CARMA, como `initRAM` ou `getMembers`. Os RAMs desenvolvidos em COBOL implementam cada uma dessas funções como programas COBOL individuais (chamados *programas de função do RAM*). No tempo de compilação, o código de origem de cada programa é concatenado e compilado em uma única DLL. Cada ID do programa será exportado para um deck paralelo de definição se a DLL estiver compilada para um PDS. O ID do programa de cada programa de função do RAM deve corresponder ao nome da função do RAM implementada por esse programa.

Nota: Essa correspondência deve fazer distinção entre maiúsculas e minúsculas. Por exemplo, o seguinte código definiria o programa que implementa a função `getInstances` do RAM:

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

A Seção de Ligação

Em um programa de função do RAM COBOL, a seção de ligação é usada para definir valores de parâmetro, estabelecer endereçabilidade para valores de ponteiro passados como parâmetros e referenciar o valor de número inteiro retornado pela função do RAM.

Cada parâmetro sendo passado à função do RAM deverá ser definido como um item de nível 77. Embora esses parâmetros não possam ser agrupados como itens de nível 77, recomenda-se que sejam definidos adjacentes entre si, na mesma sequência em que são passados ao programa (para maior clareza, localidade de referência e capacidade de leitura).

Nota: Para ajudar a facilitar o desenvolvimento, um exemplo de copybook com parâmetros predefinidos para uso em uma seção de ligação pode ser encontrado no membro da biblioteca de amostra CRACPY05.

Por exemplo, você poderia usar o seguinte código para definir os parâmetros do programa de função do RAM `getInstances`:

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

Nota: Os itens usados na divisão de procedimento anterior são exibidos conforme definidos no copybook.

Os itens de nível 77 também devem ser definidos para áreas referenciadas por ponteiros que não são dinâmicos no tamanho. Por exemplo, uma definição deve existir para referenciar o buffer de erros de 256 bytes. Use a seguinte definição para esse buffer de erros :

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

A seção de ligação também deve conter uma referência ao valor de número inteiro que está sendo retornado da função do RAM (o código de retorno). Defina esse número inteiro usando o seguinte código:

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

A endereçabilidade para o código de retorno não precisa ser estabelecida. Pode simplesmente ser usada como se fosse definida na seção de armazenamento de funcionamento.

Definindo a Divisão de Procedimento

Os parâmetros devem ser estabelecidos com uma frase USING para que possam estar disponíveis ao programa COBOL. Como os parâmetros podem ser passados por referência ou valor, determine qual método é mais apropriado aos seus parâmetros dependendo das práticas de codificação em uso.

O seguinte exemplo de divisão de procedimento para a declaração `getInstances` ilustra como você pode designar os parâmetros a serem passados.

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

Como cada função do RAM retorna um valor de número inteiro, a frase `RETURNING` é usada para especificar que um valor de número inteiro seja retornado do programa COBOL.

Nota: A ordem especificada na divisão de procedimento deve também corresponder à ordem definida no protótipo da API.

Finalizando o Programa

Como cada programa de função do RAM COBOL serve ao propósito de uma função do RAM C, cada programa de função do RAM deve ser finalizado com uma diretiva `END PROGRAM`. Ao compilar uma DLL do RAM COBOL, os programas de origem COBOL são fornecidos ao compilador de COBOL como uma série de instruções DD concatenadas. O não fornecimento das diretivas `END PROGRAM` fará com que os programas sejam tratados como aninhados, o que gerará mensagens de erro do compilador.

Passando Valores de C para COBOL

Os argumentos de função passados de um programa C para um programa de função do RAM COBOL devem ser manipulados de maneira apropriada ao método pelo qual eles estão sendo passados. Mais informações sobre esse tópico podem ser encontradas no guia: *Language Environment Writing Interlanguage Communication Applications*. Para obter informações específicas sobre como passar valores entre linguagens, consulte o Capítulo 4, *Comunicação entre C e COBOL*. Todos os exemplos nesta seção se referem ao comportamento no qual tipos de dados equivalentes devem ser definidos sem o uso de `#pragma` no programa C de chamada.

Há duas maneiras de usar parâmetros que foram passados de C. Os parâmetros podem ser incluídos na frase `USING BY VALUE` ou na frase `USING BY REFERENCE` do cabeçalho da divisão de procedimento.

Recebendo Tipos de Dados C Básicos Passados por Valor

Como regra geral, os argumentos de tipos de dados C básicos, como `int`, `double`, `float` ou `long`, que são passados na função C por valor devem ser recebidos com a frase `BY VALUE` na divisão de procedimento do programa COBOL. Para obter informações sobre cada tipo de dados C básico passado `BY VALUE` e como ele deve ser definido como item de seção de ligação, consulte o *z/OS V1R8.0-V1R9.0*

Language Environment Writing Interlanguage Communication Applications (SA22-7563-05), Capítulo 4 "Comunicação entre C e COBOL", Tabela 11. "Tipos de Dados Suportados Passados por Valor (Diretos) sem #pragma".

Os argumentos passados de C usando ponteiros (como sequências na forma de matrizes de caracteres) recebidos BY VALUE devem ser desreferenciados usando o operador SET. Como alternativa, os argumentos que usam ponteiros também podem ser recebidos com a frase da divisão de procedimento BY REFERENCE no programa COBOL de recebimento, desde que não exista possibilidade de o ponteiro passado ter um valor NULL. Mais informações sobre essa técnica podem ser encontradas em "Evitando o Desreferenciamento (Recebendo Tipos de Dados C BY REFERENCE)" na página 52, localizado mais adiante nesta seção

Exemplo: Recebendo um número inteiro BY VALUE.

Nesse exemplo, o programa COBOL está recebendo um parâmetro definido como o tipo int em C.

Primeiro, uma entrada de seção de ligação deve ser definida para o valor de número inteiro recebido.

```
77 IN-INTEGER PIC S9(9) BINARY.
```

Em seguida, devemos incluir as informações corretas na instrução PROCEDURE DIVISION para tornar o número inteiro recebido disponível ao programa.

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

No programa COBOL, IN-INTEGER pode ser usado como se fosse qualquer outro item no armazenamento.

Exemplo 2: Recebendo Matrizes de Caracteres BY VALUE.

A maioria das funções de API do RAM C recebe uma matriz de caracteres C de 256 bytes preenchida com espaços chamada memberID. Em C, essa matriz é passada por referência usando um ponteiro.

Ao receber uma matriz de caracteres BY VALUE, o programa COBOL recebe uma cópia do ponteiro que aponta para o local de armazenamento que mantém os caracteres. Esse ponteiro deve ser desreferenciado manualmente para que a sequência possa ser usada no programa COBOL.

Defina o item na seção de ligação como POINTER.

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

Você também deve definir um segundo item na seção de ligação para desreferenciar o ponteiro.

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

Assegure-se de que PROCEDURE DIVISION receba o memberID adequadamente.

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

Em seguida, antes de trabalhar com o memberID, use o operador SET para desreferenciar IN-MEMBERID.

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

5. Agora DEREFERENCED-MEMBERID pode ser usado já que foi definido na seção de armazenamento de funcionamento:

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

Evitando o Desreferenciamento (Recebendo Tipos de Dados C BY REFERENCE)

No recebimento de um parâmetro com a frase BY REFERENCE, o programa COBOL cuidará das operações de desreferenciamento desde que o item esteja definido corretamente na seção de ligação. Isso é útil para evitar operações de desreferenciamento, mas arriscado nos casos em que um ponteiro NULL possa ser passado no programa COBOL de recebimento.

Nota: Um programa COBOL que recebe um ponteiro NULL para um argumento recebido BY REFERENCE provavelmente executará ABEND com uma exceção de proteção 0C4.

Exemplo: Recebendo matrizes de caracteres BY REFERENCE.

Nesse exemplo, memberID será recebido pelo método BY REFERENCE do CARMA.

Primeiro, uma entrada de seção de ligação deve ser definida para corresponder à matriz de caracteres sendo passada.

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

A instrução PROCEDURE DIVISION deve refletir à do item que está sendo recebido BY REFERENCE.

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

IN-MEMBERID agora pode ser usado como se fosse qualquer outro item definido no armazenamento de funcionamento.

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

Sabendo Quando Receber BY REFERENCE

As situações a seguir descrevem quando é apropriado ao programa COBOL receber parâmetros BY REFERENCE:

- O item sendo recebido é passado de C como um ponteiro para um tipo de dados simples que não exige diversos níveis de desreferenciamento. (por exemplo, int *, char *, double *).
- O item sendo recebido está sendo passado em COBOL por meio de um ponteiro e seu valor no programa de chamada pode ser alterado pelo programa chamado.
- O item sendo recebido é um ponteiro com a garantia de não ser NULL.

Sabendo Quando Receber BY VALUE

As situações a seguir descrevem quando é apropriado ao programa COBOL receber parâmetros BY VALUE:

- O item que chega em COBOL está sendo passado por valor de C (o item não está sendo passado por meio de um ponteiro e seu valor no programa de chamada não deve ser modificado).
- O item que chega em COBOL é um tipo de ponteiro que exigirá diversos níveis de desreferenciamento.
- O item que chega em COBOL é um ponteiro que potencialmente pode ter um valor NULL e deve ser validado antes do uso para evitar uma exceção (especialmente 0C4).

- Qualquer ponteiro que chega em COBOL o qual exige desreferenciamento manual usando o operador SET.
- Qualquer ponteiro para funções C.
- Qualquer valor recebido que seja um ponteiro e precisará ter aritmética de ponteiro executada nele.

Passando Dados de COBOL para C

Ao chamar funções C de DLL de dentro de COBOL, o método pelo qual os parâmetros são passados do programa COBOL deve corresponder cuidadosamente aos tipos de dados de cada parâmetro no protótipo da função C de recebimento. Isso é necessário a fim de evitar problemas como finalização anormal. A regra geral é que se uma função C receber um argumento que não seja um ponteiro, ele deve ser passado de COBOL usando a frase BY VALUE. Se o argumento for um ponteiro, deverá ser passado usando a frase BY REFERENCE.

Passando Itens COBOL como Argumentos de Função C Básicos

Os tipos de dados C básicos localizados nos protótipos de função devem ser passados por valor do programa COBOL de chamada. No exemplo a seguir, é chamada uma função C que aceita dois argumentos que são tipos de dados C básicos do programa COBOL de chamada.

Protótipo de função C:

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

Itens do armazenamento de funcionamento conforme devem ser definidos no programa COBOL de chamada:

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

Um exemplo da instrução CALL no programa COBOL.

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

Passando Itens COBOL em Funções C por Referência

As funções C frequentemente recebem argumentos para modificação de referência. O exemplo mais predominante disso é uma modificação de sequência no estilo C na qual uma matriz de caracteres é recebida por meio da cópia de um ponteiro para a sequência original. Os itens podem ser passados de COBOL para C para modificação de referência usando a frase BY REFERENCE dentro da instrução CALL. O exemplo a seguir demonstra essa situação.

Exemplo:

Protótipo de função C da função de recebimento:

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

Definições para o item do armazenamento de funcionamento que está sendo passado como um argumento e o valor de retorno:

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

A instrução CALL no programa COBOL:

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

Exemplo 2: Uma função C que recebe um ponteiro para um número inteiro do programa COBOL de chamada:

Protótipo de função C:

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

Entrada do armazenamento de funcionamento no COBOL de chamada:

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

A instrução CALL no programa COBOL:

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

Lidando com Operações de Ponteiro

Operações de Ponteiro Simples

Para a maioria dos parâmetros passados aos programas de função do RAM COBOL, pouco código de desreferenciamento de ponteiro pode precisar ser implementado usando o operador SET. Por exemplo, a maior parte dos programas receberá um ponteiro para um buffer de 256 bytes para uma mensagem de erro detalhada. Para que você possa preencher esse buffer entretanto, ele deve ser desreferenciado usando o operador SET. Para itens menores, o desreferenciamento pode ser evitado com a frase USING BY REFERENCE.

Como exemplo, o código a seguir demonstra como estabelecer endereçabilidade ao buffer de erros. O ponteiro para o buffer de erros é passado por valor à divisão de procedimento para getInstances e definido na seção de ligação, conforme a seguir:

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

Mais tarde, na seção de ligação, o item de nível 77 é definido para desreferenciar e executar operações sobre o buffer de erros:

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

Em seguida, na divisão de procedimento, estabelece-se a endereçabilidade para o buffer de erros depois de verificar se GIP-ERROR não é NULL:

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

Agora, é possível tratar o buffer de erros como normalmente se faz com qualquer campo alfanumérico normal de 256 bytes. Nesse caso, o buffer de erros é uma sequência com terminação não NULL de 256 bytes.

Operações de Ponteiro Complexas

Para ponteiros com diversos níveis de via indireta, as operações de desreferenciamento podem ser complicadas. O código COBOL para executar tais operações de desreferenciamento exigiriam diversos itens de nível 77 com uma operação SET para cada nível de via indireta. Para questões complicadas, estruturas alocadas dinamicamente são difíceis de acessar sem saber qual o tamanho máximo absoluto da estrutura.

Em vez de tentar operações de ponteiro complexas em COBOL, é altamente recomendável que um código dessa natureza seja implementado de forma modular usando a origem de COBOL para C. Atualmente, as funções são implementadas para alocação de memória e inserção e recuperação de dados em buffer de conteúdo. Você pode achar útil incluí-los nesse código conforme necessário e usá-los para operações mais complexas.

Aritmética de Ponteiro

Como alternativa, as operações de ponteiro complexas podem ser executadas em COBOL, mas reduzem a capacidade de leitura e a sustentabilidade do código. Para lidar com estruturas dinâmicas, a aritmética de ponteiro é necessária. Para executar a aritmética de ponteiro, use redefinições. Para criar um ponteiro que possa ser manipulado por meio da aritmética de ponteiro, use um código semelhante ao seguinte na seção de armazenamento de funcionamento:

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

Após a definição do ponteiro, será possível manipulá-lo conforme necessário usando a versão redefinida. O código a seguir alteraria o ponteiro para apontar para a próxima estrutura em um chunk de memória alocado de forma contígua contendo diversas estruturas.

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

Alocação de Memória

Algumas funções do RAM, como `extractMember` e `getAllMemberInfo`, exigem que o RAM aloque memória. Essa memória é liberada mais tarde pelo CARMA, que usa a função C `free` para desalocar a memória. Por essa razão, um RAM implementado em COBOL deve usar a função C `malloc` ou o serviço do Language Environment CEEGTST para alocar memória. A origem de COBOL para C tem uma função C chamada `CMALLOC` para fornecer acesso a `malloc` de dentro do código COBOL. A função `CMALLOC` aceita como argumento um número inteiro que represente o número de bytes solicitado e retorne um ponteiro para a parte da memória que foi alocada. É responsabilidade do desenvolvedor do RAM assegurar-se de que o ponteiro não seja `NULL` antes de tentar usar a memória alocada.

A seguinte chamada de amostra para `CMALLOC` ilustra seu uso:

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

O serviço de chamada do Language Environment CEEGTST também está disponível para adquirir armazenamento dinamicamente. Para obter mais informações sobre esse serviço e outros serviços fornecidos pelo Language Environment, consulte o *z/OS V1R9.0 Language Environment Programming Reference* (SA22-7562-09).

Variáveis Compartilhadas entre Programas

As variáveis globais que precisam ser compartilhadas entre programas de função do RAM podem ser declaradas como externas. O exemplo a seguir ilustra como declarar variáveis usando a palavra-chave `EXTERNAL` na entrada de armazenamento de funcionamento do programa de função `initRAM`:

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


No código de amostra anterior, as variáveis globais têm seus conjuntos de valores dentro da chamada para `initRAM`. Posteriormente, quando `terminateRAM` for chamado, esses valores serão exibidos para mostrar que são persistentes e compartilhados.

Esse tipo de definição deve ser usado para valores que precisam ser compartilhados entre chamadas para diferentes programas de função do RAM. Se um item de armazenamento de funcionamento só será acessado por um programa de função do RAM, não o declare como item externo. Os itens de armazenamento de funcionamento que não precisam ser modificados por outros programas de função do RAM não devem ser tornados externos.

Manipulando Dados do Custom Action Framework

O Custom Action Framework (CAF) permite que você expanda sobre os programas de função existentes do RAM COBOL, implementando novas ações customizadas que são designadas para atender às necessidades do cliente CARMA.

Manipulando Ações Customizadas

Ações customizadas podem ser criadas usando o arquivo de origem COBOL de amostra (CRACOB16, localizado na biblioteca de amostra) como um exemplo para implementar a função do RAM `performAction`. No programa de função do RAM `performAction`, use uma instrução `EVALUATE` para executar código baseado em `ARG-ACTIONID` seletivamente:

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

Manipulando Parâmetros Customizados sem Usar as Funções de Utilitário COBOL para C

Os parâmetros customizados podem ser recuperados por meio de duas operações de desreferenciamento. Depois de assegurar-se de que o ponteiro passado ao programa do RAM não é `NULL`, estabeleça a endereçabilidade para a matriz de ponteiros. Em seguida, desreferencie cada ponteiro para acessar cada parâmetro customizado ao qual ele faz referência. O seguinte extrato da seção de ligação para o programa de função do RAM `performAction` descreve os campos conforme eles são definidos para lidar com dois parâmetros customizados:

```
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).
```

Estabeleça primeiro a endereçabilidade para a lista de ponteiros de parâmetros customizados usando o seguinte código:

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


Em seguida, estabeleça a endereçabilidade para parâmetros individuais.

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

Os parâmetros customizados agora podem ser usados como se fossem campos normais na seção de armazenamento de funcionamento. Além disso, supõe-se que a instrução de divisão de procedimento tenha especificado que PA-PARMS está sendo usado BY VALUE.

Nota: O código de exemplo anterior não inclui as verificações de ponteiros NULL que você deve incluir no código.

Manipulando Retornos Customizados sem Usar as Funções de Utilitário COBOL para C

O acesso a valores de retorno customizados em um RAM COBOL requer mais cuidado do que ao lidar com parâmetros customizados. Para que retornos customizados sejam estabelecidos, uma série de etapas concisas deve ser seguida. O código a seguir descreve os itens da seção de ligação que são usados para referenciar uma lista de dois retornos customizados. Supõe-se que a instrução de divisão de procedimento tenha especificado que PA-PARMS está sendo usado BY VALUE:

```
77 PA-RETURNS                POINTER.  
01 RETURNS-LV2               POINTER.  
01 RETURNS-LV3.  
    05 RETURN1                POINTER.  
    05 RETURN2                POINTER.  
  
01 CUSTOM-RETURN1            PIC X(8).  
01 CUSTOM-RETURN2            PIC S9(9) BINARY.
```

Comece desreferenciando o primeiro nível de via indireta:

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

Em seguida, aloque a memória necessária para a matriz de ponteiros para os parâmetros customizados:

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

Agora, configure o ponteiro de segundo nível para apontar para esse bloco de memória.

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

Em seguida, estabeleça a endereçabilidade para a lista de ponteiros para os valores de retorno que você acabou de alocar:

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

Aloque a memória necessária para os parâmetros customizados:

```
* 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.
```

Nota: Esse código configura automaticamente a lista de ponteiros em uma frase RETURNING. Sendo assim, não é necessário configurar esses ponteiros manualmente.

Por último, estabeleça a endereçabilidade para os valores de retorno e configure-os adequadamente.

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

Nota: Consulte a origem de amostra do RAM COBOL para documentação e exemplos de como usar as funções de utilitário COBOL para C.

Diferenças entre “DLL de Utilitário” e “Origem de Utilitário COBOL para C”

Em uma documentação do CARMA, há referências a uma “DLL de utilitário” e uma “origem de utilitário COBOL para C”. Existe a possibilidade de confusão entre esses dois itens.

O nome “DLL de utilitário” é uma denominação incorreta. A “DLL de utilitário” é um conjunto de código de origem C encontrada no membro CRASUTIL na biblioteca SFEKSAMP do CARMA. Nenhuma forma de DLL compilada dessa origem é fornecida. A origem é destinada a fornecer diversas funções de utilitário em C que podem ser compartilhadas por várias implementações do RAM. Essas funções implementam tarefas que os desenvolvedores do RAM podem com frequência precisar executar no código deles. Esse código é fornecido pela equipe de desenvolvimento do CARMA para facilitar o processo de desenvolvimento do RAM. É possível compilar a origem como uma DLL, ou como código de objeto e incluí-la no processo de vinculação de qualquer RAM. Em qualquer dos dois casos, o código destina-se a ser compilado com as opções do compilador para produzir código DLL (por exemplo: RENT,DLL). Na criação dos RAMs de amostra fornecidos com o CARMA, o código do objeto de utilitário foi vinculado ao módulo final.

A outra origem de utilitário fornecida, COBOL para C, também é código C encontrado no membro CRACOBC1 da biblioteca SFEKSAMP. Essa origem é fornecida como um conjunto de funções C acessíveis aos desenvolvedores do RAM COBOL para simplificar tarefas que são complicadas de implementar em COBOL. As funções fornecidas também tornam possível acessar o log do CARMA, que é difícil em COBOL devido à especificação da API do RAM CARMA para o formato da função initRAM.

A “DLL de utilitário” e o “utilitário COBOL para C” são fornecidos aos desenvolvedores como código de amostra não suportado com o objetivo de simplificar a tarefa de desenvolvimento do RAM. Os desenvolvedores de C provavelmente só considerarão o uso da “DLL de utilitário” para desenvolver um RAM, e os desenvolvedores de COBOL deverão considerar utilizar ambos para simplificar o processo de desenvolvimento.

Depurando e Evitando Finalização Anormal

Há diversas técnicas e práticas de codificação disponíveis para facilitar o desenvolvimento do RAM COBOL.

Exibindo Valores para Ajudar a Depurar o RAM COBOL

O verbo DISPLAY pode ser usado para inspecionar os valores de variáveis de programa, parâmetros sendo passados e buffers sendo preenchidos. Além disso, as instruções DISPLAY podem ser muito úteis se inseridas para rastrear o caminho de execução. O mais importante é observar que os valores exibidos para ponteiros são mostrados em decimal, não em hexadecimal. A saída do uso do verbo DISPLAY será exibida na tarefa em spool do CARMA.

Ponteiros NULL

A tentativa de desreferenciar um ponteiro NULL quase certamente resultará em uma exceção de proteção. Isso efetivamente resultará não só na finalização do RAM, mas também do CARMA. Para evitar essa finalização anormal, todos os valores de ponteiro deverão ser verificados em busca de valores NULL. É fornecida documentação adicional sobre ponteiros e verificação de valores NULL no *Enterprise COBOL for z/OS Language Reference*.

Saindo Corretamente dos Programas de Função do RAM

Por convenção, STOP RUN é usado para finalizar a execução de um programa escrito puramente em COBOL. Entretanto, codificar STOP RUN em um RAM COBOL finalizará o CARMA e o RAM COBOL. É recomendável evitar STOP RUN, a menos que as circunstâncias exijam esse tipo de comportamento. Use EXIT PROGRAM em vez de STOP RUN para sair da execução do RAM COBOL e retornar ao processamento do CARMA.

Capítulo 4. Customizando uma API do RAM Usando o CAF

O Custom Action Framework (CAF) é usado pelos desenvolvedores do RAM para descrever para os clientes CARMA como suas APIs do RAM diferem da API do RAM padrão. O CAF permite que uma API do RAM defina as seguintes diferenças entre sua API e a API do RAM padrão:

- Ações adicionais ("customizadas")
- Ações padrão desativadas
- Parâmetros adicionais ("customizados") para ações padrão
- Valores de retorno adicionais ("customizados") para ações padrão
- Campos que descrevem metadados que devem ser exibidos no cliente

Essas diferenças são definidas usando as informações do CAF. As informações do CAF podem ser consideradas um contrato entre um RAM e os clientes CARMA que usam esse RAM; é garantido que o RAM será executado corretamente desde que os clientes CARMA sigam as informações do CAF do RAM. Antes de tentar definir as informações do CAF de um RAM, convém criar um modelo conceitual das informações do CAF do RAM. Isso ajudará você a planejar como definirá as informações do CAF dos RAMs nos clusters do VSAM do CARMA. Este capítulo fornece um exemplo prático de como criar tal modelo para um RAM e como definir então as informações do CAF para o RAM usando esse modelo.

Para poder seguir o exemplo, você deve entender primeiro os tipos de objeto básicos do CAF. O modelo de exemplo do RAM foi projetado usando esses objetos.

Tipos de Objeto CAF

Há cinco tipos de objetos usados nas informações de CAF: RAMs, parâmetros, valores de retorno, ações e campos.

RAM

Os RAMs fornecem ao CARMA acesso a SCMs específicos. As informações do CAF para o RAM incluem as seguintes:

Nome O nome do RAM

Descrição

Uma descrição curta do RAM

ID do RAM

Um identificador numérico para o RAM entre 0 e 99

Linguagem de Programação

A linguagem de programação na qual o RAM foi escrito (C, COBOL ou PL/I)

Nome da DLL do RAM

O nome da DLL do RAM

Versão

O número da versão do RAM

Versão do repositório

A versão do repositório com a qual o RAM foi designado para trabalhar

Versão do CARMA

A versão do CARMA com a qual o RAM foi designado para trabalhar

Parâmetro

Parâmetros são valores passados a uma ação do cliente CARMA. Eles são definidos por RAM; assim, uma vez definido um parâmetro, seu ID pode ser usado na lista de parâmetros de qualquer ação definida para esse RAM. Isso pode ser útil se muitas das ações de um RAM precisarem dos mesmos parâmetros.

As informações do CAF para o RAM incluirão as seguintes informações sobre cada parâmetro.

Nome O nome do parâmetro

Descrição

Uma descrição curta do parâmetro

ID do parâmetro

Um identificador numérico para o parâmetro entre 000 e 999 (3 bytes).

ID do RAM

O ID do RAM ao qual o parâmetro pertence

Tipo O tipo de dados do parâmetro. Escolha na seguinte lista de tipos de dados de programação padrão: int, long, double e string.

Comprimento

Um valor numérico especificado de forma diferente com base no tipo de parâmetro:

Tipo de Parâmetro	Instruções de Especificação
int	Arbitrário (esse valor não importa)
long	Arbitrário (esse valor não importa)
double	A precisão do parâmetro
string	A largura do campo do parâmetro

Constante

Se o parâmetro sempre conterá ou não o mesmo valor

Valor Padrão

O valor padrão do parâmetro. Essa não é uma informação opcional.

Prompt

O prompt que deverá ser exibido pelos clientes CARMA ao solicitar um valor para o parâmetro dos usuários

Valor de Retorno

Valores de retorno são o resultado de uma ação chamada pelo CARMA. Eles são definidos por RAM; assim, uma vez definido um valor de retorno, seu ID pode ser usado na lista de valores de retorno de qualquer ação definida para esse RAM. Isso pode ser útil se muitas das ações de um RAM precisarem dos mesmos valores de retorno.

As informações do CAF para o RAM incluirão as seguintes informações sobre cada valor de retorno.

Nome O nome do valor de retorno.

Descrição

Uma descrição curta do valor de retorno.

ID do Valor de Retorno

Um identificador numérico para o valor de retorno entre 000 e 999 (3 bytes).

ID do RAM

O ID do RAM ao qual o valor de retorno pertence (2 bytes)

Tipo O tipo de dados do valor de retorno. Escolha na seguinte lista de tipos de dados de programação padrão: int, long, double e string.

Comprimento

Um valor numérico especificado de forma diferente com base no tipo de valor de retorno:

Tipo de Parâmetro	Instruções de Especificação
int	Arbitrário (esse valor não importa)
long	Arbitrário (esse valor não importa)
double	A precisão do valor de retorno
string	A largura do campo do valor de retorno

Constante

Se o parâmetro sempre conterá ou não o mesmo valor

Valor Padrão

O valor padrão do parâmetro

Prompt

O prompt que deverá ser exibido pelos clientes CARMA ao solicitar dos usuários um valor para o parâmetro

Ações

Todos os RAMs têm um conjunto padrão de ações definidas na API do RAM. Você pode usar o CAF para modificar essas ações padrão para usar parâmetros de entrada adicionais, para usar valores de retorno adicionais ou para serem ocultadas do CARMA (desativando essencialmente as ações).

Nota: Embora não seja possível especificar para o CAF que um parâmetro padrão em uma ação padrão seja removido, tal parâmetro pode simplesmente ser ignorado na implementação dessa ação se passado à ação por um cliente CARMA.

Você também pode declarar novas ações ("customizadas"). Cada ação customizada declarada tem um ID designado (chamado seu ID de ação). Quando um cliente CARMA tentar chamar uma ação customizada em um RAM, o CARMA chamará primeiro a função `performAction` do RAM, passando o ID de ação (fornecido pelo cliente CARMA) da ação customizada como um parâmetro. A função `performAction` deve então tentar chamar a função para a ação customizada com o ID de ação especificado.

Nota: É responsabilidade do desenvolvedor do RAM tratar o caso em que um ID de ação inválido é fornecido à função `performAction` do RAM. Uma maneira razoável de tratar esse caso seria retornar um erro ao cliente com uma mensagem de erro detalhada.

As informações do CAF para o RAM incluirão as seguintes informações sobre cada ação (para ações desativadas, apenas os IDs do RAM e de ação são necessários):

Nome O nome da ação

Descrição

Uma descrição curta da ação

ID da Ação

Um identificador numérico para a ação entre 0 e 999. Os IDs de ação entre 0 e 79 substituem ações padrão (consulte o Apêndice B, “IDs de Ação”, na página 117 para obter uma lista completa dos IDs para as ações padrão). Os IDs de ação entre 80 e 99 são reservados para uso do CARMA. Use um ID entre 100 e 999 para definir uma ação customizada.

ID do RAM

O ID do RAM ao qual a ação pertence

Lista de parâmetros

Uma lista dos IDs dos parâmetros que a ação usa. Se você estiver substituindo uma ação padrão, só precisará de uma lista daqueles parâmetros que estão sendo incluídos na lista de parâmetros padrão. Se você estiver definindo uma ação customizada, deverá listar os IDs de todos os parâmetros requeridos pela ação, exceto os IDs de instância e de membro, que são passados por padrão a cada ação customizada.

Lista de valores de retorno

Uma lista dos IDs para os valores de retorno que a ação retorna. Se você estiver substituindo uma ação padrão, só precisará de uma lista daqueles valores de retorno que estão sendo incluídos na lista de valores de retorno padrão. Se você estiver definindo uma ação customizada, deverá listar os IDs de todos os valores de retorno que estão sendo retornados pela ação, exceto o código de retorno da ação, que deve ser sempre retornado por cada ação customizada.

Campo

Campos descrevem metadados de interesse específicos dos usuários. As informações do CAF para Campos incluem as seguintes:

Nome O nome exibível localizado do campo.

Chave de Metadados

A chave de metadados a ser fornecida à função `getMemberInfo` para o campo a ser exibido.

Valor Padrão

O valor exibível localizado do campo se nenhum valor for retornado por uma chamada para `getMemberInfo`.

Descrição

Uma descrição exibível localizada dos metadados.

Desenvolvendo o Modelo de RAM para um RAM Customizado

Suponha que desejamos criar um RAM denominado RAM SAMP, capaz de acessar uma solução SCM denominada SCM de Amostra. Suponha que o SCM de Amostra opera de uma maneira que faz com que o RAM SAMP tenha as seguintes diferenças de um RAM CARMA padrão:

- Não fornece suporte para efetuar o registro de saída de arquivos

- Sua ação de bloqueio retorna o tipo de bloqueio, além dos valores de retorno para a ação de bloqueio do CARMA padrão
- Ele tem uma ação "bloquear instância", que bloqueia uma instância no SCM. Essa ação requer os seguintes parâmetros:

1. ID da Instância
2. Motivo

e retorna os seguintes valores:

1. Tipo de bloqueio
2. Código de retorno

- Tem uma ação "remover sinalizador", que remove um sinalizador de um membro no SCM. Essa ação requer os seguintes parâmetros:

1. ID da Instância
2. ID do membro
3. Motivo

e retorna os seguintes valores:

1. Código de retorno

- Tem uma ação "concatenar", que concatena o conteúdo de dois membros no SCM. Essa ação requer os seguintes parâmetros:

1. ID da instância-alvo
2. ID do membro-alvo
3. ID da instância de destino
4. ID do membro de destino

e retorna os seguintes valores:

1. Novo ID da instância
2. Novo ID do membro
3. Código de retorno

Para suportar totalmente a funcionalidade do SCM de Amostra, usaremos o CAF para customizar a API do RAM. Precisaremos criar três novas ações customizadas (para as operações bloquear instância, remover sinalizador e concatenar) e substituir duas das ações padrão (bloquear e efetuar registro de saída).

Suponha que para esse exemplo nós estejamos desenvolvendo a primeira versão do RAM SAMP (versão 1.0), que está sendo projetado para acessar o SCM de Amostra versão 1.4 e trabalhar com o CARMA versão 2.5, e que será escrito em C e compilado em uma DLL chamada SAMPRAM. Para esse exemplo, designaremos ao RAM SAMP um ID do RAM igual a 1.

Nota: Partiremos do princípio de que a DLL do RAM, SAMPRAM, está armazenada no PDS comum que contém todos os RAMs disponíveis no host do CARMA. Consulte "Construção do RAM" na página 13 para saber onde uma DLL do RAM deve ser armazenada.

Nós agora temos todas as informações sobre o RAM necessárias para o modelo de RAM SAM (consulte "RAM" na página 61). A tabela a seguir resume essas informações:

Tabela 10. Informações sobre o RAM SAMP

Nome	RAM SAMP
------	----------

Tabela 10. Informações sobre o RAM SAMP (continuação)

Descrição	Fornece ao CARMA acesso às instâncias do SCM de Amostra
ID do RAM	1
Linguagem de Programação	C
Nome da DLL do RAM	SAMPRAM
Versão	1.0
Versão do Repositório	1.4
Versão do CARMA	2.5

Neste momento, podemos achar útil tabular as informações (conforme descrito em “Ações” na página 63) para todas as ações que precisam ser criadas ou substituídas. As tabelas a seguir resumem essas informações. Observe que o ID da ação de bloqueio corresponde ao ID da ação de bloqueio padrão (consulte Apêndice B, “IDs de Ação”, na página 117) para assegurar-se de que a ação de bloqueio original seja substituída. A ação de registro de saída ativado da mesma forma tem um ID designado que corresponde à ação de registro de saída padrão.

Tabela 11. Informações sobre a ação bloquear instância do RAM SAMP

Nome	Bloquear instância
Descrição	Bloqueia uma instância no SCM
ID da Ação	100
ID do RAM	1
Lista de Parâmetros	ID da Instância Motivo
Lista de Valores de Retorno	Código de retorno Tipo de bloqueio

Tabela 12. Informações sobre a ação remover sinalizador do RAM SAMP

Nome	Remover sinalizador
Descrição	Remove um sinalizador de um membro no SCM
ID da Ação	101
ID do RAM	1
Lista de Parâmetros	ID da Instância ID do Membro Motivo
Lista de Valores de Retorno	Código de retorno

Tabela 13. Informações sobre a ação concatenar do RAM SAMP

Nome	Concatenar
Descrição	Concatena o conteúdo de dois membros no SCM
ID da Ação	102
ID do RAM	1

Tabela 13. Informações sobre a ação concatenar do RAM SAMP (continuação)

Lista de Parâmetros	ID da instância de destino ID do membro de destino ID da instância-alvo ID do membro-alvo
Lista de Valores de Retorno	Código de retorno Novo ID da instância Novo ID do membro

Tabela 14. Informações sobre a ação bloquear do RAM SAMP. Observe que não fornecemos uma descrição para esta ação, pois a descrição da ação padrão já está disponível ao cliente. Você pode substituir a descrição existente especificando uma nova nos clusters do VSAM, mas o cliente pode ou não usar a descrição atualizada.

Nome	Lock
Descrição	
ID da Ação	10
ID do RAM	1
Lista de Parâmetros	ID da Instância ID do membro
Lista de Valores de Retorno	Código de retorno Tipo de bloqueio

Tabela 15. Informações sobre a ação de registro de saída do RAM SAMP. Como essa ação está desativada, não precisamos incluir uma descrição, lista de parâmetros ou lista de valores de retorno.

Nome	Efetuar o registro de saída
Descrição	(Desativado)
ID da Ação	13
ID do RAM	1
Lista de Parâmetros	(Desativado)
Lista de Valores de Retorno	

Como os IDs da instância e do membro são passados por padrão a todas as ações (consulte a descrição de “Lista de parâmetros” em “Ações” na página 63), somente três parâmetros adicionais precisam ser definidos para as ações customizadas (bloquear instância, remover sinalizador e concatenar) e a ação de bloqueio: motivo, ID da instância-alvo e ID do membro-alvo. Para a ação concatenar, podemos mapear o ID da instância de destino e o ID do membro de destino respectivamente para o ID da instância e o ID do membro de parâmetros padrão.

Podemos agora listar todos os parâmetros necessários ao modelo de RAM SAMP. As tabelas a seguir resumem essas informações. Observe que são designados aos parâmetros ID sequenciais, começando com 0 para o primeiro parâmetro.

Nome	Motivo
Descrição	Motivo pelo qual a ação deve ser executada
ID do parâmetro	0
ID do RAM	1

Tipo	String
Comprimento	30
Constante	Não
Valor-padrão	Nenhum
Prompt	Por que você está solicitando que a ação seja executada?

Nome	ID da instância-alvo
Descrição	ID da instância que contém o membro cujo conteúdo deve ser anexado ao final do membro determinado
ID do parâmetro	1
ID do RAM	1
Tipo	String
Comprimento	15
Constante	Não
Valor-padrão	Nenhum
Prompt	Qual instância contém o membro que você deseja concatenar com o membro selecionado?

Nome	ID do membro-alvo
Descrição	ID do membro cujo conteúdo deve ser anexado ao final do membro determinado
ID do parâmetro	1
ID do RAM	1
Tipo	String
Comprimento	30
Constante	Não
Valor-padrão	Nenhum
Prompt	O conteúdo de qual membro você deseja anexar ao final do membro selecionado?

Somente três valores de retorno adicionais precisam ser definidos para o RAM SAMP, porque o código de retorno já é retornado por padrão (consulte a descrição de “Lista de valores de retorno” em “Ações” na página 63). As tabelas a seguir resumem as informações de valor de retorno necessárias para o modelo de RAM SAM. Novamente, observe que são designados aos valores de retorno ID sequenciais, começando com 0 para o primeiro valor de retorno.

Nome	Tipo de bloqueio
Descrição	O tipo de bloqueio sendo aplicado ao membro
ID do Valor de Retorno	0
ID do RAM	1
Tipo	Int

Comprimento	4
--------------------	---

Nome	Novo ID da instância
Descrição	A instância na qual os resultados da ação foram colocados
ID do Valor de Retorno	1
ID do RAM	1
Tipo	String
Comprimento	30

Nome	Novo ID do membro
Descrição	O membro que contém os resultados da ação
ID do Valor de Retorno	1
ID do RAM	1
Tipo	String
Comprimento	30

Com todas as informações necessárias para definir o RAM SAMP para o CAF perfeitamente tabulado, podemos representar as informações visualmente. A Figura 10 na página 70 ilustra o relacionamento entre as ações, os parâmetros e os valores de retorno usados no RAM SAMP. Antes de configurar os clusters para um RAM, você pode achar útil desenvolver um diagrama semelhante.

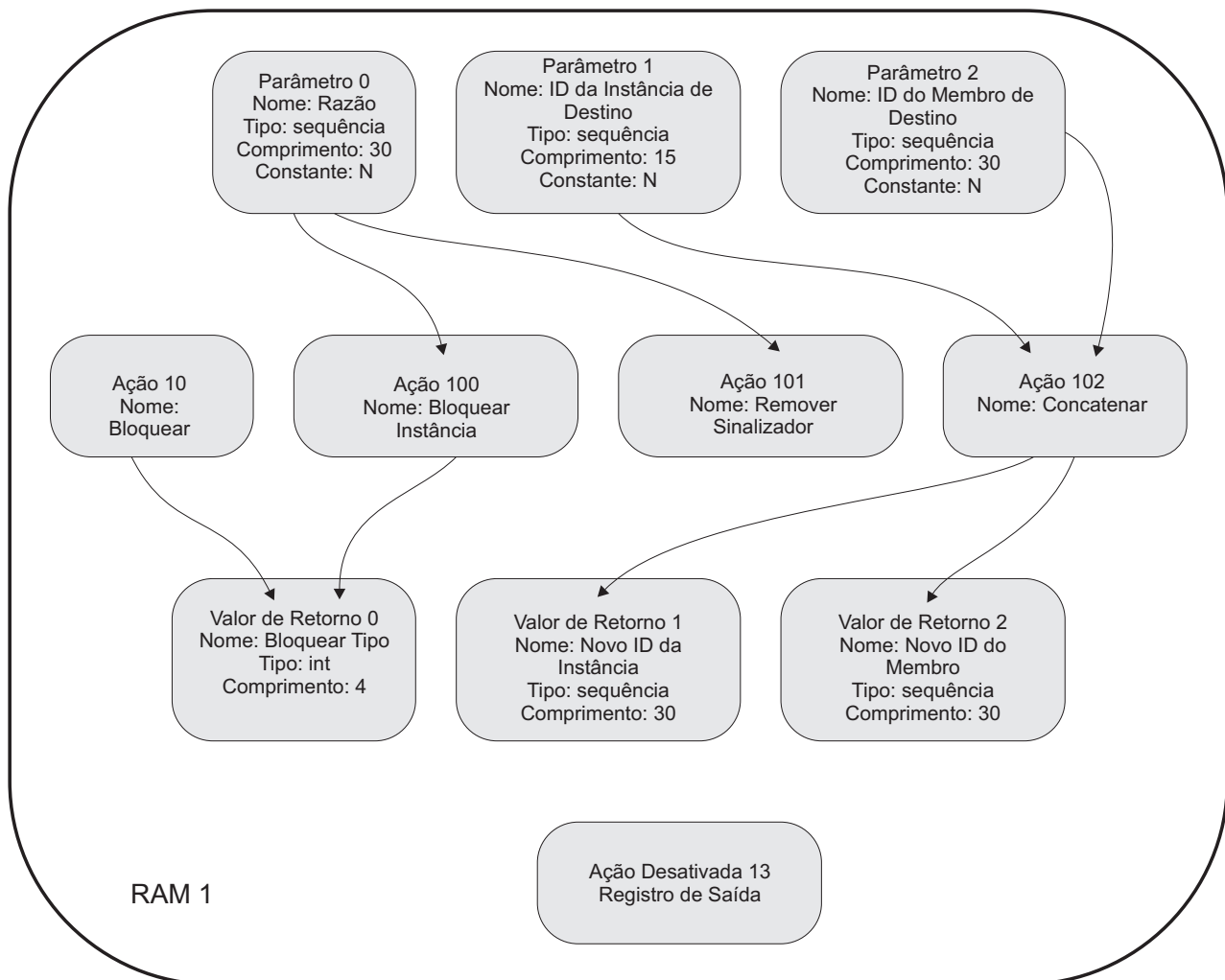


Figura 10. Representação visual do modelo do RAM SAMP. Somente informações pertinentes ao relacionamento entre os objetos são mostradas.

Criando Registros VSAM de um Modelo do RAM

Agora que temos um modelo para o RAM SAMP, podemos facilmente definir as informações do CAF do RAM SAMP. Para fazer isso, é necessário primeiro entender onde e como as informações do CAF são armazenadas. Há dois clusters do VSAM sequenciados por chave do CAF que armazenam todas as informações do CAF: CRADEF e CRASTRS. Conforme o CARMA é carregado, ele descobre os RAMs disponíveis a ele (bem como suas ações, parâmetros e valores de retorno correspondentes) lendo o CRADEF, que contém informações sobre os recursos dos RAMs disponíveis. Conforme necessário, o CARMA tenta determinar se o idioma preferencial de um usuário está disponível para um determinado RAM verificando o CRASTRS, que contém informações específicas de código de idioma para os RAMs.

CRADEF

O CRADEF armazena todos os dados do CAF independentes de idioma (dados que não precisam ser traduzidos de um código de idioma para outro), usando caracteres em inglês da página de códigos 00037. Ele contém registros para cada

tipo de objeto do CAF (RAMs, ações, parâmetros e valores de retorno), usando uma largura de registro de 1032 bytes. Entretanto, somente registros de ação podem de fato usar todos os 1032 bytes; os outros tipos de registro apenas preenchem os bytes não usados com espaços. O CRADEF usa uma chave de 8 bytes e reserva os 1024 bytes restantes para dados. A Tabela A resume a composição de um registro genérico no CRADEF:

Tabela 16. Formato de registro CRADEF

Registro de 1032 Bytes	
(8 bytes) Chave	(1024 bytes) Dados

Chaves de Registro

As chaves de registro CRADEF são compostas dos seguintes campos:

1. (1 byte) O caractere de tipo ("A" para ação, "D" para ação desativada, "P" para parâmetro, "R" para RAM, "T" para valor de retorno, "I" para Identificador e "F" para campo)
2. (2 bytes) O ID do RAM de dois dígitos preenchido com 0s à esquerda (um número de identificação exclusivo entre "00" e "99")
3. (3 bytes) O ID secundário de três dígitos preenchido com 0s à esquerda. Para todos os RAMs, o valor deverá ser "000". Para ações padrão, use o ID de ação predefinida, e para ações customizadas, use um ID de ação customizada que seja superior ou igual a 100. Para parâmetros, valores de retorno e campos, use IDs sequenciais, começando por 000.
4. (2 bytes) Não usado (reservado para uso futuro). Preencha esses bytes com espaços.

A tabela a seguir resume o formato de chave CRADEF.

Tabela 17. Formato de chave CRADEF. O número de bytes reservados para cada campo é especificado entre parênteses. Os campos marcados como "Não usado" devem ser preenchidos inteiramente com espaços.

Chave de 8 Bytes			
(1 byte) Tipo	(2 bytes) ID do RAM	(3 bytes) ID Secundário	(2 bytes) Não usado

Dados do Registro

O restante dos bytes em cada registro é usado para dados do registro. Esses 1024 bytes contêm diferentes campos dependendo do tipo de registro:

RAM

1. (8 bytes) O número da versão do RAM. Esse valor pode ser exibido aos usuários pelos clientes CARMA.
2. (8 bytes) A linguagem de programação na qual o RAM foi escrito. Selecione na seguinte lista de valores válidos: "C", "COBOL", "PLI" (como alternativa, "PL1" pode ser usado).
3. (8 bytes) O número da versão do repositório com o qual o RAM é compatível. Esse valor pode ser exibido aos usuários pelos clientes CARMA.
4. (8 bytes) o número da versão do CARMA com a qual o RAM é compatível. Esse valor pode ser exibido aos usuários pelos clientes CARMA.

5. (8 bytes) O nome da DLL do RAM

Ação

Nota: A largura combinada dos campos (1) e (3) a seguir deve ser menor ou igual a 1023.

1. (0 a 1023 bytes) Uma lista dos IDs de parâmetro usados pela ação. Os IDs listados devem ser separados por vírgulas. Não use uma vírgula final no fim da lista.
2. (1 byte) O caractere de barra vertical, "|". Esse símbolo é usado para indicar a separação entre as listas de IDs de parâmetro e de valor de retorno.

Nota: Esse caractere deve ser incluído mesmo que a lista de IDs de parâmetro ou de valor de retorno esteja vazia. Entretanto, *não* deverá ser incluído se *ambas* as listas estiverem vazias.

3. (0 a 1023 bytes) Uma lista dos IDs de valor de retorno usada pela ação. Os IDs listados devem ser separados por vírgulas. Não use uma vírgula final no fim da lista.

Ação desativada

(1024 bytes) Espaços vazios. Nenhum dado é necessário para ações desativadas.

Parâmetro

1. (16 bytes) O tipo de dados do parâmetro. Escolha entre os seguintes valores disponíveis: "INT", "LONG", "DOUBLE", "STRING".
2. (16 bytes) O comprimento do parâmetro. Essa é uma largura de precisão (para parâmetros do tipo "DOUBLE") ou de campo (para parâmetros do tipo "STRING"). Especifique esse valor numericamente (por exemplo, como "12" em vez de "doze"). Use um valor arbitrário se o tipo de parâmetro não for "DOUBLE" nem "STRING".
3. (1 byte) Um "Y" ou "N" para indicar se esse parâmetro tem ou não tem (respectivamente) um valor constante.

Valor de Retorno

1. (16 bytes) O tipo de dados do valor de retorno. Escolha entre os seguintes valores disponíveis: "INT", "LONG", "DOUBLE", "STRING".
2. (16 bytes) O comprimento do valor de retorno. Essa é uma largura de precisão (para valores de retorno do tipo "DOUBLE") ou de campo (para valores de retorno do tipo "STRING"). Especifique esse valor numericamente (por exemplo, como "12" em vez de "doze"). Use um valor arbitrário se o tipo de valor de retorno não for "DOUBLE" nem "STRING".

Campo

(64 bytes) A Chave de Metadados usada para identificar os metadados a serem exibidos. Opcionalmente, pode ser deixado todos os espaços; assim, o valor Nome (extraído do CRASTRS VSAM) será usado para a Chave de Metadados.

Identificador

(1024 bytes) Um identificador do RAM exclusivo usado para identificar o RAM. Se um ID exclusivo for fornecido, o RAM será selecionado por seu ID exclusivo e o campo ramId será ignorado.

A tabela a seguir resume os formatos de dados CRADEF para cada tipo de objeto do CAF.

Tabela 18. Formatos de dados CRADEF para cada tipo de objeto do CAF (a coluna "Tipo" lista os caracteres de tipo abreviados, em vez dos nomes de tipo completos). O número de bytes reservados para cada campo é especificado entre parênteses (um "" indica um campo de comprimento variável). Os campos marcados como "Não usado" devem ser preenchidos inteiramente com espaços.*

Tipo	Dados de 1024 Bytes				
R	(8 bytes) Versão do RAM	(8 bytes) Linguagem de Programação	(8 bytes) Versão do Repositório	(8 bytes) Versão do CARMA	(8 bytes) Nome da DLL
A	(* bytes) Lista de IDs de Parâmetro Vertical		(1 byte) Separadora de Lista	(* bytes) Lista de IDs de Valor de Retorno	
D	(1024 bytes) Não usado				
P	(16 bytes) Tipo		(16 bytes) Comprimento		(1 byte) Constante
T	(16 bytes) Tipo			(16 bytes) Comprimento	
F	(64 bytes) Chave de Metadados				
I	(1024 bytes) Identificador				

CRASTRS

O CRASTRS armazena todos os dados do CAF dependentes de idioma (dados que precisam ser traduzidos de um código de idioma para outro, como descrições e mensagens). Os idiomas são indexados no cluster do VSAM com base em um código de idioma de oito caracteres (por exemplo, "EN_US " ou "FR_FR ") e uma página de código de cinco caracteres (por exemplo, "00037"). Como um cliente CARMA inicializa o CARMA, o cliente fornece ao CARMA um código de idioma e a página de códigos, que o CARMA tenta localizar no CRASTRS. Se a combinação especificada de código de idioma e página de código não estiver disponível no ambiente do CARMA, o CARMA usará o código de idioma padrão ("EN_US") e a página de códigos ("00037") e retornará um erro ao cliente.

Quando um cliente solicitar a lista de RAMs disponíveis, o CARMA fará referência ao CRASTRS para tentar compor uma lista dos RAMs que estão disponíveis no código do idioma e na página de códigos do cliente solicitados. Por convenção, se um registro do RAM estiver disponível em um determinado código de idioma, espera-se que suas ações, parâmetros e valores de retorno também estejam disponíveis nesse mesmo código de idioma.

O CRASTRS permite sequências de um comprimento não fixo (anteriormente, o CRASTRS usava uma largura de registro de 2101 bytes compostos por uma chave de 21 bytes e 2080 bytes para dados). As sequências são separadas por um único caractere não de espaço usado para delimitar sequências em todo o arquivo. Você pode definir o caractere delimitador quando configura o arquivo CRASTRS. Se você não defini-lo, o padrão será o caractere de tabulação horizontal (0x05 em EBCDIC)

Se o CRASTRS contiver um registro com a chave "00000000000000000000" (vinte e um zeros), o CARMA usará o formato de sequência não fixo no arquivo inteiro do

VSAM. Se o CRASTRS não contiver essa chave de registro, o CARMA usará o formato de sequência de largura fixa no arquivo inteiro do VSAM.

Nota: As sequências mais longas estarão limitadas ao comprimento anterior máximo, se chamadas de função mais antigas forem usadas.

Nota: Ações desativadas não precisam de registros no CRASTRS porque não possuem sequência para ser traduzida.

Chaves de Registro

As chaves de registro CRASTRS são compostas pelos seguintes campos:

1. (8 bytes) O código de idioma do registro (por exemplo, "EN_US ")
2. (5 bytes) A página de códigos do registro (por exemplo, "00037")
3. (8 bytes) A chave para o registro CRADEF ao qual esse registro CRASTRS corresponde

A tabela a seguir resume o formato de chave CRASTRS.

Tabela 19. Formato de chave CRASTRS. O número de bytes reservados para cada campo é especificado entre parênteses.

Chave de 21 Bytes		
(8 bytes) Código do Idioma	(5 bytes) Página de Códigos	(8 bytes) Chave de Registro

Dados do Registro

O restante do registro é usado para dados do registro. Os dados do registro contêm diferentes campos dependendo do tipo de registro. Nas versões anteriores, esses dados estavam limitados a 2080 bytes. Quando CRASTRS contém um registro com a chave "00000000000000000000" (vinte e um zeros), esse limite não se aplica mais.

RAM, ação e tipo de retorno

1. O nome do objeto CAF ao qual esse registro corresponde (Anteriormente, limitado a 16 bytes)
2. Uma descrição do objeto CAF ao qual esse registro corresponde (Anteriormente, limitado a 1024 bytes)

Parâmetro

1. O nome do parâmetro ao qual esse registro corresponde (Anteriormente, limitado a 16 bytes)
2. O valor padrão do parâmetro ao qual esse registro corresponde (Anteriormente, limitado a 16 bytes)
3. O prompt que o cliente deverá exibir ao solicitar um valor para o parâmetro ao qual esse registro corresponde (Anteriormente, limitado a 1024 bytes)
4. Uma descrição do parâmetro ao qual esse registro corresponde (Anteriormente, limitado a 1024 bytes)

Campo

1. O nome correspondente a esses metadados. Para usar esse nome localizado como Chave de Metadados, deixe a Chave de Metadados em branco no CRADEF. Deixar isso em branco fará com que o nome seja a Chave de Metadados definida no CRADEF. (Anteriormente, limitado a 128 bytes)

2. O valor padrão a ser exibido se nenhum valor for fornecido pelo RAM para um determinado membro da Chave de Metadados. (Anteriormente, limitado a 256 bytes)
3. Uma descrição dos metadados mostrados nesse campo. (Anteriormente, limitado a 1024 bytes)

Todas as informações na seção de dados devem estar no código de idioma e na página de códigos especificados na chave. A tabela a seguir resume os formatos de dados CRASTRS para cada tipo de objeto do CAF.

Tabela 20. Formatos de dados CRASTRS para cada tipo de objeto do CAF (a coluna "Tipo" lista os caracteres de tipo abreviados, em vez dos nomes de tipo completos). Observe que o tipo de ação desativada não foi incluído nessa tabela porque o CRASTRS não deverá ter registros para ações desativadas.

Tipo	Data			
A R T	Nome		Descrição	
P	Nome	Valor-padrão	Prompt	Descrição
F	Nome	Valor-padrão	Descrição	

Registros VSAM do RAM SAMP

Construindo com base no exemplo anterior de RAM SAMP, é possível definir registros para o RAM SAMP no CRADEF conforme mostrado na tabela a seguir.

Tabela 21. Registros do RAM SAMP (um por linha) no CRADEF. Cada célula representa um campo. Consulte "CRADEF" na página 70 para determinar as larguras desses campos.

Tecla				Data				
Um	01	010		000				
Um	01	100		000			000	
Um	01	101					000	
Um	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		

Consulte FEK.SFEKVSM2(CRAINIT) para obter um exemplo do formato de coluna adequado. Esse conjunto de dados sequenciais é usado para inicializar o CRADEF durante a instalação do CARMA. Inicialmente, ele continha registros para o RAM PDS de amostra, o RAM SCLM de amostra e o RAM Esqueleto. Entretanto, dependendo da configuração do host, FEK.SFEKVSM2(CRAINIT) poderá ter sido modificado se RAMs foram incluídos ou removidos do ambiente do CARMA.

Para incluir um RAM no cluster CRADEF, inclua os registros dele em FEK.SFEKVSM2(CRAINIT). Assegure-se de que todas as chaves de registro estejam em ordem alfanumérica para que a operação REPROed do conjunto de dados possa ser bem-sucedida. Use o script da JCL localizado em FEK.#CUST.JCL(CRA\$VDEF) para REPRO FEK.SFEKVSM2(CRAINIT).

É necessário agora definir os registros específicos do código de idioma no CRASTRS. Suponha que o RAM SAMP precise de suporte para inglês e português do Brasil. É possível definir registros para o RAM SAMP no CRASTRS conforme mostrado na tabela a seguir.

Tabela 22. Registros do RAM SAMP (um por linha) no CRASTRS. Cada célula representa um campo. Consulte "CRASTRS" na página 73 para determinar as larguras desses campos. Observe que os registros com uma chave que termina com A01010 não têm dados. Os dados desses registros são opcionais, já que esses registros correspondem a ações padrão que foram substituídas. O CARMA fornecerá ao cliente o nome e a descrição padrão para aquelas ações padrão substituídas.

Chave			Dados			
EN_US	00037	A01010				
EN_US	00037	A01100	Bloquear Instância		Bloqueia a instância	
EN_US	00037	A01101	Remover sinalizador		Remove um sinalizador	
EN_US	00037	A01102	Concatenar		Concatena dois conjuntos de dados	
EN_US	00037	P01000	Razão	Por que não?	Por que você deseja que eu execute a ação?	A razão para executar a ação
EN_US	00037	P01001	ID da Instância-alvo	MyInstance	Em qual instância o membro está localizado?	A instância contendo o membro a ser concatenado
EN_US	00037	P01002	ID do Membro-alvo	MyMember	Que membro você gostaria de concatenar?	O membro a ser concatenado
EN_US	00037	R01000	RAM de amostra		Um RAM de exemplo	
EN_US	00037	T01000	Tipo de Bloqueio		O tipo de bloqueio que o SCM coloca no membro	
EN_US	00037	T01001	Novo ID da Instância		O ID da instância de concatenação	
EN_US	00037	T01002	Novo ID do Membro		O ID do membro da concatenação	
PT_BR	01047	A01010				
PT_BR	01047	A01100	Bloquear Instfncia		Bloqueia a instfncia	
PT_BR	01047	A01101	Tirar sinalizador		Remove um sinalizador	
PT_BR	01047	A01102	Concatenar		Concatena dois conjuntos de dados	
PT_BR	01047	P01000	Motivo	Por que não?	Por que você deseja que eu execute a ação?	O motivo para executar a ação

Tabela 22. Registros do RAM SAMP (um por linha) no CRASTRS. Cada célula representa um campo. Consulte “CRASTRS” na página 73 para determinar as larguras desses campos. Observe que os registros com uma chave que termina com A01010 não têm dados. Os dados desses registros são opcionais, já que esses registros correspondem a ações padrão que foram substituídas. O CARMA fornecerá ao cliente o nome e a descrição padrão para aquelas ações padrão substituídas. (continuação)

Chave			Dados			
PT_BR	01047	P01001	ID de Instfncia de Destino	MyInstance	Em qual instfncia o membro está localizado?	A instfncia que cont,m 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 Instfncia		O ID de instfncia de concatena#o	
PT_BR	01047	T01002	Novo ID do Membro		O ID do membro de concatena#o	

Consulte FEK.SFEKVSM2(CRASINIT) para obter um exemplo do formato de coluna adequado. Esse conjunto de dados sequenciais é usado para inicializar o CRASTRS durante a instalação do CARMA. Como FEK.SFEKVSM2(CRAINIT), inicialmente, ele continha as sequências para o RAM PDS de amostra, o RAM SCLM de amostra e o RAM Esqueleto. Dependendo da configuração do host, FEK.SFEKVSM2(CRASINIT) poderá também ter sido modificado se RAMs foram incluídos ou removidos do ambiente do CARMA.

Para incluir um RAM no cluster CRASTRS, inclua os registros dele em FEK.SFEKVSM2(CRASINIT). Assegure-se de que todas as chaves de registro estejam em ordem alfanumérica para que a operação REPROed do conjunto de dados possa ser bem-sucedida. Use o script da JCL localizado em FEK.#CUST.JCL(CRA\$VSTR) para REPRO FEK.SFEKVSM2(CRASINIT).

Acesso a Clusters do VSAM

Ao editar clusters do VSAM, assegure-se de que nenhum cliente esteja acessando o CARMA. O CARMA pode apresentar um comportamento anormal se o cluster do VSAM for alterado enquanto ele está operando. É recomendável que apenas administradores de sistema e desenvolvedores de RAM tenham acesso de gravação aos clusters do VSAM, mas que todos os usuários tenham acesso de leitura.

Capítulo 5. Desenvolvendo um Cliente CARMA

Os clientes CARMA podem ser projetados para trabalhar especificamente com um RAM, podem fornecer uma interface genérica para qualquer RAM a ser usado ou podem fazer uma combinação dos dois. Um bom exemplo de cliente genérico que pode também ser modificado para trabalhar especificamente com determinados RAMs é o IBM Rational Developer for System z. O Rational Developer foi projetado para suportar as funções básicas que todos os RAMs têm em comum; por isso, um RAM que se ajuste perfeitamente à especificação da API do RAM CARMA funcionaria com o Rational Developer sem complicações. O Rational Developer fornece também pontos de extensão com os quais os desenvolvedores do RAM podem customizar o cliente para seus RAM(s). Por outro lado, um cliente não interativo e extremamente específico poderia ser escrito para apenas executar operações de manutenção por meio de um RAM.

Os clientes CARMA podem usar algumas ou todas as funções básicas da API do CARMA. As únicas funções que é obrigatório implementar são `initCarma`, `initRAM` e `terminateCarma`. `terminateRAM` não é obrigatório porque `terminateCarma` cuidará da limpeza dos RAMs se for chamado e o CARMA ainda tiver RAMs carregados. Entretanto, deve ser tomado um cuidado especial com a memória que é passada de/para o CARMA. Muitas vezes, o RAM alocará memória que o cliente é obrigado a liberar. Leia “Armazenando Resultados para Uso Posterior” na página 80 e “Alocação de Memória” na página 6 cuidadosamente até o fim, visto que fugas de memória e finalização anormal do programa podem facilmente resultar do não seguimento das recomendações sobre a manipulação de memória de cada função.

Compilando o Cliente CARMA

Os clientes CARMA podem incluir o deck paralelo da DLL do CARMA durante a compilação (fazendo com que a DLL do CARMA seja carregada implicitamente) ou podem ser compilados sem o deck paralelo (fazendo com que a DLL do CARMA seja carregada explicitamente). O cliente de exemplo (CRACLISA na biblioteca de amostra) carrega implicitamente a DLL do CARMA. O código JCL para compilar um cliente que carregará implicitamente a DLL do CARMA está no arquivo de amostra denominado CRACLICM.

Executando o Cliente

Ao executar um cliente CARMA, você deve assegurar-se de que o CARMA e todos os RAMs dele tenham os recursos que eles precisam disponíveis a eles. O CARMA requer acesso ao seu cluster de mensagens do VSAM (CRAMSG), aos cluster de CAF do VSAM (CRADEF e CRASTRS) e ao PDS que contém os RAMs. Navegue na JCL usada para executar clientes (CRACLIRN, localizada na biblioteca de amostra) para ver as instruções DD que o CARMA requer (CRASTRS, CRAMSG e CRADEF) e como a DLL do CARMA e o PDS que contém todos os RAMs são incluídos na instrução STEPLIB DD. Os RAMs devem documentar todos os recursos que eles precisam. Por exemplo, o RAM PDS de amostra e o RAM SCLM de amostra exigem cada um que um cluster de mensagens esteja disponível; por isso, a JCL usada para executar o cliente deve ser modificada para que o RAM possa acessar esses recursos. Não fornecer ao CARMA ou aos RAMs o acesso a seus recursos requeridos pode resultar em comportamento anormal.

Ao fornecer recursos aos RAMs, as bibliotecas de mensagens do TSO/ISPF também devem ser consideradas. Os RAMs poderão usar as mensagens do TSO/ISPF se erros ocorrerem. Por padrão, a JCL usada para executar um cliente fornecerá aos RAMs a versão em inglês (página de códigos 00037) dessas mensagens. A JCL deverá ser editada apropriadamente se o RAM precisar retornar as mensagens do TSO/ISPF ao cliente em um idioma diferente.

Armazenando Resultados para Uso Posterior

O cliente deve armazenar os resultados da maioria das operações executadas durante uma sessão do CARMA, especialmente os resultados das funções de navegação, como `getMembers` e `getInstances`. Todas as instâncias, membros simples e contêineres têm um ID e um nome de exibição. O nome de exibição é o que o cliente deve mostrar ao usuário. O nome de exibição para uma entidade deve ser fornecido no contexto da instância dessa entidade e, se aplicável, a todos os contêineres-pai necessários para atingir essa entidade. O ID define a entidade para o RAM exclusivamente. Por exemplo, o ID da entidade pode simplesmente conter seu caminho absoluto. Como alternativa, o RAM pode usar uma função hash para obter do ID o caminho absoluto da entidade. O ID deve ser armazenado pelo cliente para que possa ser novamente passado ao RAM conforme necessário. Por exemplo, um usuário pode obter uma lista de membros em uma instância e, em seguida, verificar se um desses membros é um contêiner.

As outras partes dos dados que podem precisar ser armazenadas pelo cliente (se não forem conhecidas ainda) são chaves de metadados, informações de CAF do RAM e nomes. As informações de CAF do RAM são requeridas por virtualmente cada função que usa um RAM para executar uma operação. As informações de CAF que são obrigatórios podem ser tão simples quanto o ID do RAM pelo qual a ação deve ser executada.

Estruturas de Dados Predefinidas pelo Cliente

Em sua maioria, as funções do RAM são estruturas predefinidas para passar informações de volta ao CARMA e depois ao RAM. Algumas estruturas passam informações sobre um RAM, enquanto outras são usadas para comunicação de/para o RAM. É responsabilidade do Cliente liberar memória usada por essas estruturas, incluindo qualquer matriz de caracteres de comprimento indefinido. Essas matrizes terão terminação nula no estilo C normal.

Ao executar uma ação com relação ao CARMA, o cliente deve ver se a respectiva estrutura `Action` da ação existe para o RAM com o qual está sendo trabalhado. Se a resposta for sim, deve-se usar a estrutura `Action` e as estruturas `Parameter` relacionadas para chamar a ação. Após a ação ser concluída, o cliente deve usar as estruturas `returnValue` relacionadas à ação chamada para analisar adequadamente a resposta da ação.

As estruturas aplicáveis são resumidas nas tabelas a seguir. Essas estruturas estão disponíveis no arquivo de cabeçalho `CRADSDEF` localizado na biblioteca de amostra. Essas estruturas quase sempre são alocadas pelo RAM; por isso, é improvável que o cliente ainda tenha de inicializar qualquer um de seus buffers. Entretanto, o cliente terá de liberar qualquer memória alocada pelo RAM.

As estruturas `RAMRecord` e `RAMRecord2` consistem em um ID do RAM de número inteiro, um campo de caractere `name` de 16 bytes e diversos outros campos de caractere que descrevem o RAM.

Tabela 23. Estruturas de dados RAMRecord e RAMRecord2

Campo		Descrição
RAMRecord	RAMRecord2	
int id	int id	ID exclusivo para descrever o RAM
char name[16]	char* name	Nome de exibição
char version[8]	char* version	Versão do RAM
char reposLevel[8]	char* reposLevel	O nível do SCM que o RAM acessa.
char language[8]	char* language	Linguagem na qual o RAM é escrito
char CRALevel[8]	char* CRALevel	O nível do CARMA ao qual o RAM foi designado.
char moduleName[8]	char* moduleName	Nome do módulo do RAM a ser carregado
char description[2048]	char* description	Exibido como uma descrição do RAM pelo cliente.
não-aplicável	char* uniqueId	Especifica um ID do RAM exclusivo.

A estrutura Descriptor consiste em um campo de caractere name de 64 bytes e um campo de caractere ID de 256 bytes. É usado para descrever instâncias, contêineres e membros simples.

Tabela 24. Estrutura de dados Descriptor

Campo	Descrição
char id[256]	ID exclusivo para descrever a entidade
char name[64]	Nome de exibição

A estrutura DescriptorWithInfo consiste em um campo de caractere name de 64 bytes e um campo de caractere ID de 256 bytes, um número inteiro que armazena o número de pares de valores de chaves e uma matriz de pares de valores de chaves contendo informações de metadados.

Tabela 25. Estrutura de dados DescriptorWithInfo

Campo	Descrição
char id[256]	ID exclusivo para descrever a entidade
char name[64]	Nome de exibição
int infoCount	O número de pares de valores de chaves na matriz de informações
KeyValPair* info	Uma matriz contendo informações de metadados na forma de pares de valores de chaves.

A estrutura MemberDescriptorWithInfo consiste em um campo de caractere name de 64 bytes e um campo de caractere ID de 256 bytes, um número inteiro que armazena o número de pares de valores de chaves e uma matriz de pares de valores de chaves contendo informações de metadados, bem como um número inteiro indicando se o membro é um contêiner.

Tabela 26. Estrutura de dados *MemberDescriptorWithInfo*

Campo	Descrição
char id[256]	ID exclusivo para descrever a entidade
char name[64]	Nome de exibição
int infoCount	O número de pares de valores de chaves na matriz de informações
KeyValPair* info	Uma matriz contendo informações de metadados na forma de pares de valores de chaves.
int isContainer	Um valor igual a 0 indica que o membro não é um contêiner. Um valor igual a 1 indica que ele é um contêiner.

A estrutura *KeyValPair* consiste em um campo-chave de 64 bytes e um campo de valor de 256 bytes. É usado para pares de valores de chaves de metadados.

Tabela 27. Estrutura de dados *KeyValPair*

Campo	Descrição
char key[64]	Um índice
char value[256]	Os dados

As estruturas *Action* e *Action2* consistem em um ID de número inteiro, um nome de 16 bytes, um ponteiro para uma matriz de números inteiros para armazenar os IDs dos parâmetros relacionados à ação, um número inteiro que armazena o número de parâmetros associados à ação, um ponteiro para uma matriz de números inteiros para armazenar os IDs dos valores de retorno relacionados à ação, um número inteiro que armazena o número de valores de retorno associados à ação e uma descrição de 1024 bytes.

Tabela 28. Estruturas de dados *Action* e *Action2*

Campo		Descrição
Action	Action2	
int id	int id	Um identificador numérico para a ação entre 0 e 999. Os IDs de ação entre 0 e 79 substituem ações padrão, enquanto os IDs entre 100 e 999 definem ações customizadas. Os IDs de ação entre 80 e 99 são reservados para uso do CARMA.
char name[16]	char* name	O nome da ação
int* paramArr	int* paramArr	Uma lista dos IDs para os parâmetros que a ação usa
int numParams	int numParams	O número de elementos na matriz paramArr
int* returnArr	int* returnArr	Uma lista dos IDs para os valores de retorno que a ação retorna

Tabela 28. Estruturas de dados Action e Action2 (continuação)

Campo		Descrição
Action	Action2	
int numReturn	int numReturn	O número de elementos na matriz returnArr
char description[1024]	char* description	Uma descrição curta da ação

As estruturas Parameter e Parameter2 consistem em um ID de número inteiro, um nome de 16 bytes, um tipo de 16 bytes, um valor padrão de 16 bytes, um comprimento de número inteiro, um número inteiro especificando se ele é constante (um valor igual a 1 indica que é), um prompt de 1024 bytes e uma descrição de 1024 bytes.

Tabela 29. Estruturas de dados Parameter e Parameter2

Campo		Descrição
Parameter	Parameter2	
int id	int id	Um identificador numérico para o parâmetro entre 0 e 999
char name[16]	char* name	O nome do parâmetro
char type[16]	char* type	O tipo de dados do parâmetro ("INT", "LONG", "DOUBLE" ou "STRING")
char defaultValue[16]	char* defaultValue	O valor padrão do parâmetro
int length	int length	A precisão do parâmetro (se for do tipo "DOUBLE") ou a largura do campo do parâmetro (se for do tipo "STRING"). Se o parâmetro for de algum outro tipo, esse valor poderá ser ignorado.
int isConstant	int isConstant	Se o parâmetro sempre conterá ou não o mesmo valor
char prompt[1024]	char* prompt	O prompt que o cliente CARMA deverá exibir ao solicitar um valor para o parâmetro dos usuários
char description[1024]	char* description	Uma descrição curta do parâmetro

As estruturas returnValue e returnValue2 consistem em um ID de número inteiro, um nome de 16 bytes, um tipo de 16 bytes, um comprimento de número inteiro e uma descrição de 1024 bytes.

Tabela 30. Estruturas de dados *returnValue* e *returnValue2*

Campo		Descrição
returnValue	returnValue2	
int id	int id	Um identificador numérico para o valor de retorno entre 0 e 999
char name[16]	char* name	O nome do valor de retorno
char type[16]	char* type	O tipo de dados do valor de retorno ("INT", "LONG", "DOUBLE" ou "STRING")
int length	int length	A precisão do valor de retorno (se for do tipo "DOUBLE") ou a largura do campo do valor de retorno (se for do tipo "STRING"). Se o valor de retorno for de algum outro tipo, esse valor poderá ser ignorado.
char description[1024]	char* description	Uma descrição curta do valor de retorno

As estruturas *Field* e *Field2* consistem em um ID de número inteiro, uma chave de membro de 64 bytes, um nome de 128 bytes, um valor padrão de 256 bytes e uma descrição de 1024 bytes.

Tabela 31. Estruturas de dados *Field* e *Field2*

Campo		Descrição
Campo	Field2	
int id	int id	Um identificador numérico para o valor de campo entre 0 e 999
char memberKey[64]	char* memberKey	A chave de metadados a ser fornecida à função <i>getMemberInfo</i> para o campo a ser exibido.
char name[128]	char* name	O nome exibível localizado do campo.
char defaultValue[256]	char* defaultValue	O valor exibível localizado do campo se nenhum valor for retornado por uma chamada para <i>getMemberInfo</i> .
char description[1024]	char* description	Uma descrição exibível localizada dos metadados.

Log

O CARMA e os RAMs gravarão mensagens em um log por sessão do CARMA. Ao inicializar o CARMA, um nível de rastreamento deve ser passado a ele. Os níveis de rastreamento são mostrados na Tabela 3 na página 9. A criação de log pode ser desativada enviando ao CARMA um nível de rastreamento igual a -1.

Manipulando Parâmetros e Valores de Retorno Customizados

Parâmetros customizados são passados ao RAM usando o parâmetro `void** params`. `params` é uma matriz de ponteiros `void` que apontam para variáveis de diversos tipos. A função `getCAFData` ou `getCAFData2` retornará as informações de Estrutura de Ação Customizada para todas as funções do RAM. Chame esta antes de executar qualquer outra função do RAM para determinar quais parâmetros e valores de retorno customizados as funções do RAM usam. Os parâmetros customizados obrigatórios devem ser passados ao RAM usando o parâmetro `params`. Se não houver parâmetros customizados obrigatórios, configure `params` como `NULL`. Para preencher `params`, basta designar os ponteiros `void` na matriz a cada parâmetro customizado. Use o seguinte código C como exemplo:

```
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; /*o ponteiro char não deve ser desreferenciado*/
params[2] = (void*) &param2;

/* Chamada de função aqui...*/

free(params);
```

O clientes CARMA devem passar um parâmetro `void***` a todas as funções do RAM definidas para retornar valores de retorno customizados. Pode-se simplesmente passar um ponteiro a uma variável `void**` que você define. Depois que os valores de retorno customizados tiverem sido retornados, eles poderão ser desempacotados conforme demonstra o código C a seguir. É responsabilidade do cliente liberar os retornos customizados:

```
/* Declarado no início */
int return0;
char return1[15];
void ** returnVals = NULL;

/* Chamar a função do CARMA com &returnVals para retornos customizados */

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

/*Liberar cada retorno, além da matriz*/
free(returnVals[0]);
free(returnVals[1]);
free(returnVals);
```

Metadados Definidos pelo CARMA

Extensão de Arquivo Especificada pelo RAM

Ao usar um cliente CARMA, os recursos do CARMA podem obter automaticamente extensões sugeridas da propriedade de metadados especificada pelo RAM. Isso é feito para eliminar a necessidade de o usuário configurar a extensão em cada recurso do CARMA. Entretanto, em alguns casos, uma extensão pode não ser especificada pelo RAM, obrigando o cliente a fornecer uma extensão padrão. Em casos como esse, o cliente deve ser configurado para ignorar a extensão de arquivo fornecida pelo RAM e, em seu lugar, utilizar uma extensão especificado de dentro do cliente. Exemplos de como o cliente pode substituir uma

extensão de arquivo especificada pelo RAM podem ser encontrados em “Extensão de Arquivo Especificada pelo RAM” na página 22.

Extrair para Externo

O CARMA fornece aos clientes a capacidade de extrair arquivos de um SCM em um ambiente normal do host de PDSs e arquivos sequenciais.

copyFromExternal

Copia um membro de um PDS ou um SDS.

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

int ramID	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância que contém o membro sendo copiado
char memberID[256]	Entrada	O ID do membro sendo copiado
char external[256]	Entrada	O local do qual copiar. Um membro PDS ou um membro SDS. Exemplos: FEK.#CUST.EXT.STOR FEK.#CUST.EXT.PDS(MEMBER)
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

copyToExternal

Copia um membro para um PDS ou um SDS.

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

int ramID	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância que contém o membro sendo copiado
char memberID[256]	Entrada	O ID do membro sendo copiado
char target[256]	Entrada	O local para o qual copiar. Um membro PDS ou um membro SDS. Exemplos: FEK.#CUST.EXT.STOR FEK.#CUST.EXT.PDS(MEMBER)
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)

char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.
-----------------	-------	--

Funções de Estado

O CARMA espera que determinadas funções sejam executadas na ordem. Essas funções de estado e sua ordem esperada são:

1. `initCARMA` — O CARMA inicializa diversas variáveis globais, o log de sessão e o código do idioma a ser usado para a sessão com essa função. Essa função não deve ser chamada uma segunda vez, a menos que uma chamada `terminateCarma` seja feita primeiro.
2. `getRAMList` ou `getRAMList2` — Deve ser chamado antes de carregar qualquer RAM, mas os clientes poderão armazenar em cache a lista de RAMs e ignorar essa função, se desejarem. Entretanto, há pouco benefício de desempenho em fazer isso, porque o CARMA executará a função conforme ele precisar da lista ele mesmo.
3. `initRAM` — Deve ser chamado para cada RAM antes de tentar executar qualquer uma das funções desse RAM. Uma vez executado isso, o CARMA manterá um ponteiro para o RAM até a finalização. Os RAMs não devem ser reinicializados sem que primeiro sejam finalizados.
4. `reset` — Poderá ser chamado se o usuário desejar recarregar o ambiente do SCM por causa de uma mudança ocorrida. Isso dirá para o RAM restaurar-se ao seu estado inicial.
5. `terminateRAM` — Essa função não precisa ser chamada. A função `terminateRAM` de cada RAM carregado será chamada por `terminateCarma` se `terminateCarma` for chamado primeiro. Depois que `terminateRAM` é chamado, cada RAM deve ser reinicializado usando a função `initRAM` para que qualquer outra função possa ser chamada para esse RAM.
6. `terminateCarma` — Essa função deve ser chamada sempre ao sair da sessão do CARMA. Ela manipulará a limpeza de todos os RAMs que estão carregados no momento. Uma vez chamada, a função `initCarma` deverá ser executada novamente antes da tentativa de chamar qualquer outra função do CARMA.

`initCarma`

Configurará o ambiente do CARMA, o log de sessão e o código de idioma de sessão

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

int traceLev	Entrada	O nível de rastreo para a sessão atual. Consulte “Log” na página 84 para obter informações adicionais.
char locale[5]	Entrada	Buffer de cinco caracteres com terminação não nula, contendo o código de idioma para o qual todas as sequências exibíveis devem ser configuradas
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Se essa função não for chamada, um código de idioma padrão igual a "EN_US" e um nível de rastreo padrão igual a 0 serão usados.

getRAMList

Recupera a lista de RAMs disponíveis do CARMA

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

RAMRecord** records	Saída	Conterá uma matriz de estruturas de dados RAMRecord a ser usada para exibição de informações sobre os RAMs e acessá-las com outras funções
int* numRecords	Saída	O número de estruturas de dados RAMRecord contidas na matriz records
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

A lista de RAMs que é retornada depende do código de idioma que foi passado no `initializeCarma`. Todos os RAMs armazenados no ambiente do CARMA que tiverem sequências de exibição para o código de idioma do cliente especificado serão retornados.

getRAMList2

Recupera a lista de RAMs disponíveis do CARMA

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

RAMRecord2** records	Saída	Conterá uma matriz de estruturas de dados RAMRecord2 a ser usada para exibição de informações sobre os RAMs e acessá-las com outras funções
int* numRecords	Saída	O número de estruturas de dados RAMRecord2 contidas na matriz records
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

A lista de RAMs que é retornada depende do código de idioma que foi passado no `initializeCarma`. Todos os RAMs armazenados no ambiente do CARMA que tiverem sequências de exibição para o código de idioma do cliente especificado serão retornados.

initRAM

Inicializa um RAM. O CARMA armazenará um ponteiro para o RAM para acesso futuro rápido.

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


int RAMid	Entrada	Informa ao CARMA qual RAM deve ser inicializado. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char locale[8]	Entrada	Informa ao CARMA o código de idioma das sequências que devem ser retornadas ao cliente
char codepage[5]	Entrada	Informa ao CARMA a página de códigos das sequências que devem ser retornadas ao cliente
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

reset

Diz para o RAM reconfigurar-se ao seu estado inicial

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

int RAMid	Entrada	Informa ao CARMA qual RAM deve ser reconfigurado. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

terminateRAM

Diz para o RAM limpar seu ambiente. O CARMA liberará o módulo do RAM.

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

int RAMid	Entrada	Informa ao CARMA qual RAM deve ser finalizado. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

terminateCarma

Limpará o ambiente do CARMA, incluindo os ambientes de qualquer RAM carregado

```
int terminateCarma(char error[256])
```

char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.
-----------------	-------	--

Funções de Navegação

getInstances

Recupera a lista de instâncias disponíveis no SCM

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
Descriptor** RIrecords	Saída	Isso será alocado e preenchido com os IDs e os nomes das instâncias.
int* numRecords	Saída	O número de registros que foram alocados e retornados
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char filter[256]	Entrada	Isso pode ser passado do cliente para filtrar conjuntos de instâncias
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Nota: Certifique-se de liberar a matriz RIrecords

getInstancesWithInfo

Recupera a lista de instâncias disponíveis no SCM e os metadados associados a essas instâncias. Essa é uma função opcional. O cliente deverá chamar essa função se desejar recuperar uma lista de instâncias e os metadados associados a todas essas instâncias. Se o RAM não suportar essa função, o cliente deverá fazer fallback para chamar getInstances.

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
-----------	---------	--

DescriptorWithInfo** records	Saída	Isso será alocado e preenchido com os IDs e os nomes das instâncias. Conterá também as informações de metadados para cada instância.
int* numRecords	Saída	O número de registros que foram alocados e retornados
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char filter[256]	Entrada	Isso pode ser passado do cliente para filtrar conjuntos de instâncias
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Nota: Certifique-se de liberar a matriz de informações de metadados para cada registro retornado. Após a liberação das informações de metadados individuais, certifique-se de liberar a matriz de registros.

getMembers

Recupera a lista de membros disponíveis na instância especificada

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância para a qual os membros devem ser recuperados
Descriptor** memberArr	Saída	Isso será alocado e preenchido com os IDs e os nomes das instâncias.
int* numRecords	Saída	O número de registros que foram alocados e retornados na matriz

void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char filter[256]	Entrada	Isso pode ser passado do cliente para filtrar conjuntos de membros.
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Nota: Certifique-se de liberar a matriz memberArr.

getMembersWithInfo

Recupera a lista de membros disponíveis na instância especificada

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância para a qual os membros devem ser recuperados
MemberDescriptorWithInfo** members	Saída	Isso será alocado e preenchido com os IDs e os nomes dos membros. Conterá também as informações de metadados para cada membro e se esse membro é um contêiner.
int* numRecords	Saída	O número de registros que foram alocados e retornados na matriz
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)

<code>void*** customReturn</code>	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
<code>char filter[256]</code>	Entrada	Isso pode ser passado do cliente para filtrar conjuntos de membros.
<code>char error[256]</code>	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Nota: Certifique-se de liberar a matriz de informações de metadados para cada registro retornado. Após a liberação das informações de metadados individuais, certifique-se de liberar a matriz de membros.

isMemberContainer

Configura `isContainer` como `true` se o membro for um contêiner; caso contrário, `false`

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

<code>int RAMid</code>	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de <code>getRAMList</code> ou <code>getRAMList2</code> .
<code>char instanceID[256]</code>	Entrada	A instância na qual o membro está
<code>char memberID[256]</code>	Entrada	O membro que pode ser um contêiner
<code>int* isContainer</code>	Saída	Configure isso como 1 se o membro for um contêiner; e como 0, se não for.
<code>void** params</code>	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
<code>void*** customReturn</code>	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
<code>char error[256]</code>	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

getContainerContents

Recupera a lista de membros em um contêiner

```
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	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância na qual o membro está
char memberID[256]	Entrada	O contêiner para o qual os membros estão sendo recuperados
Descriptor** contents	Saída	Isso será alocado e preenchido com os IDs e os nomes dos membros dentro do contêiner.
int* numRecords	Saída	O número de registros de membros que foram alocados e retornados na matriz
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char filter[256]	Entrada	Isso pode ser passado do cliente para filtrar conjuntos de membros
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Nota: Certifique-se de liberar a matriz de conteúdo.

getContainerContentsWithInfo

Recupera a lista de membros disponíveis no contêiner especificado e os metadados associados a esses membros. Essa é uma função opcional. O cliente deverá chamar essa função se desejar recuperar uma lista de membros e os metadados associados a todos esses membros. Se o RAM não suportar essa função, o cliente deverá fazer fallback para chamar getContainerContents.

```
int getContainerContentsWithInfo(int RAMid, char instanceID[256],
                                char memberID[256], MemberDescriptorWithInfo** members,
```

```
int* numRecords, void** params, void*** customReturn,  
char filter[256], char error[256])
```

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância na qual o membro está
char memberID[256]	Entrada	O contêiner para o qual os membros estão sendo recuperados
MemberDescriptorWithInfo** members	Saída	Isso será alocado e preenchido com os IDs e os nomes dos membros. Conterá também as informações de metadados para cada membro e se esse membro é um contêiner.
int* numRecords	Saída	O número de registros de membros que foram alocados e retornados na matriz
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char filter[256]	Entrada	Isso pode ser passado do cliente para filtrar conjuntos de membros
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Nota: Certifique-se de liberar a matriz de informações de metadados para cada registro retornado. Após a liberação das informações de metadados individuais, certifique-se de liberar a matriz de membros.

Criar/Excluir

Criar e excluir fornecem a funcionalidade para criar e excluir membros e contêineres em um ambiente do CARMA.

createMember

Cria um novo membro

```
int createMember(int RAMid, char instanceID[256], char memberID[256], char name[64],  
char parentID[256], int* lrec1, char recFM[4], void** params,  
void*** customReturn, char error[256]);
```

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância que contém o membro sendo criado
char memberID[256]	Saída	O ID do membro que está sendo criado
char name[64]	Entrada/Saída	O ID do membro sendo criado
char parentID[256]	Entrada	O ID do contêiner-pai (Se nenhum pai existir, deve ser preenchido com espaço)
int* lrecl	Saída	O número de colunas no conjunto de dados e na matriz
char recFM[4]	Saída	Contém o formato de registro do conjunto de dados (FB, VB, ect)
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte "Manipulando Parâmetros e Valores de Retorno Customizados" na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte "Manipulando Parâmetros e Valores de Retorno Customizados" na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Para considerar as convenções de nomenclatura específicas do RAM, um cliente chama a operação criar solicitando um determinado nome. O RAM pode então fornecer um memberID, lrecl, recFM e um nome exibível apropriado exclusivos de volta ao cliente.

Se o cliente solicitar o nome "bob", por exemplo, o RAM poderá retornar um memberID igual a "BOB", bem como um nome exibível igual a "BOB". Se o membro "bob" já existir, ele poderá retornar "BOB2", ou então retornar um erro informando que não é possível criar o membro solicitado.

parentID é o memberID para o pai do membro que está sendo criado. Se o membro sendo criado não tiver um pai (está diretamente na instância de repositório), parentID deverá ser deixado em branco (tudo com espaços).

Um RAM não precisa criar um membro quando createMember é chamado, mas pode apenas fornecer o memberID, lrecl, recFM e o nome exibível adequados ao cliente. É responsabilidade do cliente fazer uma chamada para putMember com o

novo memberID a fim de criar um membro concreto. Os RAMs devem suportar a inclusão de um membro sem registros (mesmo que tenham de criar um único registro em branco para o membro).

createContainer

Cria um novo contêiner

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância que contém o contêiner sendo criado
char memberID[256]	Saída	O ID do membro que está sendo criado
char name[64]	Entrada/Saída	O ID do contêiner sendo criado
char parentID[256]	Entrada	O ID do contêiner-pai (Se nenhum pai existir, deve ser preenchido com espaço)
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte "Manipulando Parâmetros e Valores de Retorno Customizados" na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte "Manipulando Parâmetros e Valores de Retorno Customizados" na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Para considerar as convenções de nomenclatura específicas do RAM, um cliente chama a operação criar solicitando um determinado nome. O RAM pode então fornecer um memberID, lrecI, recFM exclusivos e um nome exibível apropriado de volta ao cliente.

Se o cliente solicitar o nome "bob", por exemplo, o RAM poderá retornar um memberID igual a "BOB", bem como um nome exibível igual a "BOB". Se o contêiner "bob" já existir, ele poderá retornar "BOB2", ou então retornar um erro informando que não é possível criar o contêiner solicitado.

parentID é o memberID para o pai do contêiner que está sendo criado. Se o contêiner sendo criado não tiver um pai (está diretamente na instância de repositório), parentID deverá ser deixado em branco (tudo com espaços).

Ao contrário da função `createMember`, quando `createContainer` é chamado, o contêiner deve sempre ser criado imediatamente pelo RAM, a menos que um erro ocorra.

delete

Exclui um membro ou contêiner

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de <code>getRAMList</code> ou <code>getRAMList2</code> .
char instanceID[256]	Entrada	A instância que contém o membro ou o contêiner sendo excluído
char memberID[256]	Entrada	O ID do membro que está sendo excluído
int force	Entrada	Usado para forçar uma exclusão. Um valor igual a 1 forçará uma exclusão
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

O parâmetro `force` pode ser configurado como 1 para ordenar a um RAM que exclua um membro que ele normalmente não excluiria, como um contêiner não vazio. Se um RAM puder excluir um item, mas precisar de um parâmetro `force` para fazê-lo, ele poderá enviar um determinado código de retorno, com um erro apropriado, para informar ao cliente. O cliente pode então oferecer a opção de excluir com o parâmetro `force`.

Como opção, o cliente pode também permitir que o usuário configure o parâmetro `force` antes de chamar a exclusão.

A função de exclusão pode ser usada para excluir membros e contêineres; entretanto, não deve ser usada para excluir uma Instância do RAM.

Funções de Transferência de Arquivo

extractMember

Recupera o conteúdo do membro

```
int extractMember(int RAMid, char instanceID[256],
                 char memberID[256], char*** contents, int* lrec1,
                 int* numRecords, char recFM[4], int* moreData,
                 int* nextRec, void** params, void*** customReturn,
                 char error[256])
```

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância que contém o membro
char memberID[256]	Entrada	O ID do membro que está sendo extraído
char*** contents	Saída	Será alocado como matriz bidimensional para manter o conteúdo do membro
int* lrec1	Saída	O número de colunas no conjunto de dados e na matriz
int* numRecords	Saída	O número de registros no conjunto de dados ou o número de linhas na matriz
char recFM[4]	Saída	Conterá o formato de registro do conjunto de dados (FB, VB, etc.)
int* moreData	Saída	Configure como 1 o valor da variável para o qual isso aponta, se a extração tiver de ser chamada novamente (por ainda haver mais dados a serem extraídos). Caso contrário, designe como 0 o valor para o qual ela aponta.
int* nextRec	Entrada/Saída	Entrada: O registro do membro no qual o RAM deve iniciar extração Saída: O primeiro registro no conjunto de dados que não foi extraído se *moreData estiver configurado como 1; caso contrário, indefinido
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)

char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.
-----------------	-------	--

O buffer de conteúdo é uma matriz de caracteres bidimensional que será preenchida pelo RAM e retornada ao cliente. Para a primeira chamada `extractMember`, `nextRec` deverá ser 0. O RAM pode optar por retornar os dados em chunks de registros. A extração deve ser chamada até que `moreData` seja 0. Se `moreData` for 1, `extractMember` precisará ser chamado novamente, e a extração do membro será iniciada com o registro indexado pelo valor de `nextRec` retornado na chamada anterior. O RAM precisará que o cliente passe esse valor de `nextRec` de volta para a chamada seguinte.

Consulte Capítulo 3, “Desenvolvendo um RAM”, na página 13 para obter um exemplo da operação de `extractMember` do ponto de vista do RAM.

Nota: Certifique-se de liberar `contents` adequadamente. Ele foi alocado como um grande chunk de dados contíguos, por isso deve ser liberado da seguinte maneira (o exemplo é em C):

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

putMember

Atualiza o conteúdo de um membro ou cria um novo membro se o `memberID` especificado não existir na instância

```
int putMember(int RAMid, char instanceID[256],
              char memberID[256], char** contents, int lrec1,
              int* numRecords, char recFM[4], int moreData,
              int nextRec, int eof, void** params, void*** customReturn,
              char error[256])
```

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de <code>getRAMList</code> ou <code>getRAMList2</code> .
char instanceID[256]	Entrada	A instância que contém o membro
char memberID[256]	Entrada	O ID do membro que está sendo atualizado/criado
char** contents	Entrada	Mantém o novo conteúdo do membro
int lrec1	Entrada	O número de colunas no conjunto de dados e na matriz
int* numRecords	Entrada/Saída	O número de registros no conjunto de dados ou o número de linhas na matriz
char recFM[4]	Entrada	Contém o formato de registro do conjunto de dados (FB, VB etc.)
int moreData	Entrada	Será 1 se o cliente tiver mais chunks de dados para enviar; caso contrário, 0

int nextRec	Entrada	O registro no conjunto de dados para o qual o registro 0 da matriz de conteúdo é mapeado
int eof	Entrada	Se 1, indica que a última linha da matriz deverá marcar a última linha no conjunto de dados; caso contrário, 0.
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

O cliente pode escolher um tamanho de chunk para a função ou tentar passar todo o conteúdo do arquivo de uma vez. O cliente pode também optar por dar voltas dentro de um arquivo. Por exemplo, os registros de 0 a 15 podem ser passados primeiro, de 40 a 50 depois e, em seguida, 16 a 30. Entretanto, nem todos os RAMs podem manipular chunks de dados não sequenciais dessa forma adequadamente.

Para envio de dados em chunks, `moreData` deve ser 1 em cada chamada até a chamada final, durante a qual deve ser 0. `nextRec` deve sempre ser configurado como o primeiro registro a ser atualizado no membro. Lembre-se de que isso usa um índice baseado em 0. `eof` é usado para especificar que o registro do membro em `nextRec + numRecords` deve ser o último no membro atualizado. Por exemplo, se essa soma for 15 e houver atualmente 30 registros no membro, os registros de 16 a 29 serão excluídos pelo RAM depois que ele atualizar até o registro 15.

Consulte a origem do cliente de amostra (CRACLISA na biblioteca de amostra) para obter mais ajuda.

Nota: O buffer de conteúdo deve ser alocado antes da chamada de maneira semelhante à seguinte (o exemplo é em C):

```
contents = (char**) malloc(sizeof(char*) * (numRecords));
*contents = (char*) malloc(sizeof(char) * (lrec1) * (numRecords));
for(i = 0; i < numRecords; i++)
    (contents)[i] = ((*contents) + (i * (lrec1)));
```

e deve ser liberado após a chamada de maneira semelhante à seguinte (o exemplo é em C):

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

Transferência de Arquivo Binário

extractBinMember

Recupera o conteúdo do membro.

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância que contém o membro sendo extraído.
char memberID[256]	Entrada	O ID do membro que está sendo extraído.
char** contents	Saída	Ponteiro para o conteúdo do membro
int* length	Saída	O comprimento do conteúdo do membro.
int* moreData	Saída	Se a extração precisar ser chamada novamente porque há mais dados, configure como 1 o valor da variável para o qual isso aponta, ou então designe como 0 o valor para o qual ela aponta.
int start	Entrada	O local de byte do arquivo do qual iniciar a extração.
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

putBinMember

Atualiza o conteúdo de um membro ou cria um novo membro se o memberID especificado não existir na instância.

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância que contém o membro sendo atualizado/criado.
char memberID[256]	Entrada	O ID do membro que está sendo atualizado/criado.
char* contents	Entrada	Mantém o novo conteúdo dos membros.
int length	Entrada	Ponteiro para o comprimento dos dados a serem gravados.
int moreData	Entrada	Será 1 se o cliente tiver mais chunks de dados para enviar; caso contrário, 0.
int start	Entrada	O local de byte do arquivo para iniciar a colocação de dados.
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Funções de Metadados

getAllMemberInfo

Recupera todos os metadados para o membro determinado

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância na qual o membro está

char memberID[256]	Entrada	O ID do membro para o qual metadados estão sendo retornados. O ID poderá ficar vazio se as informações do membro forem recuperadas para a instância e não para um membro específico.
KeyValPair** metadata	Saída	Isso será alocado e preenchido com chaves e valores dos metadados.
int* num	Saída	O número de estruturas de metadados KeyValPair alocado e retornado na matriz
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Nota: Certifique-se de liberar a matriz de metadados.

getFieldsData

Recupera os dados dos campos para o determinado RAM. Os campos fornecem sugestões para metadados que devem ser exibidos ao usuário.

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

int RAMid	Entrada	Informa ao CARMA para qual RAM reunir dados. Esse ID foi obtido após a execução de getRAMList.
Field** fields	Saída	Isso será alocado e preenchido com ID, chave de metadados, nome, valor padrão e descrição de cada campo.
int * numFields	Saída	O número de estruturas de Campo alocadas e preenchidas na matriz.
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Nota: Certifique-se de liberar a matriz de campos.

getFieldsData2

Recupera os dados dos campos para o determinado RAM. Os campos fornecem sugestões para metadados que devem ser exibidos ao usuário.

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

int RAMid	Entrada	Informa ao CARMA para qual RAM reunir dados. Esse ID foi obtido após a execução de getRAMList2.
Field2** fields	Saída	Isso será alocado e preenchido com ID, chave de metadados, nome, valor padrão e descrição de cada campo.
int * numFields	Saída	O número de estruturas de Campo alocadas e preenchidas na matriz.
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Nota: Certifique-se de liberar a matriz de campos.

getMemberInfo

Recupera uma parte específica de metadados para o membro determinado

```
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	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância na qual o membro está
char memberID[256]	Entrada	O membro para o qual metadados estão sendo recuperados
char key[64]	Entrada	A chave do valor de metadado a ser recuperado
char value[256]	Saída	O valor que está sendo recuperado
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)

void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

updateMemberInfo

Atualiza uma parte específica de metadados para o membro determinado

```
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	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância na qual o membro está
char memberID[256]	Entrada	O membro para o qual metadados estão sendo configurados
char key[64]	Entrada	A chave do valor de metadado a ser configurada
char value[256]	Entrada	O valor que está sendo configurado
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Outras Operações

lock

Bloqueia o membro

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância na qual o membro está
char memberID[256]	Entrada	O membro a ser bloqueado
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

unlock

Desbloqueia o membro

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância na qual o membro está
char memberID[256]	Entrada	O membro a ser desbloqueado
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

checkin

Efetue o registro de entrada do membro. Isso apenas configura um sinalizador. Uma chamada putMember é esperada imediatamente após essa chamada.

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância na qual o membro está
char memberID[256]	Entrada	O membro cujo registro de entrada é efetuado
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

checkout

Efetue o registro de saída do membro. Isso apenas configura um sinalizador. Uma chamada extractMember é esperada imediatamente após essa chamada.

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância na qual o membro está
char memberID[256]	Entrada	O membro cujo registro de saída é efetuado
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)

void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

performAction

Instrui o RAM especificado a retornar a ação identificada no actionID usando os parâmetros fornecidos e os valores de retorno em customReturn (quando aplicável).

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

int RAMid	Entrada	Informa ao CARMA o RAM no qual trabalhar. Esse ID foi obtido após a execução de getRAMList ou getRAMList2
int actionID	Entrada	A ação customizada que está sendo solicitada, conforme definido no CRADEF VSAM.
char instanceID[256]	Entrada	A instância na qual a ação está sendo executada. O ID poderá ficar em branco caso se espere que a ação seja executada no próprio RAM, em vez de em uma instância ou membro específico.
char memberID[256]	Entrada	O membro no qual a ação está sendo executada. O ID poderá ficar em branco caso se espere que a ação seja executada em uma instância ou no próprio RAM, em vez de em um membro específico.
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte “Manipulando Parâmetros e Valores de Retorno Customizados” na página 85)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

getCAFData

Recupera os dados do CAF para o RAM solicitado

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

Tabela 32. Parâmetros getCAFData

int RAMid	Entrada	Informa ao CARMA para qual RAM os dados do CAF devem ser extraídos. Esse ID foi obtido após a execução de getRAMList.
Action** actions	Saída	Isso será alocado e preenchido com as ações customizadas para o determinado RAM.
int* numActions	Saída	O número de ações sendo retornadas
int** disabledActions	Saída	Isso será alocado e preenchido com as ações desativadas para o determinado RAM.
int* numDisabled	Saída	O número de ações desativadas sendo retornadas
Parameter** params	Saída	Isso será alocado e preenchido com os parâmetros customizados para o determinado RAM.
int* numParams	Saída	O número de parâmetros sendo retornados
returnValue** returnVals	Saída	Isso será alocado e preenchido com os valores de retorno customizados para o determinado RAM.
int* numReturn	Saída	O número de valores de retorno sendo retornados
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Consulte o Capítulo 4, “Customizando uma API do RAM Usando o CAF”, na página 61 para obter mais informações sobre os tipos de dados que podem ser retornados. Os dados retornados devem ser armazenados durante o restante da sessão para que possam ser verificados antes de qualquer chamada de função para o respectivo RAM.

getCAFData2

Recupera os dados do CAF para o RAM solicitado

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

```

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

```

Tabela 33. getCAFData2 parameters

int RAMid	Entrada	Informa ao CARMA para qual RAM os dados do CAF devem ser extraídos. Esse ID foi obtido após a execução de getRAMList2.
Action2** actions	Saída	Isso será alocado e preenchido com as ações customizadas para o determinado RAM.
int* numActions	Saída	O número de ações sendo retornadas
int** disabledActions	Saída	Isso será alocado e preenchido com as ações desativadas para o determinado RAM.
int* numDisabled	Saída	O número de ações desativadas sendo retornadas
Parameter2** params	Saída	Isso será alocado e preenchido com os parâmetros customizados para o determinado RAM.
int* numParams	Saída	O número de parâmetros sendo retornados
returnValue2** returnVals	Saída	Isso será alocado e preenchido com os valores de retorno customizados para o determinado RAM.
int* numReturn	Saída	O número de valores de retorno sendo retornados
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

Consulte o Capítulo 4, “Customizando uma API do RAM Usando o CAF”, na página 61 para obter mais informações sobre os tipos de dados que podem ser retornados. Os dados retornados devem ser armazenados durante o restante da sessão para que possam ser verificados antes de qualquer chamada de função para o respectivo RAM.

getVersionList

Fornece uma lista de versões disponíveis para um determinado membro

```

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

```

Tabela 34. *getVersionList* parameters

int RAMid	Entrada	Informa ao CARMA para qual RAM os dados do CAF devem ser extraídos. Esse ID foi obtido após a execução de getRAMList ou getRAMList2.
char instanceID[256]	Entrada	A instância na qual o membro está
char memberID[256]	Entrada	O membro para o qual obter uma lista de versões
VersionIdent** versions	Saída	Uma lista de todas as versões do membro disponíveis. Deve ser uma lista ordenada, com a versão 'mais recente' primeiro e a versão mais antiga por último.
int* num	Saída	O número de versões
void** params	Entrada	Ponteiro para uma matriz de parâmetros customizados (consulte "Manipulando Parâmetros e Valores de Retorno Customizados" na página 21)
void*** customReturn	Saída	Usado para referenciar uma matriz de valores de retorno customizados (consulte "Manipulando Parâmetros e Valores de Retorno Customizados" na página 21)
char error[256]	Saída	Se um erro ocorrer, deverá ser preenchido com uma descrição do erro.

VersionIdent será identificado pela seguinte estrutura:

```
typedef struct {
char memberID[256]; /*Um memberID com versão, como
                    baseMemberID VerNum*/
char versionKey[64]; /* Uma maneira de se referir à versão, como
                    "1.2.3"...deve ser igual ao valor
                    para a chave de metadados carma.version*/
char comments[256]; /* Os comentários fornecidos pelo RAM sobre a versão,
                    podem ser registro de data/hora, mudanças, etc.. */
} VersionIdent;
```

A lista de versões será uma lista completa de versões ordenadas, mas o Desenvolvedor do RAM pode optar por usar um ID 'com versão' para a versão atual ou usar o ID inalterado. Como exemplo, a versão atual de um membro pode ser acessível por meio de "location(Member)" ou "location(Member)_1.4", em que o arquivo está na versão 1.4. O desenvolvedor do RAM pode portanto optar por retornar "location(Member)_1.4" ou "location(Member)" como a versão mais recente na lista.

Ao retornar uma lista de membros por meio das funções de navegação, como `getMembers`, os RAMs NÃO DEVEM incluir a versão no `memberID`. Alterar o `memberID` para um membro impede que os clientes CARMA rastreiem adequadamente esse membro.

Para suportar versão, os Desenvolvedores do RAM devem manipular as chamadas do CARMA quando apresentadas com um ID 'com versão' para o `memberID`.

Os clientes devem suportar o código de retorno 130, que significa "O membro não suporta versão"

Os clientes podem suportar uma variedade de chamadas com relação a membros com versão, como as funções de transferência de arquivo e as funções de metadados.

Apêndice A. Códigos de Retorno

Código de Retorno	Descrição
20	Erro interno
22	Nenhuma RAM definida para este código do idioma
24	CRADEF não pôde ser aberto para leitura
25	CRSTRS não pôde ser aberto para leitura
26	Nenhum registro localizado em CRADEF
28	Erro de leitura do CRADEF
30	(marcador)
32	Foi encontrado registro inválido do CRADEF
34	RAM solicitado não localizado
36	Não foi possível carregar o módulo do RAM
38	Não foi possível carregar o ponteiro para a função do RAM
40	O RAM solicitado <i>nome do RAM</i> não foi carregado
42	Foi encontrado registro inválido do CRSTRS
44	O CARMA não foi inicializado
46	Falha ao tentar carregar a lista RAM
48	Sem memória
50	O registro no CRADEF não tem equivalente no CRSTRS para esse código de idioma
52	A ação faz referência a parâmetro desconhecido
54	A ação faz referência a tipo de retorno desconhecido
56	Erro de leitura do CRSTRS
58	Não foi possível localizar nem o código do idioma especificado nem o padrão (EN_US, página de códigos 00037) no CRSTRS
60	CRMSG não localizado
62	Erro de leitura do CRMSG
64	Erro do CRADEF: A ação 16 não pode ter parâmetros e/ou retornos customizados
66	Tipo inválido especificado no registro VSAM
68	Valor padrão inválido no registro VSAM
101	Não foi possível alocar memória
102	Funções da biblioteca TSO/ISPF não disponíveis
103	Identificador de membro inválido
104	Não é possível alocar (sem espaço)

Código de Retorno	Descrição
105	Membro não localizado
106	Instância não localizada
107	Função não suportada
108	O membro não é um contêiner
109	Valor de parâmetro inválido
110	O membro não pode ser atualizado
111	O membro não pode ser criado
112	Não Autorizado
113	Não foi possível inicializar
114	Não foi possível finalizar
115	Recurso fora de sincronização
116	Arquivo bloqueado
117	Próximo registro especificado fora do intervalo
118	Formato de registro não suportado
119	LRECL inválido
120	Chave de metadados inválida
121	Não é possível atualizar o valor da propriedade
122	Valor de metadados inválido
123	O valor da propriedade é de leitura
124	O membro solicitado está vazio
125	Instância vazia
126	Nenhum membro localizado
127	Erro de reconfiguração
128	Erro de exclusão
129	Membro/Versão é somente leitura
130	O membro não suporta versão
197	(mensagem de erro ISPF/LMF encapsulada)
198	Impossível acessar o arquivo de log
199	Erro desconhecido de RAM
222	Erro ao recuperar lista de parâmetros da Estrutura da Ação Customizada
223	Está faltando um parâmetro esperado da Estrutura de Ação Customizada
224	Tipo de dados desconhecido especificado para o parâmetro da Estrutura de Ação Customizada
225	Erro ao recuperar valores de retorno da Estrutura da Ação Customizada

Apêndice B. IDs de Ação

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

Nota: * Se as funções "WithInfo" precisarem dos parâmetros customizados ou devoluções customizadas, configure a função para usar os mesmos parâmetros customizados e devoluções customizadas como funções base. Este requisito permite que o cliente CARMA chame a função apropriada sem entrada adicional do usuário.

Apêndice C. RAMs de Amostra

Este apêndice funciona como um recurso para os RAMs de amostra que são enviados com o Common Access Repository Manager (CARMA). Os RAMs de amostra são fornecidos com o propósito de testar a configuração do ambiente CARMA e como exemplos para que você desenvolva seus próprios RAMs. **NÃO utilize os RAMs de amostra fornecidos em um ambiente de produção.**

PDS RAM

Descrição do RAM

O RAM Partitioned Data Set (PDS) permite que você acesse um PDS associado a usuários do TSO alavancando os serviços do ISPF. Nesse caso, os serviços do TSO/ISPF são o SCM e o repositório é o PDS do usuário.

Estrutura de Navegação

No RAM PDS, o CARMA exibe uma lista de todos os conjuntos de dados PDS que estão disponíveis a você em uma conexão específica. Cada PDS pode então ser expandido para exibir uma coleção de Conjuntos de Dados Sequenciais (SDS), também chamados membros que compõem cada PDS.

Ações Suportadas

As seguintes ações estão disponíveis atualmente apenas para arquivos com um formato de registro de "Bloco Fixo".

- Extract
- Upload Local File
- Replace Local File

Ações Não-Suportadas

As seguintes ações do CARMA não são suportadas pelo PDS RAM, pois não possui recursos de controle de versão:

- Lock
- Unlock
- Registro de Saída
- Registro de Entrada

RAM do SCLM

Descrição do RAM

O RAM de amostra do Software Configuration Library Manager (SCLM) é outra demonstração da capacidade do CARMA de fazer interface com os Source Code Managers (SCMs). O propósito deste apêndice é dar ao desenvolvedor do RAM um entendimento da implementação do RAM SCLM do CARMA. O RAM SCLM faz interface com um IBM Software Configuration and Library Manager (SCLM).

O IBM SCLM fornece uma API semelhante ao ISPF que fornece o gerenciador de diálogo. Além disso, o SCLM faz interface com os serviços de gerenciamento de

biblioteca do ISPF para a maioria de suas funções. O ISPF/SCLM cria e acessa variáveis, listas e relatórios como resultado das chamadas de API que ele faz. Para obter uma descrição completa de todos os serviços de programação do ISPF/SCLM, consulte o Software Configuration Library Manager Reference, z/OS Versão 1 Liberação 7.0, e também o manual de serviços do ISPF para obter informações detalhadas sobre serviços de gerenciamento de biblioteca do ISPF. O CARMA utiliza os serviços de gerenciamento de biblioteca do ISPF e os serviços do SCLM no RAM de amostra do SCLM.

Estrutura de Navegação

No RAM SCLM, o CARMA exibe um projeto do SCLM selecionado que está disponível a você em uma conexão específica. Cada projeto do SCLM pode então ser expandido para exibir os grupos e os tipos associados ao projeto.

Ações Suportadas

O RAM de amostra do SCLM usa as principais funções do ISPF e do SCLM na Interface com o Usuário do CARMA. Essa funcionalidade permite que os usuários enviem solicitações ao RAM SCLM no host do z/OS e, em seguida, exibam os resultados em suas estações de trabalho. Segue uma lista de funções do SCLM que podem ser chamadas a partir de uma seleção do membro na UI do CARMA.

Tabela 35. Funções Básicas

Nome da Função	Descrição
LOCK	Essa é uma função independente que permite a um usuário bloquear um membro ou incluir uma chave de acesso para limitar ou restringir o acesso a ela por outros usuários. Essa função pode ser ativada clicando com o botão direito do mouse em um membro do SCLM na UI do CARMA e selecionando Bloquear no menu de contexto.
UNLOCK	Esta função desbloqueará um membro que foi bloqueado removendo a chave de acesso. Ela pode ser acessada clicando com o botão direito do mouse no membro bloqueado na UI do CARMA e selecionando Desbloquear no menu de contexto.
DELETE	Essa função excluirá todos os rastreios de um membro do SCLM, incluindo todo texto e metadados, de um projeto do SCLM. Essa função pode ser acessada clicando com o botão direito do mouse em um membro do SCLM na UI do CARMA e selecionando Excluir no menu de contexto.

As funções a seguir são ações customizadas que são específicas do conteúdo do RAM SCLM. Elas podem ser acessadas clicando com o botão direito do mouse em um membro do SCLM na UI do CARMA e selecionando Customizado no menu de contexto.

Nota: Os comandos customizados a seguir solicitarão aos usuários parâmetros adicionais.

Tabela 36. Ações Customizadas

Nome do Serviço	Descrição
MIGRATE	O serviço MIGRATE cria ou atualiza as informações de conta do SCLM para membros em uma biblioteca de desenvolvimento. A correspondência de padrões não é fornecida nesse momento.

Tabela 36. Ações Customizadas (continuação)

Nome do Serviço	Descrição
BUILD	O serviço BUILD compila, vincula e integra componentes de software de acordo com a definição de arquitetura do projeto. Antes que um membro seja construído entretanto, as informações de dependência do membro devem existir no banco de dados do projeto. Por essa razão, o serviço STORE ou SAVE para um membro deve ser concluído com êxito para que o serviço BUILD possa ser executado.
PROMOTE	O serviço PROMOTE move dados, ou promove dados, por meio do banco de dados do projeto de acordo com a definição e a arquitetura do projeto. Para que o SCLM possa promover um membro, ele deve ter uma chave de acesso em branco, além de ter concluído com êxito o serviço BUILD. Se um membro tiver uma chave de acesso, você deverá chamar o serviço UNLOCK para reconfigurar a chave a fim de que seja possível promover o membro.
DELETE	O serviço DELETE exclui os componentes do banco de dados. Você pode excluir um membro inteiro, seus registros de conta associados e o mapa de construção, apenas os registros de conta e o mapa de construção ou simplesmente o mapa de construção do membro.
EDIT	O serviço EDIT não é como a edição do SCLM. Em vez disso, é usado para anunciar a intenção de uma edição. O usuário será solicitado a informar o grupo de desenvolvimento para o qual mover o membro. Uma atualização é necessária para que o usuário possa selecionar o membro no nível de desenvolvimento, clicando duas vezes no membro. Nesse ponto, o código de origem aparecerá no painel de edição da UI.

Seguem os serviços da API do SCLM que o RAM SCLM usa para fornecer funcionalidade.

Tabela 37. Serviços Integrados do SCLM

Nome do Serviço	Descrição
SCLMINFO	O serviço SCLMINFO é usado pela função getmembers do CARMA para recuperar todos os grupos e tipos do SCLM armazenados nas variáveis do ISPF para recuperação posterior de getContainer.
SAVE	O serviço SAVE bloqueia e analisa um membro e depois prossegue para armazenar as informações estatísticas, de dependência e históricas desse membro, tudo em uma só chamada. O serviço SAVE chamado na função putMember do CARMA chama os serviços do SCLM LOCK, PARSE e STORE.

Os serviços a seguir do SCLM mantêm a integridade de sessão nas funções existentes do CARMA.

Tabela 38. Serviços do SCLM

Nome do Serviço	Descrição
INIT	O serviço INIT inicializa um ID do SCLM. Durante esse processo, ele também inicializa a definição de projeto especificada.
START	O serviço START inicializa uma sessão de serviços do SCLM. Ele gera um ID do aplicativo que identifica a sessão de serviços.

Tabela 38. Serviços do SCLM (continuação)

Nome do Serviço	Descrição
END	O serviço END para uma sessão de serviço do SCLM e libera um ID de aplicativo gerado pelo serviço START. Cada chamada de serviço START precisa de uma chamada de serviço END correspondente. Esse serviço também chama o serviço FREE para liberar qualquer ID do SCLM associado ao determinado ID do aplicativo que não foi liberado explicitamente.
FREE	O serviço FREE libera um ID do SCLM gerado pelo serviço INIT. Cada chamada de serviço INIT precisa de uma chamada de serviço FREE correspondente. Depois de liberar o ID do SCLM, o SCLM fecha todos os conjuntos de dados do projeto e libera a definição de projeto especificada no serviço INIT.

Ações Não Suportadas

O SCLM RAM não suporta as ações a seguir. A tentativa de execução de qualquer uma destas ações resultará em um diálogo de erro.

- Registro de Saída

Para obter acesso exclusivo a um arquivo de origem para edição, utilize a ação Lock. Outros usuários ainda poderão acessar o arquivo de origem extraíndo-o do repositório, mas eles não poderão efetuar registro de entrada de suas atualizações para este arquivo até desbloqueá-lo.

- Registro de Entrada

Para permitir que outro usuário edite um arquivo de origem, utilize a ação Unlock.

RAM COBOL

Descrição do RAM

O RAM COBOL é uma implementação do RAM PDS escrito em COBOL. É constituído de uma DLL que resulta do link da origem COBOL e C compilada. O RAM COBOL fornece funcionalidade para navegar em ativos PDS da mesma maneira que o RAM de amostra PDS. Alguma funcionalidade presente no RAM PDS não é implementada, mas programas esqueleto são fornecidos para implementação de funcionalidade adicional.

Estrutura de Navegação

No RAM COBOL, o CARMA exibe uma lista de todas as instâncias PDS disponíveis com o qualificador de alto nível do usuário no host do CARMA. Cada instância pode assim ser expandida para exibir a lista de membros que compõem cada instância.

Recursos Suportados

As funções a seguir já estão configuradas no RAM COBOL de amostra. Dependendo do que o RAM COBOL estará suportando, funções adicionais podem precisar ser implementadas.

- extractMember

Extraí um membro da mesma maneira que um RAM PDS. A função é codificada de modo que a extração de qualquer membro associado a uma instância do

RAM COBOL retornará os registros de um conjunto de dados referenciado pela DD CBLIN. Essa DD deve ser incluída no CLIST de inicialização do CARMA para que `extractMember` funcione.

- `putMember`
Armazena um membro PDS à instância PDS especificada.
- `getInstances`
Fornecer uma lista de instâncias PDS.
- `getMembers`
Retorna a lista de membros associados a uma instância PDS.
- `initRAM`
Configura variáveis globais e demonstra a função de criação de log de COBOL para C.
- `performAction`
Contém código de amostra para executar uma ação customizada. A amostra para `performAction` aceita parâmetros customizados e, em seguida, fornece-os como retornos customizados em ordem inversa. As informações de configuração do CAF para usar a ação customizada podem ser encontradas na documentação do programa para `performAction`.

Para obter informações mais detalhadas, consulte “Desenvolvimento do RAM Usando COBOL” na página 48

RAM Esqueleto

Descrição do RAM

O RAM Esqueleto é o mais básico dentre todas as amostras. Ele fornece uma estrutura simples que pode ser usada para desenvolver um RAM que atenderá suas necessidades. Esse RAM deve ser usado como ponto de início para desenvolver seu próprio RAM customizado.

Notas de Documentação para IBM Rational Developer for System z

© Copyright IBM Corporation - 2010

Direitos Restritos para Usuários do Governo dos Estados Unidos - Uso, duplicação e divulgação restritos pelo documento GSA ADP Schedule Contract com a IBM Corp.

Gerência de Relações Comerciais e Industriais da IBM Brasil
IBM Corporation
Botafogo
Rio de Janeiro, RJ

Para pedidos de licenças com relação a informações sobre DBCS (Conjunto de Caracteres de Byte Duplo), entre em contato com o Departamento de Propriedade Intelectual da IBM em seu país ou envie pedidos de licença, por escrito, para:

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

O parágrafo a seguir não se aplica ao Reino Unido ou a nenhum país em que tais disposições estejam inconsistentes com a legislação local: A INTERNATIONAL BUSINESS MACHINES CORPORATION FORNECE ESTA PUBLICAÇÃO "NO ESTADO EM QUE SE ENCONTRA", SEM GARANTIA DE NENHUM TIPO, SEJA EXPRESSA OU IMPLÍCITA, INCLUINDO, MAS NÃO SE LIMITANDO, AS GARANTIAS IMPLÍCITAS DE NÃO-INFRAÇÃO, COMERCIALIZAÇÃO OU ADEQUAÇÃO A UM DETERMINADO PROPÓSITO. Alguns países não permitem a exclusão de garantias expressas ou implícitas em determinadas transações; portanto, esta disposição pode não se aplicar ao Cliente.

Essas informações podem conter imprecisões técnicas ou erros tipográficos. São feitas alterações periódicas nas informações aqui contidas; tais alterações serão incorporadas em futuras edições desta publicação. A IBM pode, a qualquer momento, aperfeiçoar e/ou alterar os produtos e/ou programas descritos nesta publicação a qualquer momento sem aviso prévio.

Referências nestas informações a Web sites que não sejam da IBM são fornecidas apenas por conveniência e não representam de forma alguma um endosso a estes Web sites. Os materiais contidos nesses Web sites não fazem parte dos materiais deste produto IBM e a utilização desses Web sites é de inteira responsabilidade do Cliente.

A IBM pode utilizar ou distribuir as informações fornecidas da forma que julgar apropriada sem incorrer em qualquer obrigação para com o Cliente.

Licenciados deste programa que desejam obter informações sobre este assunto com objetivo de permitir: (i) a troca de informações entre programas criados independentemente e outros programas (incluindo este) e (ii) a utilização mútua das informações trocadas, devem entrar em contato com:

Gerência de Relações Comerciais e Industriais da IBM Brasil Rational Software
IBM Corporation
Botafogo
Rio de Janeiro, RJ

Tais informações podem estar disponíveis, sujeitas a termos e condições apropriados, incluindo em alguns casos o pagamento de uma taxa.

O programa licenciado descrito nesta publicação e todo o material licenciado disponível são fornecidos pela IBM sob os termos do Contrato com o Cliente IBM, do Contrato de Licença de Programa Internacional IBM ou de qualquer outro contrato equivalente.

Quaisquer dados de desempenho contidos aqui foram determinados em ambientes controlados. Portanto, os resultados obtidos em outros ambientes operacionais poderão variar significativamente. Algumas medidas podem ter sido tomadas em sistemas em nível de desenvolvimento e não há garantia de que estas medidas serão as mesmas em sistemas disponíveis em geral. Além disso, algumas medidas podem ter sido estimadas por extrapolação. Os resultados reais podem ser diferentes. Os usuários deste documento devem verificar os dados aplicáveis para o ambiente específico.

As informações sobre produtos não-IBM foram obtidas junto aos fornecedores dos respectivos produtos, de seus anúncios publicados ou de outras fontes disponíveis publicamente. A IBM não testou estes produtos e não pode confirmar a precisão de seu desempenho, compatibilidade nem qualquer outra reivindicação relacionada a produtos não IBM. Dúvidas sobre os recursos de produtos não IBM devem ser encaminhadas diretamente a seus fornecedores.

Todas as declarações relacionadas aos objetivos e intenções futuras da IBM estão sujeitas a alterações ou cancelamento sem aviso prévio e representam apenas metas e objetivos.

Estas informações contêm exemplos de dados e relatórios usados nas operações diárias de negócios. Para ilustrá-los da forma mais completa possível, os exemplos podem incluir nomes de indivíduos, empresas, marcas e produtos. Todos estes nomes são fictícios e qualquer semelhança com nomes e endereços utilizados por uma empresa real é mera coincidência.

Licença de Direitos Autorais

Essas informações contêm programas de exemplos aplicativos na linguagem fonte, ilustrando as técnicas de programação em diversas plataformas operacionais. O Cliente pode copiar, modificar e distribuir estes programas de exemplo sem a necessidade de pagar à IBM, com objetivos de desenvolvimento, utilização, marketing ou distribuição de programas aplicativos em conformidade com a interface de programação de aplicativo para a plataforma operacional para a qual os programas de exemplo são criados. Esses exemplos não foram completamente testados em todas as condições. Portanto, a IBM não pode garantir ou implicar confiabilidade, manutenção, ou função destes programas. Os programas de amostra são fornecidos "NO ESTADO EM QUE SE ENCONTRAM", sem garantia de qualquer tipo. A IBM não será responsabilizada por qualquer dano decorrente do uso dos programas de amostra.

Reconhecimentos de Marca Registrada

IBM, o logotipo IBM e ibm.com são marcas ou marcas registradas da International Business Machines Corp., registradas em várias jurisdições no mundo todo. Outros nomes de produtos e serviços podem ser marcas registradas da IBM ou de outras empresas. Uma lista atual das marcas registradas da IBM está disponível na Web em www.ibm.com/legal/copytrade.shtml.

Rational é uma marca registrada da International Business Machines Corporation e da Rational Software Corporation nos Estados Unidos e/ou em outros países.

Intel e Pentium são marcas registradas da Intel Corporation nos Estados Unidos e/ou em outros países.

Microsoft, Windows e o logotipo Windows são marcas ou marcas registradas da Microsoft Corporation nos Estados Unidos e/ou em outros países.

Java e todas as marcas e logotipos baseados em Java são marcas ou marcas registradas da Sun Microsystems, Inc. nos Estados Unidos e em outros países.

UNIX é uma marca registrada da The Open Group nos Estados Unidos e em outros países.

Comentários do Leitor

IBM Rational Developer for System z
Guia do Desenvolvedor do Common Access Repository Manager
8.5

Publicação N° S517-9992-06

Neste formulário, faça-nos saber sua opinião sobre este manual. Utilize-o se encontrar algum erro, ou se quiser externar qualquer opinião a respeito (tal como organização, assunto, aparência...) ou fazer sugestões para melhorá-lo.

Para pedir publicações extras, fazer perguntas ou tecer comentários sobre as funções de produtos ou sistemas IBM, fale com o seu representante IBM.

Quando você envia seus comentários, concede direitos, não exclusivos, à IBM para usá-los ou distribuí-los da maneira que achar conveniente, sem que isso implique em qualquer compromisso ou obrigação para com você.

Não se esqueça de preencher seu nome e seu endereço abaixo, se deseja resposta.

Comentários:

Nome

Endereço

Companhia ou Empresa

Telefone

IBM Brasil - Centro de Traduções
Rodovia SP 101 Km 09
CEP 13185-900
Hortolândia,
SP

Índice Remissivo

A

- acesso a clusters do VSAM 77
- Ações Customizadas 56
- ações genéricas 3
- alocação de memória 6, 55
- aritmética de ponteiro 55
- armazenando resultados 80

B

- buffers de caracteres 8

C

- C para COBOL
 - passando valores 50
- CAF
 - tipos de objeto
 - ação 63
 - campo 64
 - parâmetro 62
 - RAM 61
 - valor de retorno 62
 - usando para customizar a API do RAM 61
- CARMA
 - introdução 1
 - metadados definidos 22
- COBOL para C
 - dados, passando 53
- códigos de retorno 9, 115
- compilando
 - cliente CARMA 79
- compilando um RAM 13
- conceitos gerais
 - alocação de memória 6
 - buffers de caracteres 8
 - códigos de retorno 9
 - conteúdo de membro 8
 - criação de log 9
 - efetuando o registro de entrada 6
 - efetuando o registro de saída de 6
 - parâmetros customizados 10
 - procurando 5
 - valores de retorno 10
- Construção, PDS 13
- Construção, PDS/E 14
- construção de PDS 13
- construção do PDS/E 14
- conteúdo de membro 8
- copyFromExternal 39
- copyToExternal 39
- CRADEF 70
- CRARAMCM 13
- CRASTRS 73
- createContainer 33, 97
- createMember 32, 95
- criação
 - registros VSAM 70
- criação de log 9, 84

- criação de log (*continuação*)
 - definição 20
- Criar/Excluir 32
- customizando
 - API do RAM 61

D

- definindo
 - RAM para CARMA 19
- delete 34, 98
- Depuração 59
- Descriptor 20
- desenvolvendo
 - cliente CARMA 79
 - modelo de RAM 64
- desenvolvendo um RAM 13
- desenvolvimento do RAM
 - utilizando o COBOL 48
- Desreferenciamento, Evitando 52

E

- efetuando o registro de entrada 6
- efetuando o registro de saída de 6
- estrutura de dados
 - Action 82
 - Action2 82
 - Campo 84
 - DescriptorWithInfo 81
 - descritor 81
 - Field2 84
 - KeyValPair 82
 - MemberDescriptorWithInfo 81
 - parameter 83
 - parameter2 83
 - RAMRecord 80
 - RAMRecord2 80
 - returnValue 83
 - returnValue2 83
- estruturas de dados
 - Descriptor 20
 - KeyValPair 20
 - predefinidas 20
- estruturas de dados, predefinidas
 - cliente 80
- estruturas de dados predefinidas 20
 - cliente 80
- executando o cliente CARMA 79
- exportando funções 20
- extensão de arquivo especificada pelo RAM 85
- extensão do arquivo
 - especificada pelo cliente 23
 - especificada pelo RAM 22
 - Herança 23
 - sugerida pelo RAM 22
- extractBinMember 40, 102
- extractMember 34, 98
- extrair para externo 39

F

- Finalização, anormal 59
- formato de registro
 - CRADEF 70
 - CRASTRS 73
- funções
 - criação de log 20
 - Criar/Excluir 95
 - Estado 24
 - exportando 20
 - metadados 41
 - procurando 25, 90
 - state 87
 - transferência de arquivo 34, 98
- Funções Criar/Excluir
 - createContainer 33, 97
 - createMember 32, 95
 - delete 34, 98
- funções de estado
 - getRAMList 88
 - getRAMList2 88
 - initCarma 87
 - initRAM 24, 88
 - reset 25, 89
 - terminateCarma 89
 - terminateRAM 25, 89
- funções de metadados
 - getAllMemberInfo 41, 103
 - getFieldsData 104
 - getFieldsData2 105
 - getMemberInfo 42, 105
 - updateMemberInfo 43, 106
- funções de navegação
 - Criar/Excluir 32
 - getContainerContents 30, 94
 - getContainerContentsWithInfo 31, 94
 - getInstances 25, 90
 - getInstancesWithInfo 26, 90
 - getMembers 27, 91
 - getMembersWithInfo 28, 92
 - isMemberContainer 30, 93
- funções de transferência de arquivo
 - binário, extractBinMember 102
 - binário, putBinMember 102
 - extractMember 34, 98
 - putMember 37, 100
 - transferência de arquivo binário 40
 - extractBinMember 40
 - putBinMember 41

G

- gerenciadores de acesso a repositório
 - Veja* RAM
- getAllMemberInfo 41, 103
- getContainerContents 30, 94
- getContainerContentsWithInfo 31, 94
- getFieldsData 104
- getFieldsData2 105
- getInstances 25, 90

getInstancesWithInfo 26, 90
getMemberInfo 42, 105
getMembers 27, 91
getMembersWithInfo 28, 92
getRAMList 88
getRAMList2 88

H

herança de extensão de arquivo 23
hierarquia do SCM 5

I

IDs de ação 117
IDs versus nomes 20
INFILE 13
initCarma 87
initRAM 24, 88
isMemberContainer 30, 93

K

KeyValPair 20

L

loais de arquivos de amostra 3
localizando
arquivos de amostra 3

M

matriz de caracteres bidimensional 7
matriz unidimensional 6
metadados
definidos pelo CARMA 85
metadados, definidos 22
metadados definidos 22
metadados definidos pelo CARMA 85
métodos
módulo de utilitários
utilCloseMemberList 15
utilCopyPDStoPDS 17
utilCopyPDStoSDS 17
utilCopySDStoPDS 17
utilCopySDStoSDS 17
utilExtractMemberClose 19
utilExtractMemberInit 18
utilExtractMemberRec 19
utilGetAllMemberInfo 15
utilGetAllPDSInfo 16
utilGetMemberInfo 16
utilGetNextMember 15
utilInitMemberList 14
utilPutMemberClose 18
utilPutMemberInit 17
utilPutMemberRec 18
utilPutMemberRecs 18
utilSetMemberInfo 16
modelo de RAM 64
módulo de utilitários
métodos
utilCloseMemberList 15
utilCopyPDStoPDS 17

módulo de utilitários (*continuação*)

métodos (*continuação*)

utilCopyPDStoSDS 17
utilCopySDStoPDS 17
utilCopySDStoSDS 17
utilExtractMemberClose 19
utilExtractMemberInit 18
utilExtractMemberRec 19
utilGetAllMemberInfo 15
utilGetAllPDSInfo 16
utilGetMemberInfo 16
utilGetNextMember 15
utilInitMemberList 14
utilPutMemberClose 18
utilPutMemberInit 17
utilPutMemberRec 18
utilPutMemberRecs 18
utilSetMemberInfo 16
módulo de utilitários, RAM 14

N

níveis de rastreio 9
nomes versus IDs 20

O

operações, outras
bloqueio 106
checkout 108
getCAFDData 110
getCAFDData2 110
getVersionList 111
performAction 109
registro de entrada 108
unlock 107
operações, ponteiro 54
operações de membros, outras
check_in 45
check_out 45
getVersionList 47
lock 44
performAction 46
unlock 44
operações de ponteiro 54
operações não suportadas 21
operações suportadas 3
OUTFILE 13
outras operações
bloqueio 106
checkout 108
getCAFDData 110
getCAFDData2 110
getVersionList 111
performAction 109
registro de entrada 108
unlock 107
outras operações de membros
check_in 45
check_out 45
getVersionList 47
lock 44
performAction 46
unlock 44

P

parâmetros customizados 10, 21, 85
Parâmetros customizados 56
Ponteiros nulos 59
procurando 5
putBinMember 41, 102
putMember 37, 100

R

RAM
amostras 2
compilando 13
definindo para o CARMA 19
desenvolvendo 13
estruturas de dados predefinidas 20
extensão de arquivo especificada 22
módulo de utilitários 14
padrão de função 13
RAM COBOL
divisão de procedimento,
definindo 50
estrutura de programa 49
finalizando o programa 50
ID do programa, codificando 49
seção de ligação 49
RAM customizado 64
RAM SAMP 75
RAMs de amostra 2
registros VSAM
RAM SAMP 75
reset 25, 89
Retornos customizados 57

S

SYSDEFSD 13
SYSLIB 13

T

terminateCarma 89
terminateRAM 25, 89
transferência de arquivo binário 40

U

updateMemberInfo 43, 106
utilCloseMemberList 15
utilCopyPDStoPDS 17
utilCopyPDStoSDS 17
utilCopySDStoPDS 17
utilCopySDStoSDS 17
utilExtractMemberClose 19
utilExtractMemberInit 18
utilExtractMemberRec 19
utilGetAllMemberInfo 15
utilGetAllPDSInfo 16
utilGetMemberInfo 16
utilGetNextMember 15
utilInitMemberList 14
utilPutMemberClose 18
utilPutMemberInit 17
utilPutMemberRec 18
utilPutMemberRecs 18

utilSetMemberInfo 16

V

valores de retorno 10, 21, 85

variáveis, compartilhadas 55

variáveis compartilhadas 55



Número do Programa: 5724-T07

Impresso no Brasil

S517-9992-06

